# **Description Of FloatingPoint Unit:**

## **Fp\_Wrapper:**

This fp\_wrapper module is a Verilog wrapper around a floating-point processing unit (FPU) module (fpnew\_top). Its primary purpose is to manage communication between an Arithmetic Processing Unit (APU) and the FPU, handling request and response channels to execute floating-point operations based on specified formats, operations, and rounding modes. Here's a detailed breakdown of each signal:

**Clock and Reset Signals**

* **clk\_i**: Input clock signal. Used to synchronize operations within the module.
* **rst\_ni**: Active-low reset signal. Resets internal states and initializes the module when 0.

**APU Side: Master Port**

* **apu\_req\_i**: Input request signal from the APU, indicating a new operation request. When high, it triggers the FPU to process the requested operation.
* **apu\_gnt\_o**: Output grant signal to the APU, signaling that the FPU is ready to receive the next operation when high.

**Request Channel**

* **apu\_operands\_i**: Input bus carrying operand data for the FPU operation. The width is determined by APU\_NARGS\_CPU and is organized as a multi-dimensional array ([APU\_NARGS\_CPU-1:0][31:0]). It carries up to APU\_NARGS\_CPU operands, each 32-bits wide.
* **apu\_op\_i**: Input bus specifying the operation type for the FPU. Its width is determined by APU\_WOP\_CPU. It includes fields for the operation type, whether it's a vector operation, and operation mode.
* **apu\_flags\_i**: Input flags indicating formats and rounding modes. The width of this bus is determined by APU\_NDSFLAGS\_CPU-5. The flags specify the destination and source formats, integer format, and rounding mode for the operation.

**Response Channel**

* **apu\_rvalid\_o**: Output signal indicating the validity of data on the response channel. It goes high when the FPU has a result ready.
* **apu\_rdata\_o**: Output bus containing the result of the FPU operation. It's a 32-bit bus ([31:0]).
* **apu\_rflags\_o**: Output flags bus containing additional status information related to the operation, such as exception flags. Its width is determined by APU\_NUSFLAGS\_CPU.

**Internal Signals and Configuration Parameters**

* **fpu\_op, fpu\_op\_mod, fpu\_vec\_op**: These signals are derived from apu\_op\_i and are used to specify the type, modification, and vector nature of the FPU operation.
* **fpu\_dst\_fmt, fpu\_src\_fmt, fpu\_int\_fmt, fp\_rnd\_mode**: Derived from apu\_flags\_i to specify floating-point and integer formats as well as the rounding mode.

**Local Parameters (FPU Configurations)**

* **FPU\_FEATURES**: This configuration parameter (structure) defines supported features, including width, vector support, NaN boxing, and format masks.
* **FPU\_IMPLEMENTATION**: Specifies implementation details like pipeline registers, unit types, and configuration of the pipeline.

**FPU Instance (fpnew\_top)**

The fpnew\_top instance is configured using FPU\_FEATURES and FPU\_IMPLEMENTATION, which define the capabilities of the FPU, such as supported formats and internal pipeline configurations. It connects to various signals in the fp\_wrapper module:

* **clk\_i, rst\_ni**: Clock and reset signals.
* **operands\_i**: Connected to apu\_operands\_i, providing operands for FPU calculations.
* **rnd\_mode\_i**: Specifies the rounding mode using a typecasted value from fp\_rnd\_mode.
* **op\_i, op\_mod\_i, src\_fmt\_i, dst\_fmt\_i, int\_fmt\_i, vectorial\_op\_i**: Signals specifying the operation type, mode, formats, and whether it’s a vector operation.
* **in\_valid\_i**: Connected to apu\_req\_i, marking the start of an operation.
* **in\_ready\_o**: Connected to apu\_gnt\_o, indicating readiness to receive a new operation.
* **result\_o**: Connected to apu\_rdata\_o, carrying the FPU operation result.
* **status\_o**: Connected to apu\_rflags\_o, carrying status flags such as exceptions.
* **out\_valid\_o**: Connected to apu\_rvalid\_o, indicating that a result is valid and ready.
* **busy\_o**: Not connected; could be used to indicate if the FPU is currently processing.

This wrapper provides a structured interface for initiating floating-point calculations with specified formats, operations, and rounding modes, using an external APU as the controlling entity. The FPU performs the requested operation and provides the results and status flags back to the APU

## **Fpnew\_Top:**

The `fpnew\_top` module is a top-level floating-point unit (FPU) designed to perform various floating-point operations, with configurations that allow for custom implementations, including support for SIMD (Single Instruction, Multiple Data) operations. Here’s a detailed breakdown of each input, output, and the overall working of this module.

### Input and Output Signals:

### Clock and Reset:

**clk\_i**: Input clock signal that synchronizes all internal operations.

**rst\_ni**: Active-low reset signal that initializes the module when asserted (low).

### Input Signals (for Operations):

**operands\_i::** Array of input operands for FPU operations. Organized as `[NUM\_OPERANDS-1:0][WIDTH-1:0]`, where each operand has a bit width defined by `WIDTH`.

**rnd\_mode\_i:** Rounding mode for the floating-point operation, defined by the `roundmode\_e` type.

**op\_i:** Specifies the type of floating-point operation to perform (e.g., addition, multiplication), defined by the `operation\_e` type.

**op\_mod\_i:** Operation modifier (used for additional operation specifications).

**src\_fmt\_i:** Specifies the format of the source operand (e.g., FP32, FP64), defined by `fp\_format\_e`.

**dst\_fmt\_i:** Specifies the format of the destination operand.

**int\_fmt\_i:** Specifies the integer format when integer operations are involved, defined by `int\_format\_e`.

**vectorial\_op\_i:** Boolean signal indicating whether the operation is vectorial (SIMD).

**tag\_i:** Input tag, a type-defined field that can be used to identify or track an operation.

**simd\_mask\_i:** Mask for SIMD operations, used to enable or disable specific lanes in SIMD mode.

**Handshake Inputs/Outputs (for Control)**

* **in\_valid\_i**: Input signal indicating that valid data is present for processing.
* **in\_ready\_o**: Output signal indicating that the module is ready to receive new data. Controlled internally based on availability and readiness of operation blocks.
* **flush\_i**: Input signal used to flush or reset the current operation(s).

**Output Signals (for Results)**

* **result\_o**: Result of the FPU operation. Its width is defined by WIDTH.
* **status\_o**: Status output, defined by status\_t, which typically holds exception flags or other status information related to the operation.
* **tag\_o**: Output tag, corresponding to tag\_i, to identify the result.
* **out\_valid\_o**: Output signal indicating that the result is valid and ready to be read.
* **out\_ready\_i**: Input signal from downstream logic indicating that it is ready to accept the result.
* **busy\_o**: Output signal indicating if the FPU is currently processing any operation.

**Module Parameters**

* **Features**: Defines various configurable features of the FPU, such as vector support, format support, and width.
* **Implementation**: Defines specific configurations for the FPU, like the number of pipeline registers and type of units (e.g., merged, parallel).
* **DivSqrtSel**: Specifies the division/square root unit type (options include PULP, TH32, THMULTI).
* **TagType**: Specifies the type for the tag signals, which can be logic or other types.
* **TrueSIMDClass** and **EnableSIMDMask**: Parameters controlling SIMD class and mask enablement.

**Working of the Module**

1. **Input Handling and Nan-Boxing**:
   * The in\_ready\_o signal is controlled based on the input handshake and whether the target operation group (based on op\_i) is ready.
   * NaN-boxing checks ensure that values fit the format requirements for each operation. When enabled, higher bits of the operand (outside the specified FP width) are checked for NaN-boxing.
2. **SIMD Mask Filtering**:
   * The simd\_mask is computed by filtering out unused lanes based on EnableSIMDMask. This mask is applied only if SIMD operations are enabled.
3. **Operation Block Generation**:
   * The module is structured to handle multiple operation groups (e.g., add, multiply, divide/square root). Each operation group is instantiated with specific parameters and handles one type of floating-point operation.
   * Each operation group block (fpnew\_opgroup\_block) receives relevant inputs (operands, format, etc.) and handles operation-specific processing. The handshake signals (in\_valid, in\_ready, etc.) control data flow between the wrapper and the operation blocks.
   * Each operation block produces outputs (result, status, and tag) and handshake signals (out\_valid, out\_ready), which are later arbitrated.
4. **Output Arbitration**:
   * A round-robin arbiter (rr\_arb\_tree) is used to select outputs from multiple operation blocks. This arbitration ensures that the output signals (result\_o, status\_o, and tag\_o) carry data from the currently granted operation block.
   * The out\_valid\_o signal indicates that the arbiter has a valid result ready for output. If multiple operation blocks have valid data simultaneously, the arbiter will cycle between them in a round-robin fashion.
5. **Busy Signal**:
   * The busy\_o signal is a logical OR of all opgrp\_busy signals from each operation block. It indicates if any of the operation blocks is still processing.

**Overall Purpose**

The fpnew\_top module coordinates floating-point operations, handling input data, selecting the appropriate operation block, processing the data, and managing the output. It includes features for SIMD support, rounding mode selection, format specification, and customizability to accommodate various FPU configurations. The module performs floating-point operations in parallel operation blocks and uses arbitration to manage outputs in a consistent manner, making it suitable for high-performance floating-point computations.

# **Fpnew\_opgroup\_block:**

The fpnew\_opgroup\_block module is a parameterized floating-point operation block, part of a larger floating-point unit (FPU) system. Its purpose is to handle operations based on format and type, process multiple formats and SIMD operations, and generate output results. Here’s an overview of the module, including its inputs, outputs, and functionality.

**Inputs and Outputs**

**Input Signals**

1. **clk\_i**: The clock input for synchronizing operations within the module.
2. **rst\_ni**: The reset signal, active low. Resets internal states of the module.
3. **operands\_i**: Array of operands to process. The width and number of operands are determined by the Width and NUM\_OPERANDS parameters.
4. **is\_boxed\_i**: Specifies if each operand is "NaN-boxed" (a mechanism for handling NaN values). The array is formatted per format and operand.
5. **rnd\_mode\_i**: Defines the rounding mode to be applied to floating-point results.
6. **op\_i**: Specifies the operation (e.g., add, subtract, multiply) to perform.
7. **op\_mod\_i**: Operation modifier, adjusting the specific behavior of the operation.
8. **src\_fmt\_i**: Source format (e.g., single, double precision) of the operation.
9. **dst\_fmt\_i**: Destination format to which results should be cast.
10. **int\_fmt\_i**: Specifies the integer format, if the operation involves integer representation.
11. **vectorial\_op\_i**: When high, signals that this is a vector operation rather than scalar.
12. **tag\_i**: A generic tag input that allows data associated with the operation to be tagged, often for tracking in a larger system.
13. **simd\_mask\_i**: Specifies active SIMD lanes. Only lanes indicated in this mask will perform the operation.
14. **in\_valid\_i**: Input validity signal, indicating that the provided inputs are valid and ready for processing.
15. **flush\_i**: Signal to flush the pipeline, used to reset or clear ongoing operations.

**Output Signals**

1. **in\_ready\_o**: Output signal indicating that the module is ready to receive new input data.
2. **result\_o**: The final result of the operation, output at the specified width.
3. **status\_o**: Status of the floating-point operation (e.g., overflow, underflow), indicating any exceptions.
4. **extension\_bit\_o**: Extra bit used to extend the output or signal specific conditions in the result.
5. **tag\_o**: Propagates the tag from tag\_i to the output, useful for tracking data in pipelined systems.
6. **out\_valid\_o**: Signals that the output data (result and status) is valid and ready to be used.
7. **busy\_o**: Indicates that there is ongoing processing within the module, useful for handshaking.
8. **out\_ready\_i**: Signals from outside the module when the output has been received, allowing for the next result to be generated.

**Internal Logic and Functionality**

1. **Parameterization**: The module parameters allow it to be configured with different widths, formats, and capabilities, making it flexible for various floating-point operations and precision requirements.
2. **Parallel Slice Generation**:
   * The gen\_parallel\_slices loop generates parallel computation slices based on the format (NUM\_FORMATS).
   * Each format can have its dedicated slice, enabling the module to perform operations in parallel for different data types (e.g., single vs. double precision).
   * A specific slice is activated based on the destination format (dst\_fmt\_i), and it only processes valid operations.
3. **NaN Boxing and SIMD Lane Masking**:
   * is\_boxed\_i enables NaN-boxing checks to ensure that invalid data (like NaNs) is handled properly across different precision formats.
   * simd\_mask\_i allows the module to perform operations selectively across lanes, supporting vector-based calculations in hardware.
4. **Merged Slice Generation**:
   * If multi-format operations are enabled, a merged slice is created to handle multiple formats in one go.
   * This is controlled by parameters like FmtUnitTypes, which specify how each format should be handled (e.g., merged vs. separate).
5. **Round-Robin Arbiter**:
   * The arbiter decides which operation’s result to use based on the format selected.
   * It handles requests from each format slice and ensures that only one result is output at a time.
6. **Output Assignment**:
   * The output values (result\_o, status\_o, extension\_bit\_o, tag\_o) are assigned based on the selected arbiter output.
   * If multiple slices are producing results, the arbiter ensures they are serialized correctly to avoid conflicts.

**Overall Module Functionality**

The fpnew\_opgroup\_block module provides a flexible and parameterized way to perform floating-point operations across various formats, with support for SIMD (Single Instruction, Multiple Data) operations and multiple precision formats. The module receives operands and processes them through format-specific or merged slices, depending on the configuration. An internal arbiter then selects the appropriate result and forwards it as output, along with associated status and tag information, ensuring compatibility with larger pipelined systems. The module also includes NaN-boxing, rounding, and format control mechanisms to handle floating-point exceptions and adhere to IEEE standards.

## **RR\_ARB\_TREE:**

The rr\_arb\_tree module is a round-robin arbitration tree. This module is designed to select one of multiple inputs (up to 64 by default) in a fair, non-starving manner, ensuring each input has a chance to be serviced. Here’s an explanation of the input and output signals, as well as an overview of the module's overall functionality:

**Input Signals**

1. **clk\_i (Clock)**: The clock signal. The module is positive-edge triggered, so any state changes occur on the rising edge of clk\_i.
2. **rst\_ni (Reset)**: An active-low asynchronous reset. When rst\_ni is low, the module resets to a default state.
3. **flush\_i (Flush)**: Clears the arbiter state. It’s used to reset internal round-robin counters and locks (if enabled) without requiring a full module reset.
4. **rr\_i (External Round-Robin Priority)**: Allows an external module to control the round-robin priority instead of using the module’s internal counter. This is only used if ExtPrio is set to 1'b1.
5. **req\_i (Request)**: A vector with each bit representing an input request (e.g., for 64 inputs, this vector has 64 bits). Each bit indicates if the corresponding input has an active request for arbitration.
6. **data\_i (Input Data)**: A vector of NumIn data elements, with each element being of type DataType. It holds the payload data associated with each request, which will be passed to the output if its corresponding request is granted.
7. **gnt\_i (Grant Input)**: Indicates whether the downstream receiver is ready to accept data. When this is high, the arbiter can proceed with the grant process.

**Output Signals**

1. **gnt\_o (Grant Output)**: A vector with each bit indicating whether the corresponding input request has been granted. Only one bit is high at a time, reflecting the round-robin selection.
2. **req\_o (Request Output)**: This is high if there is an active request among the inputs. It indicates that the arbiter has a request to send downstream.
3. **data\_o (Output Data)**: The data from the granted input request, which will be forwarded to the output.
4. **idx\_o (Index Output)**: Provides the index of the input whose data is currently being sent out. This allows the downstream logic to identify the source of data\_o.

**Working and Purpose of the Module**

The primary function of the rr\_arb\_tree module is to fairly and efficiently arbitrate requests from multiple inputs based on a round-robin mechanism, with features to handle fair versus unfair arbitration.

**Key Functionalities:**

1. **Round-Robin Arbitration**: The round-robin approach ensures that each input has a chance to be selected in turn, preventing any input from being "starved" (i.e., never selected even if it continually requests). When an input is granted, the internal rr\_q counter is incremented (or reset) to point to the next input with the highest priority.
2. **Fair vs. Unfair Arbitration**:
   * **Fair Arbitration (FairArb = 1'b1)**: If set, the arbiter considers the current request pattern when deciding the next state. For instance, it will rotate to the next unserved request with a higher index to ensure that active inputs receive equal access over time.
   * **Unfair Arbitration (FairArb = 1'b0)**: Here, the arbiter simply advances the priority in a fixed rotation, regardless of which inputs are actively requesting, leading to potentially unequal throughput distribution if not all inputs are requesting.
3. **External vs. Internal Priority**:
   * **External Priority (ExtPrio = 1'b1)**: Allows another module to control the round-robin counter directly via rr\_i.
   * **Internal Priority**: When ExtPrio is 1'b0, the arbiter manages the round-robin state internally, incrementing after each granted request.
4. **AXI Compliant Handshaking**: The module can optionally comply with the AXI handshake protocol by setting AxiVldRdy. This ensures compatibility with systems that use AXI-style signals.
5. **Lock-in Mechanism**: If LockIn is set, the arbiter "locks" onto a request if the downstream module isn’t ready (i.e., gnt\_i is low). This prevents the arbiter from switching to another request until the locked request is fully served.

**Internal Operation**

* **Arbitration Tree**: The module implements a tree structure where requests propagate up the tree, and grants propagate down. This structure helps achieve efficient arbitration among many inputs by handling requests in smaller groups.
* **Masked Requests**: For fair arbitration, the module uses a "masked" request system that checks both higher and lower priority requests around the current priority. This ensures that unserved requests are rotated fairly based on the rr\_q counter.

The rr\_arb\_tree module, through its round-robin arbitration and configuration options, provides a highly customizable, efficient way to handle multiple input requests fairly while meeting timing and area constraints for various digital systems.

## **FPNEW\_OPGROUP\_FMT\_SLICE:**

The fpnew\_opgroup\_fmt\_slice module is a parameterized floating-point computation unit designed for operations in different formats (like single-precision, double-precision) and for specific groups of operations, such as addition/multiplication (ADDMUL) and non-comparative operations (NONCOMP). It is part of a larger floating-point unit (FPU) system that likely has vectorized (SIMD) floating-point capabilities. Here’s an explanation of its components, purpose of each input/output, and a breakdown of its functionality.

**Module Inputs and Outputs**

**Inputs**

1. **clk\_i**: Clock input to synchronize data processing.
2. **rst\_ni**: Active-low reset input to initialize or reset the module.
3. **operands\_i**: Array of floating-point operands for each lane, with each element being Width bits wide. The NUM\_OPERANDS parameter defines how many operands each operation can take (e.g., 2 for binary operations).
4. **is\_boxed\_i**: Array that flags whether each operand is “boxed” (represents a non-standard or exception value).
5. **rnd\_mode\_i**: Specifies the rounding mode, such as round-to-nearest or round-toward-zero.
6. **op\_i**: Operation code to determine the specific operation (e.g., addition, subtraction).
7. **op\_mod\_i**: Modifier for the operation (for example, to change behavior based on operation type).
8. **vectorial\_op\_i**: Indicates if the operation should be performed in vectorized mode (SIMD).
9. **tag\_i**: Tag input for attaching additional metadata to operations, which could help identify or track operations.
10. **simd\_mask\_i**: SIMD mask specifying which lanes are active in vector operations. If EnableVectors is false, this is ignored.
11. **in\_valid\_i**: Input validity signal indicating whether incoming data is valid and ready for processing.
12. **flush\_i**: Signal to clear pipeline data, typically to prevent processing of stale data.
13. **out\_ready\_i**: Indicates that downstream logic is ready to receive output data.
14. **reg\_ena\_i**: Register enable signal for each pipeline register, used for selectively enabling registers in multi-cycle operations.

**Outputs**

1. **in\_ready\_o**: Indicates that the module is ready to accept new input data.
2. **result\_o**: Result of the floating-point computation, with width equal to Width.
3. **status\_o**: Status flags providing additional information on the result, such as exceptions (overflow, underflow, etc.).
4. **extension\_bit\_o**: A bit output that signals extension-related information, possibly for extending precision.
5. **tag\_o**: Tag output to propagate metadata through the pipeline.
6. **out\_valid\_o**: Indicates that the output data is valid and can be consumed by downstream logic.
7. **busy\_o**: Indicates that the module is actively processing data.

**Overall Module Functionality**

This module implements a SIMD-capable floating-point operation pipeline where each lane can process part of the overall input vector. It uses various local parameters and conditions to configure lanes to process inputs either as a single large operation (scalar) or as multiple smaller operations (vectorized). Here’s how it works in general:

1. **Input Handling**: Each lane gets a portion of the operands\_i based on the width and lane index. in\_ready\_o is generated from lane\_in\_ready[0] to signal the module is ready for input when the first lane is ready.
2. **Lane Instantiation**:
   * Depending on OpGroup, different lanes instantiate specific operations. For example, if OpGroup is ADDMUL, it instantiates an adder or multiplier for that lane. OpGroup helps modularize functionality, allowing the same design to support multiple floating-point operations by conditionally instantiating the required submodules.
   * In vectorized mode, multiple lanes execute in parallel on different data elements. Each lane outputs its result (op\_result), status (op\_status), and other control signals.
3. **Result Aggregation**:
   * The results from each lane (local\_result) are combined to form slice\_result, a wider vector of floating-point results.
   * If the operation is classification-based, a mask (slice\_vec\_class\_result) is generated to indicate the type (e.g., QNaN, SNAN, POSINF).
   * The final result (result\_o) is selected based on whether the result represents a class mask or a regular floating-point result.
4. **Status and Output Handling**:
   * status\_o is the aggregate of all lane statuses, providing a consolidated view of operation flags.
   * busy\_o indicates whether any lanes are actively processing, while out\_valid\_o signals that the output is ready.

**Overall working:**

The fpnew\_opgroup\_fmt\_slice module provides configurable, vectorized floating-point operations tailored to different functional groups. Each lane processes data independently, which allows parallel execution for SIMD operations. The overall design is intended for use in high-performance FPU systems where various floating-point operations (addition, multiplication, classification) can be selectively instantiated and processed based on a format and operational requirements.