IIT Ropar

Design and Implementation of Decision Tree Algorithm Using Verilog HDL

Department of Computer Science and Engineering

Team Members

Harsh Modi	2023CSB1122
Aditya Gupta	2023CSB1095
Arth	2023CSB1100
Aamod Jain	2023CSB1092
Raghav Jha	2023CSB1149
Pratham Garg	2023CSB1147

Contents

List of Figures

1. Introduction

This document describes a hardware-optimized Decision Tree classifier implemented in synthesizable Verilog RTL. The design features parameterized depth and feature dimensionality, enabling flexible customization. The architecture is organized into a three-tier structure for efficient operation:

- **DecisionTreeNodes**: Responsible for performing threshold comparisons at each node of the tree.
- **TreeController**: Manages traversal optimization, ensuring efficient path selection during classification.
- **Top-Level Integration**: Combines the components into a cohesive system.

Key Features

- **8-bit Quantized Processing**: All computations are performed using 8-bit quantized data for resource efficiency.
- **Automated Binary Tree Indexing**: Supports up to $2^{\text{DEPTH}} - 1$ nodes for rapid indexing and traversal.
- **Pipelined Execution**: Implements a pipelined architecture to improve throughput and reduce latency.

System Validation and Testing

The system includes a CSV-driven testbench with the following features:

- Real-time accuracy monitoring.
- Error logging for detailed analysis of misclassifications.
- Comprehensive performance metrics to assess the design's efficiency.

This approach enables efficient validation and is optimized for FPGA/ASIC deployment in high-performance classification applications.

1.1 Background and Motivation

The Decision Tree-based classifier is a hardware-accelerated system designed for efficient classification tasks. By implementing the decision tree logic directly in hardware, this system achieves real-time performance with low latency, making it suitable for embedded systems, IoT devices, and machine learning applications with stringent performance requirements.

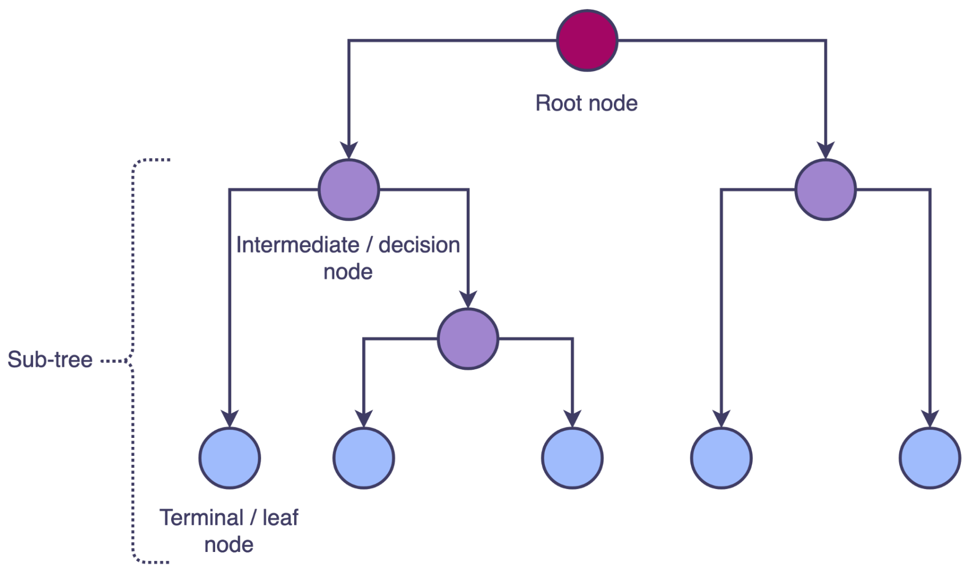


Figure 1.1: Decision Tree Overview

1.2 Project Objectives

1. To design a hardware-based decision tree classifier that processes feature inputs and outputs classification results in real time.
2. To implement modular Verilog code for scalability across varying tree depths and feature counts.
3. To test and validate the functionality of the system with real-world test cases.

1.3 Methodology

The project follows a systematic approach, including design, implementation, and testing.

1. **Design:**
   - Break down the decision tree into nodes, each responsible for a specific comparison and decision.
   - Implement a controller to handle traversal logic and manage the state of the tree during evaluation.
2. **Implementation:**
   - Write modular Verilog code for `DecisionTreeNode`, `TreeController`, and `DecisionTree`.
   - Ensure parameterization to support variable depths, thresholds, and feature counts.
3. **Testing:**
   - Use a testbench to simulate and verify the design using test cases from a CSV file.
   - Evaluate accuracy and identify misclassified cases for debugging.

2. Theoretical Foundation

A decision tree is a flowchart-like structure used to make decisions or predictions. It consists of nodes representing decisions or tests on attributes, branches representing the outcome of these decisions, and leaf nodes representing final outcomes or predictions. Each internal node corresponds to a test on an attribute, each branch corresponds to the result of the test, and each leaf node corresponds to a class label or a continuous value.

2.1 Decision Tree Algorithm

Decision trees are powerful tools for supervised learning tasks, designed to partition the feature space into regions associated with distinct output categories. The decision-making process relies on recursive binary splitting, where each node in the tree represents a decision based on a single feature and a threshold value. Below, we elaborate on the mathematical foundation and considerations for hardware mapping of decision trees.

2.1.1 Mathematical Foundation

The decision tree algorithm can be described formally as follows:

For a feature space X and output space Y:

- **Decision function**: $d(x) : X \to \{0, 1\}$
- **Threshold function**:
$$t(x) = \begin{cases} 1 & \text{if } x \geq \theta \\ 0 & \text{otherwise} \end{cases}$$
- **Classification function**: $C(x) : X \to Y$

2.1.2 Hardware Mapping Considerations

To implement the decision tree algorithm in hardware, several considerations must be addressed for efficient and accurate operation:

- **Threshold comparison**: Each node performs a comparison between a feature value and its threshold. This can be efficiently implemented using dedicated comparator units. These comparators operate at high speeds to ensure minimal latency.
- **Tree traversal**: A state machine manages the traversal through the tree. It determines the current node based on the outputs of the comparator units, dynamically updating the traversal path until a leaf node is reached.
- **Feature selection**: The features used for comparison at each node are selected using a multiplexed input system. Multiplexers ensure the correct feature is routed to the comparator based on the node being evaluated.
- **Classification storage**: Leaf node classifications are stored in distributed memory elements. These memory blocks are accessed only when traversal reaches a leaf node, enabling efficient retrieval of the final classification result.

2.2 Hardware Acceleration Principles

To maximize the performance of decision tree inference on hardware, various acceleration techniques are applied. These principles leverage the inherent parallelism and low-level optimizations of hardware architectures.

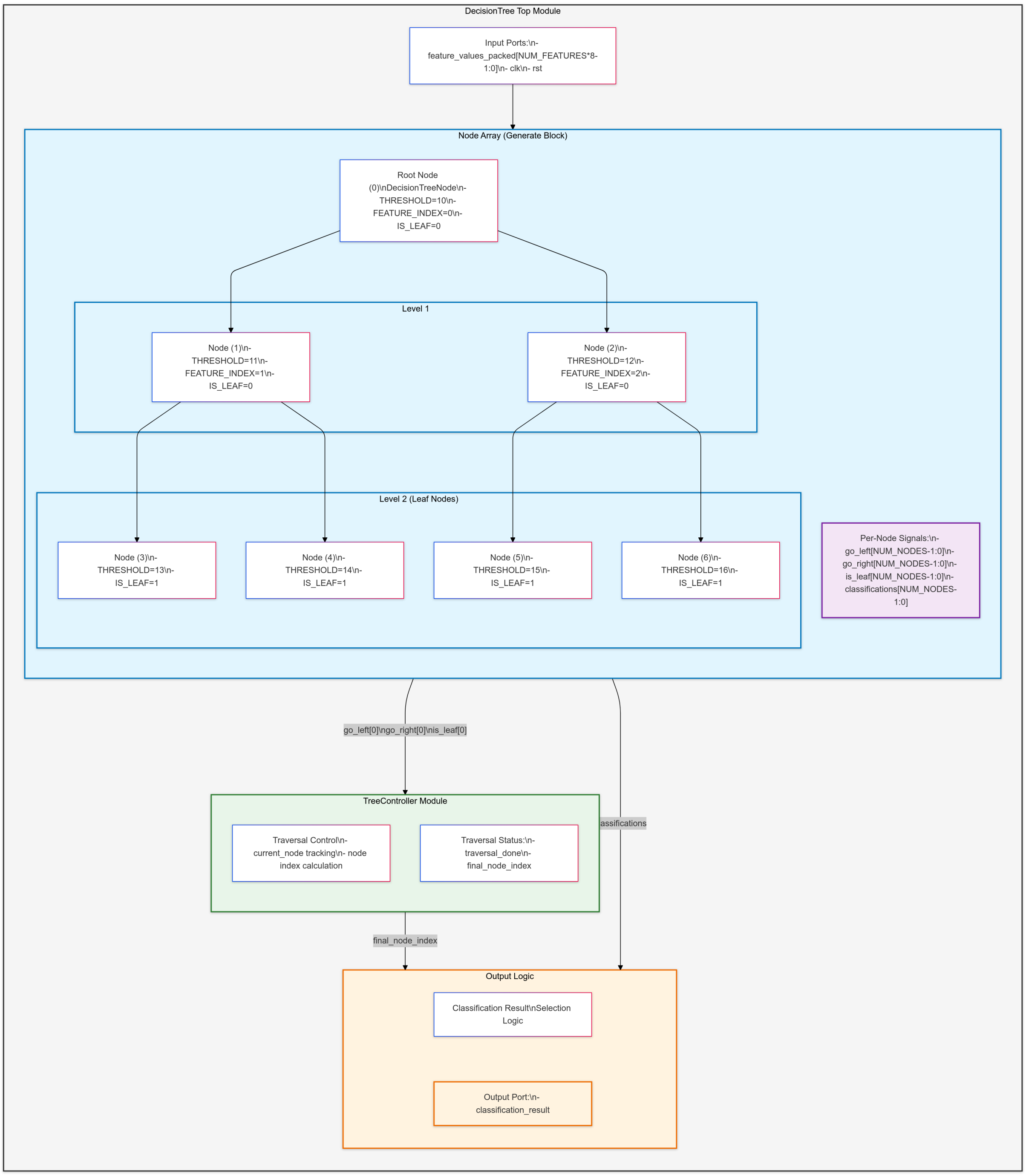
2.2.1 Parallelization Opportunities

- **Concurrent node evaluation**: Hardware allows multiple nodes to evaluate feature comparisons simultaneously. This is particularly effective in shallower parts of the tree where nodes can operate in parallel to expedite traversal decisions.
- **Pipelined Feature Processing**: By introducing a pipeline architecture, feature comparisons and traversal decisions are processed in stages. This approach reduces bottlenecks, ensuring that the hardware operates at maximum throughput.
- **Parallel Threshold Comparison**: For nodes evaluating the same feature or sharing dependencies, parallel comparator units are employed. This technique minimizes the delay introduced by sequential evaluations.

2.2.2 Memory Architecture

- **Distributed Threshold Storage**:Threshold values for each node are stored in distributed memory blocks located near the comparator units. This reduces access latency and facilitates faster comparisons.
- **Local Classification Memory**:Leaf node classifications are stored locally within the hardware fabric, ensuring minimal delay when the traversal reaches the final node. This approach also improves scalability for larger trees.
- **State Encoding Optimization**:The state machine responsible for traversal uses optimized state encoding techniques to reduce the number of bits required for state representation. This improves hardware efficiency and minimizes resource usage.

3. System Architecture



DecisionTree Top Module
Input Ports:\n- feature_values_packed[NUM_FEATURES*8-1:0]\n- clk\n- rst
Node Array (Generate Block)
Root Node (0)\nDecisionTreeNode\n- THRESHOLD=10\n- FEATURE_INDEX=0\n- IS_LEAF=0
Level 1
Node (1)\n- THRESHOLD=11\n- FEATURE_INDEX=1\n- IS_LEAF=0
Node (2)\n- THRESHOLD=12\n- FEATURE_INDEX=2\n- IS_LEAF=0
Level 2 (Leaf Nodes)
Node (3)\n- THRESHOLD=13\n- IS_LEAF=1
Node (4)\n- THRESHOLD=14\n- IS_LEAF=1
Node (5)\n- THRESHOLD=15\n- IS_LEAF=1
Node (6)\n- THRESHOLD=16\n- IS_LEAF=1
Per-Node Signals:\n- go_left[NUM_NODES-1:0]\n- go_right[NUM_NODES-1:0]\n- is_leaf[NUM_NODES-1:0]\n- classifications[NUM_NODES-1:0]
go_left[0]\ngo_right[0]\nis_leaf[0]
assifications
TreeController Module
Traversal Control\n- current_node tracking\n- node index calculation
Traversal Status:\n- traversal_done\n- final_node_index
final_node_index
Output Logic
Classification Result\nSelection Logic
Output Port:\n- classification_result

Figure 3.1: High-Level System Architecture Overview

3.1 High-Level Organization

Key components include:

1. **Feature Input Interface**
   - **Data Synchronization**:Aligning incoming feature data with the system clock to avoid timing mismatches.
   - **Input Buffering**:Temporarily storing feature values to accommodate variations in data arrival rates and maintain smooth processing.
   - **Feature Selection**:Isolating and routing the appropriate feature values to specific nodes in the decision tree for evaluation.
2. **Decision Tree Core**
   - **Node Array**:A collection of decision nodes, each responsible for evaluating a specific feature and threshold.
   - **Threshold Comparators**:Hardware comparators that evaluate feature values against predetermined thresholds at each node.
   - **Classification Storage**:Memory elements that store the classification results for leaf nodes in the tree.
3. **Control Unit**
   - **State Machine**:A finite state machine (FSM) that governs the system's operational flow, from loading input to outputting results.
   - **Timing Controller**:Ensures that each operation occurs within a specified clock cycle, maintaining real-time performance.
   - **Error Handler**:Detects and manages faults or inconsistencies, such as invalid inputs or hardware malfunctions.
4. **Output Interface**
   - **Classification Register**:Stores the final classification result for the current set of feature inputs.

- **Status Indicators**: Provide real-time feedback on the system's operational status, such as ”processing,” ”idle,” or ”error.”
- **Error Flags**:Signal any faults or issues encountered during processing, enabling corrective actions.

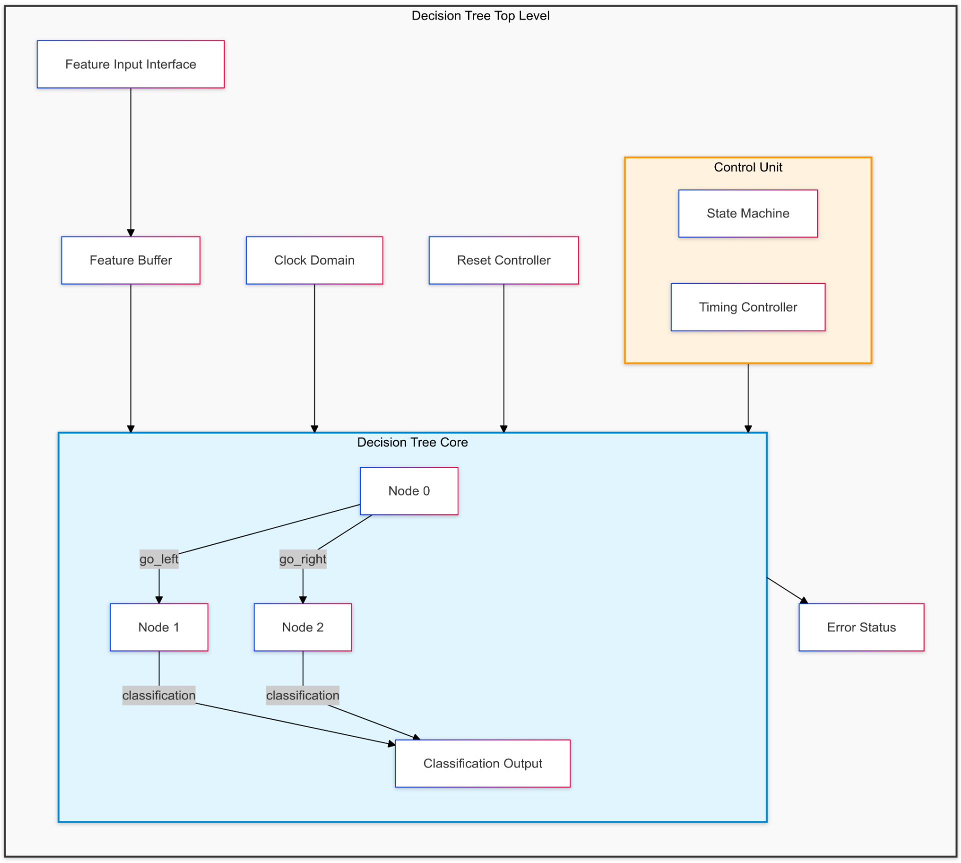


Figure 3.2: High-Level System Architecture

3.2 Node Architecture

Input Processing

- **Feature Buffer:** Temporarily holds the feature values for the current node, ensuring seamless data availability during processing.
- **Synchronization Logic:** Aligns the input data with the system's operational clock for precise and error-free processing.
- **Valid Data Detection:** Verifies that the incoming feature values meet required validity criteria before proceeding to comparison.

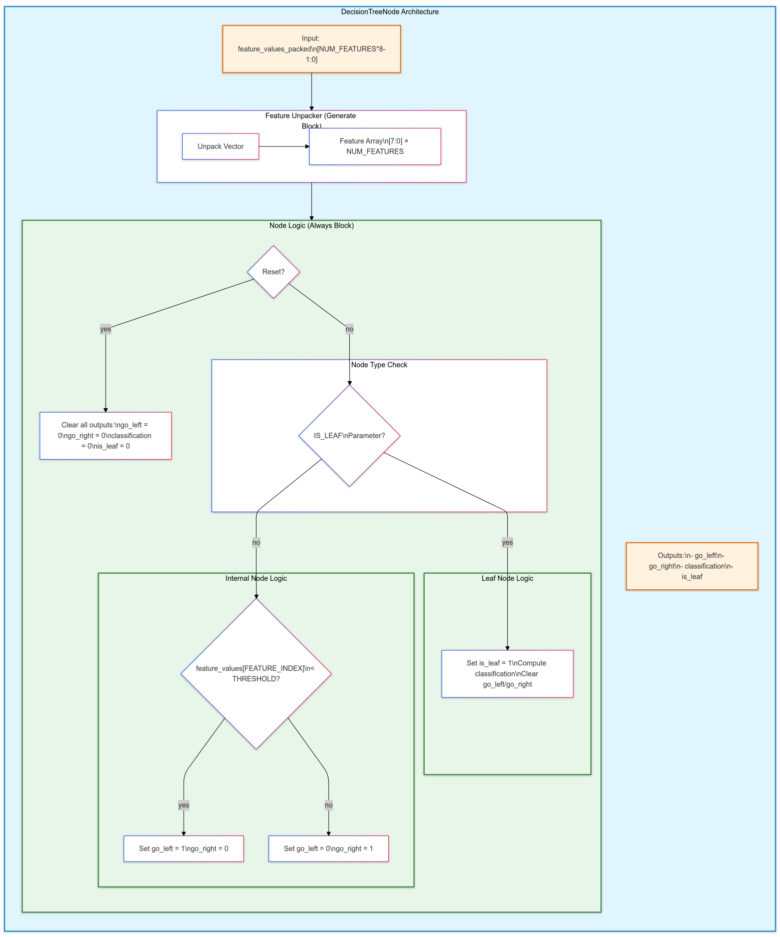

DecisionTreeNode Architecture
Input: feature_values_packed\n[NUM_FEATURES*8-1:0]
Feature Unpacker (Generate Block)
Unpack Vector
Feature Array\n[7:0] × NUM_FEATURES
Node Logic (Always Block)
Reset?
yes
no
Clear all outputs:\ngo_left = 0\ngo_right = 0\nclassification = 0\nis_leaf = 0
Node Type Check
IS_LEAF\nParameter?
no
yes
Internal Node Logic
Leaf Node Logic
feature_values[FEATURE_INDEX]\n< THRESHOLD?
Set is_leaf = 1\nCompute classification\nClear go_left/go_right
yes
no
Set go_left = 1\ngo_right = 0
Set go_left = 0\ngo_right = 1
Outputs:\n- go_left\n- go_right\n- classification\n- is_leaf

Figure 3.3: Detailed Node Architecture Design

Comparison Unit

- **8-bit Magnitude Comparator:** Compares the feature value against the node's threshold to decide the traversal path.
- **Threshold Register:** Stores the threshold value specific to the node, programmable for different decision tree configurations.
- **Result Latch:** Temporarily holds the comparison result, ensuring stable outputs for subsequent processing.

Control Logic

- **Next Node Selection:** Routes control to the appropriate child node (left or right) based on the comparison outcome.
- **Classification Storage:** Outputs a classification result if the node is a leaf, bypass-

ing further traversal.

- **Error Detection:** Identifies and flags any issues, such as invalid thresholds or data anomalies, at the node level.

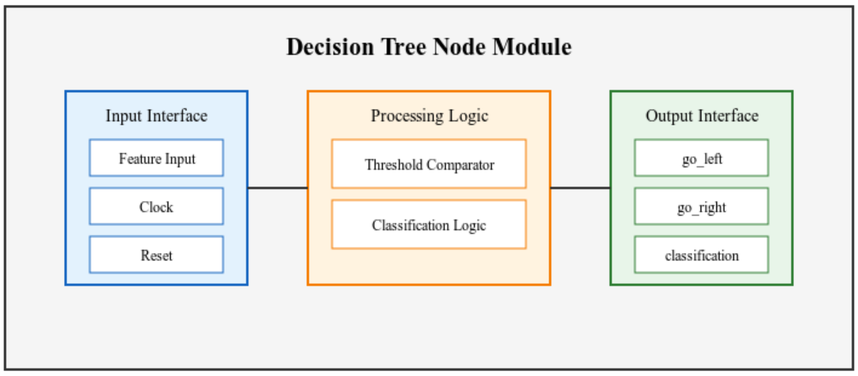


Figure 3.4: Node Architecture

3.3 State Machine Design

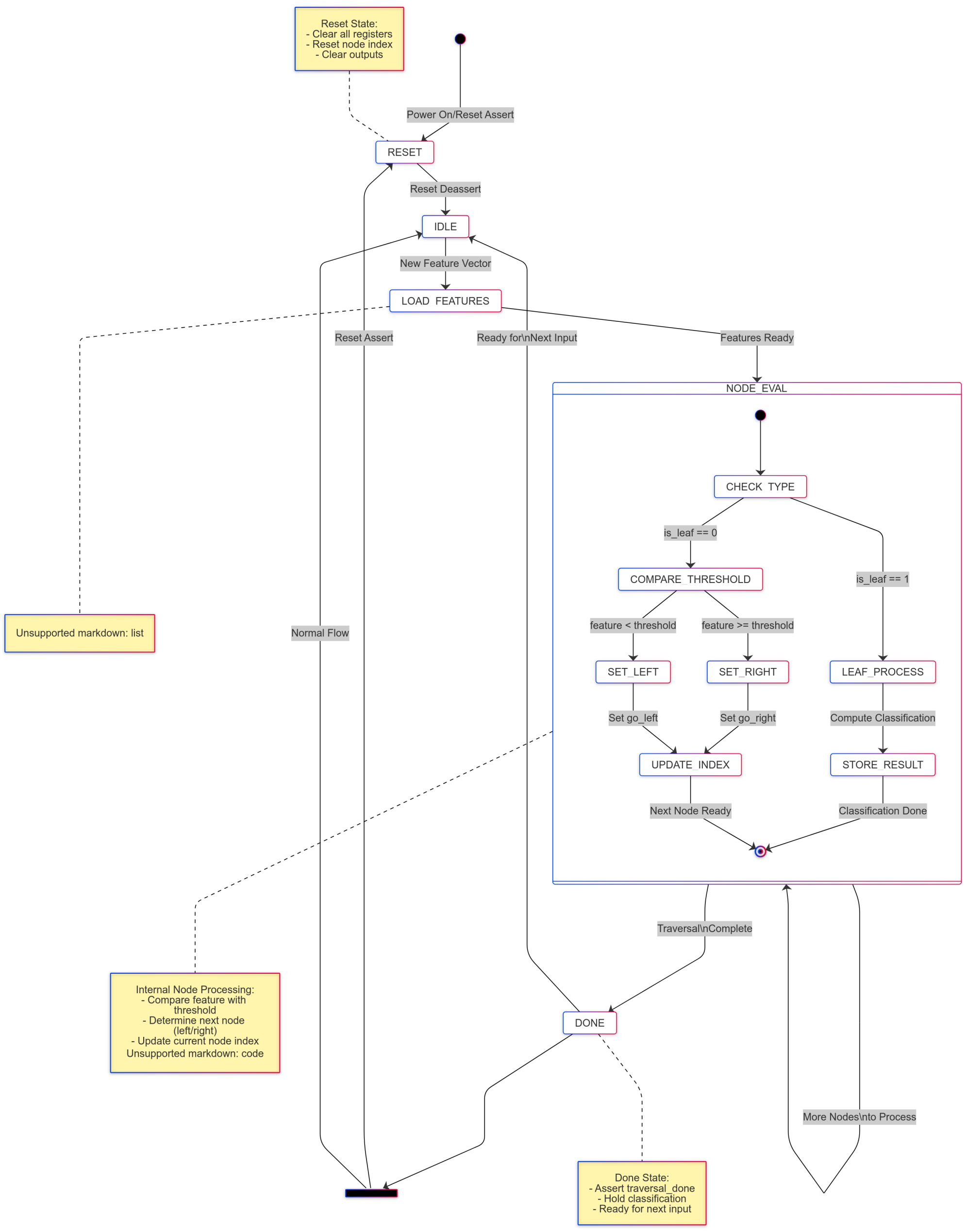


Figure 3.5: Decision Tree State Machine Flow

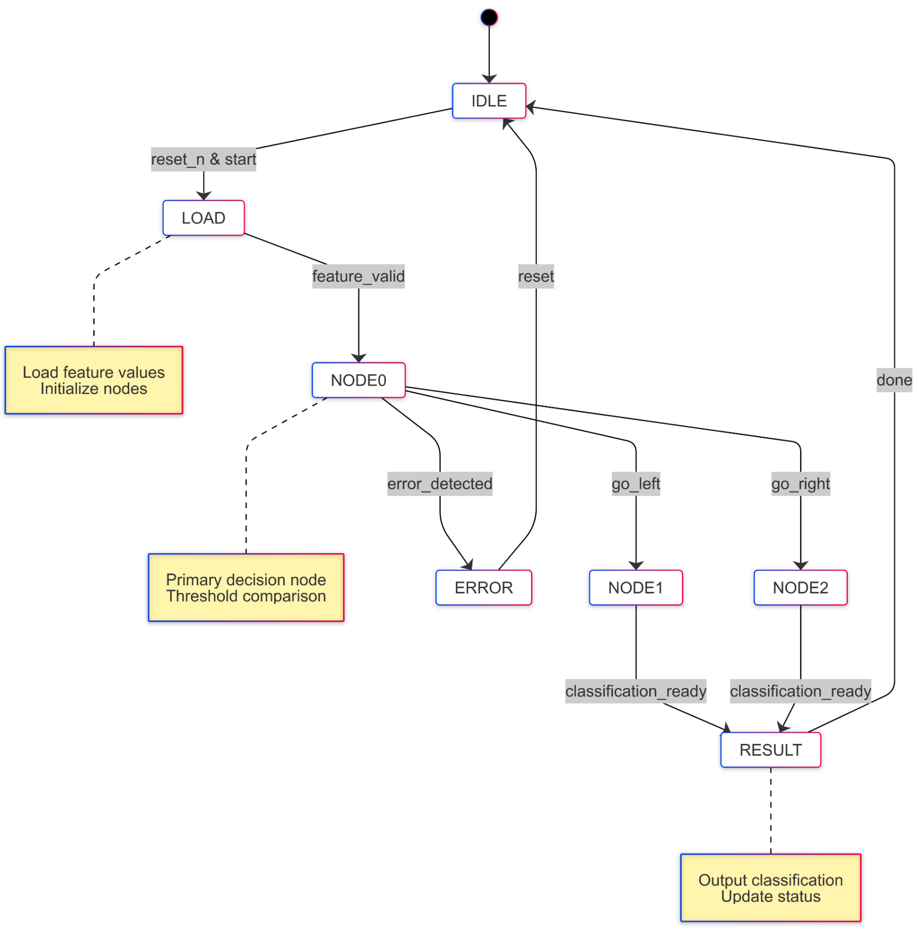


Figure 3.6: State Machine Diagram

4. Hardware Implementation

4.1 RTL Design

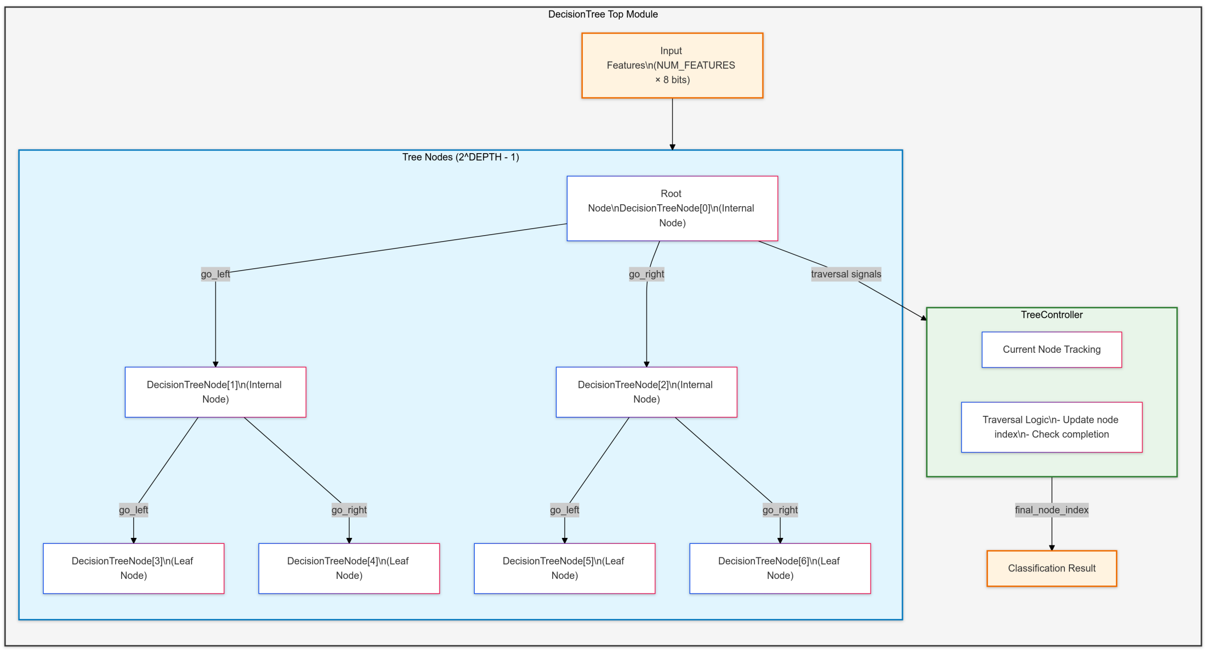

DecisionTree Top Module
Input Features\n(NUM_FEATURES × 8 bits)
Tree Nodes (2^DEPTH - 1)
Root Node\nDecisionTreeNode[0]\n(Internal Node)
go_left
go_right
traversal signals
DecisionTreeNode[1]\n(Internal Node)
DecisionTreeNode[2]\n(Internal Node)
go_left
go_right
go_left
go_right
DecisionTreeNode[3]\n(Leaf Node)
DecisionTreeNode[4]\n(Leaf Node)
DecisionTreeNode[5]\n(Leaf Node)
DecisionTreeNode[6]\n(Leaf Node)
TreeController
Current Node Tracking
Traversal Logic\n- Update node index\n- Check completion
final_node_index
Classification Result

Figure 4.1: Decision Tree Implementation Flow

4.2 RTL Design

4.2.1 Decision Tree Node (`DecisionTreeNode`)

Each node in the decision tree evaluates a feature value against a threshold and determines the next step in the traversal—either moving to the left child, the right child, or producing a classification if it is a leaf node.

Hardware Design

- **Input Interface**: Takes packed feature values, clock, reset signals.
- **Processing Logic**:

  - Extracts specific feature values.
  - Compares the feature value with the threshold.
  - Generates control signals (`go_left`, `go_right`) or outputs a classification if it is a leaf.

- **Output Interface**: Sends traversal or classification signals.

Pseudocode:

```
Input: feature_values_packed, clk, rst
Output: go_left, go_right, classification, is_leaf

1. On reset:
   Set go_left, go_right, classification, and is_leaf to 0.

2. Unpack feature values from packed input.

3. If IS_LEAF:
     is_leaf <= 1
     classification <= (feature_value[FEATURE_INDEX] <
        THRESHOLD) ? 1 : 0
     go_left <= 0
     go_right <= 0
   Else:
     is_leaf <= 0
     If feature_value[FEATURE_INDEX] < THRESHOLD:
        go_left <= 1
        go_right <= 0
     Else:
        go_left <= 0
        go_right <= 1
```

Listing 4.1: DecisionTreeNode Pseudocode

4.2.2 Control Unit Implementation: Tree Controller (`TreeController`)

This module controls the traversal of the decision tree. It maintains the current node index and updates it based on the traversal signals (`go_left` and `go_right`). It also detects when the traversal reaches a leaf node.

Hardware Design

- **Input Interface**: Takes traversal signals (`go_left`, `go_right`), `is_leaf`, clock, and reset.
- **Processing Logic**:
  - Maintains a register to track the current node.
  - Updates the node index based on traversal signals.
  - Signals traversal completion when a leaf is reached.
- **Output Interface**: Outputs the index of the final node and traversal completion signal.

Pseudocode:

```
Input: clk, rst, go_left, go_right, is_leaf
Output: traversal_done, final_node_index

1. On reset:
   current_node <= 0
   traversal_done <= 0
   final_node_index <= 0

2. On clock edge:
   If is_leaf:
      traversal_done <= 1
      final_node_index <= current_node
   Else If go_left:
      current_node <= (current_node << 1) + 1
   Else If go_right:
        current_node <= (current_node << 1) + 2
```

Listing 4.2: TreeController Pseudocode

4.2.3 Decision Tree Top-Level Module (`DecisionTree`)

The top-level module integrates all the decision tree nodes and the controller to form a complete decision tree classifier.

Hardware Design

- **Input Interface**: Takes packed feature values, clock, and reset.
- **Processing Logic**:
  - Instantiates and connects all `DecisionTreeNode` modules.
  - Connects the root node's outputs to the `TreeController`.
  - Uses the traversal completion signal and final node index to fetch the classification result.
- **Output Interface**: Outputs the final classification result.

Pseudocode:

```
Input: feature_values_packed, clk, rst
Output: classification_result

1. Instantiate NUM_NODES 'DecisionTreeNode' modules.

2. Connect outputs of each node to corresponding child nodes.

3. Connect root node traversal signals to 'TreeController'.

4. On clock edge:
   If traversal_done:
      classification_result <= classifications[final_node_index]
```

Listing 4.3: DecisionTree Top-Level Pseudocode

5. Optimization Techniques Used

5.1 Parameterization and Configurability

- **Dynamic Parameters**: Utilizes parameters such as `THRESHOLD`, `FEATURE_INDEX`, `NUM_FEATURES`, and `DEPTH` to allow flexible configuration of the decision tree structure without modifying the core logic.
- **`localparam` Usage**: Employs `localparam` for internal constants like `NUM_NODES` and `LEAF_START`, ensuring consistent and error-free calculations based on tree depth.

5.2 Bit Packing for Feature Representation

- **Packed Feature Vector**: Consolidates multiple 8-bit feature values into a single wide input (`feature_values_packed`), reducing the number of I/O ports and simplifying data routing within the hardware.
- **Efficient Unpacking**: Uses generate blocks to systematically unpack the feature values, enabling parallel access and evaluation within each `DecisionTreeNode`.

5.3 Generate Blocks for Scalable Node Instantiation

- **Automated Module Generation**: Implements generate statements to instantiate a large number of `DecisionTreeNode` modules based on `NUM_NODES`, facilitating scalability to deep trees without manual repetition.
- **Conditional Leaf Node Assignment**: Dynamically assigns the `IS_LEAF` parameter during instantiation, allowing seamless differentiation between leaf and internal nodes within the same module framework.

5.4 Efficient Traversal Logic with Bit Manipulation

- **Binary Tree Navigation**: Leverages bitwise operations (shifting and addition) to calculate child node indices, which are more resource-efficient compared to arithmetic operations.
- **Minimal Bit-Width Allocation**: Utilizes the `$clog2` macro to determine the minimum number of bits required for node indexing, optimizing register sizes and conserving hardware resources.

5.5 Parallel Node Processing

- **Concurrent Evaluation**: All `DecisionTreeNode` instances operate in parallel, allowing simultaneous evaluation of multiple nodes and significantly accelerating the traversal process.
- **Scalable Parallelism**: Maintains high throughput regardless of tree depth by exploiting inherent hardware parallelism, making the design suitable for real-time classification tasks.

5.6 Comprehensive and Automated Testbench Design

- **CSV-Based Test Case Handling**: Reads and processes test cases from an external CSV file, enabling easy scalability and modification of test scenarios without altering the testbench code.
- **Automated Result Verification**: Compares the classification output against expected results automatically, providing detailed logs for misclassifications and statistical performance metrics to streamline verification.

6. Verification and Testing

6.1 Automated Testbench Structure

- **Clock Generation**: Produces a consistent clock signal to synchronize operations across all modules.
- **Reset Logic**: Initializes the system by asserting the reset signal, ensuring all modules start in a known state.

6.2 Automated Verification Mechanisms

- **Result Comparison**: Automatically compares the module's classification output against expected results, incrementing counters for correct and misclassified cases.
- **Detailed Logging**: Logs comprehensive information for each misclassified test case, including feature values and discrepancies between expected and actual results, facilitating efficient debugging.

6.3 Performance Metrics and Reporting

- **Accuracy Calculation**: Computes classification accuracy based on the number of correct and total test cases processed.
- **Comprehensive Reporting**: Generates a summarized report detailing total test cases, correct classifications, misclassifications, and overall accuracy upon completion of all tests.

6.4 Synchronization and Timing Control

- **Clock Edge Synchronization**: Aligns verification checks with the design's operational timing by waiting for specific clock edges, ensuring accurate and timely evaluation of results.

- **Traversal Completion Detection**: Monitors the `traversal_done` signal to determine when the classification result is ready, aligning result evaluation with traversal completion.

6.5 Test Infrastructure

6.5.1 Testbench Architecture

The testbench reads input test cases from a CSV file, feeds them to the `DecisionTree` module, and compares the outputs against expected results to calculate accuracy.

6.5.2 Hardware Setup for Testing

- **Input Interface**: Reads test cases from a file.
- **Processing Logic**:
  - Stimulates the `DecisionTree` module with feature values.
  - Checks classification results.
- **Output Interface**: Displays test case results and overall accuracy.

Pseudocode:

```
Input: test_cases.csv
Output: Accuracy Report

1. Open the test case file.

2. For each test case:
    Extract feature values and expected result.
    Feed feature values to the DecisionTree module.
    Wait for classification_result.
    Compare classification_result with expected result.
    Update correct_count, total_count, misclassified_count.
```

```

3. Display final statistics:
   Accuracy = (correct_count / total_count) * 100
```

Listing 6.1: Testbench Pseudocode

7. Results and Analysis

7.1 Accuracy Analysis

Depth	**Number of Features**	**Total Test Cases**	**Correct Classifications**	**Misclassified Ca**
3	7	30	17	13
4	25	1000	509	491
5	42	500	262	238
6	69	1000	504	496
7	254	500	266	234
8	269	500	259	241
9	696	500	267	233
10	20000	77	46	31

8. Conclusions and Future Work

8.1 Achievements

- **Scalable Design:** Utilized parameterization and `generate` blocks to seamlessly scale the decision tree to extensive depths without manual module instantiation.
- **Efficient Resource Utilization:** Implemented bit packing for feature representation and minimal register usage, optimizing hardware resource consumption.
- **Parallel Processing:** Enabled concurrent evaluation of all `DecisionTreeNode` instances, significantly accelerating classification throughput.
- **Early Traversal Termination:** Incorporated a traversal completion flag to halt processing upon reaching a leaf node, reducing latency and computational overhead.
- **Robust Verification:** Developed an automated testbench that reads external test cases, performs comprehensive result comparisons, and provides detailed logging for misclassifications.

8.2 Future Enhancements

- **Tree Pruning:** Implement algorithms to remove redundant branches, reducing tree depth and conserving hardware resources.
- **Floating-Point Support:** Extend feature handling to accommodate floating-point values for increased precision in classifications.
- **Pipelining:** Introduce pipeline stages within node processing to enhance throughput and enable higher operating frequencies.
- **Dynamic Threshold Adjustment:** Enable runtime updates of node thresholds to allow the decision tree to adapt to evolving data patterns.

9. References

[1] Harris, D. M., & Harris, S. L. (2022). *Digital Design and Computer Architecture: ARM Edition.*

[2] Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning.*

[3] IEEE Standard for Verilog Hardware Description Language (2017).