Project 1: Multiply-Accumulate (MAC) Unit Design

GitHub Repository: https://github.com/karans5/mac

Malavika Sanjeevan EE24M098

Karan S NS24Z028

A **Multiply-Accumulate (MAC)** unit is a fundamental digital component used in arithmetic operations, particularly in signal processing, machine learning, and various computational algorithms. It performs a sequence of multiplication and accumulation tasks, where two numbers are multiplied, and the resulting product is added to an accumulator. This operation is essential in applications like digital filters, neural networks, and matrix multiplications, making it a crucial part of digital signal processors (DSPs), microcontrollers, and hardware accelerators. The MAC unit's efficiency is critical for high-speed, real-time processing, as it enables simultaneous execution of multiple arithmetic operations, thereby improving overall system performance.

1. MAC UNPIPELINED DESIGN

a. BSV DESIGN

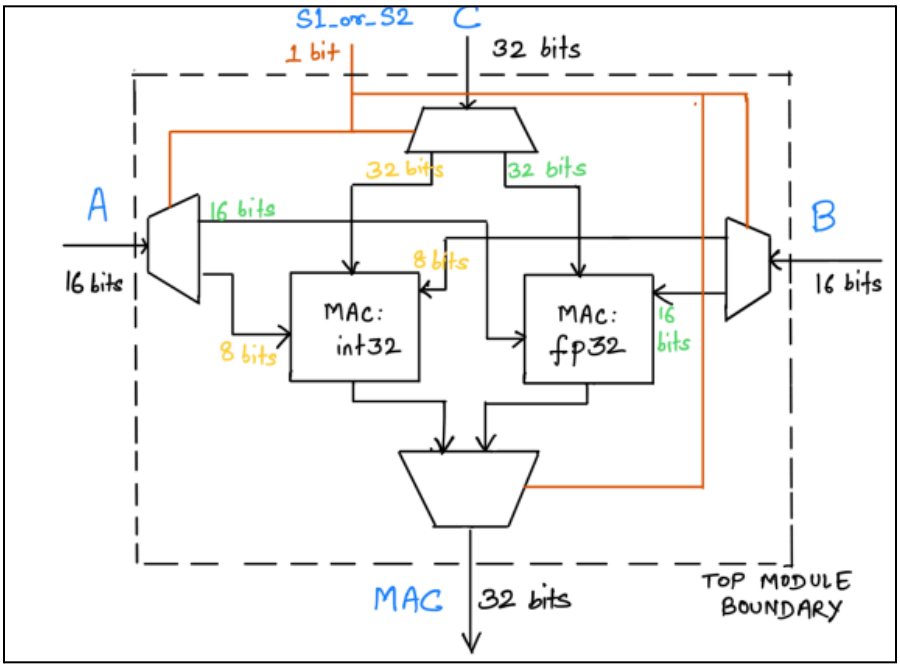


Fig1: Top-level architecture of MAC module

The top-level `Mac` module is designed to operate in either **integer** or **floating-point mode**, providing flexible functionality using two distinct sub-modules: `MAC_int` for integer operations and `MAC_fp` for floating-point operations. The module determines the operation mode based on a **single-bit signal** (`select`), which switches between the two modes. The module's interface, `MAC_ifc`, offers a configurable set of methods for setting the operands (`A`, `B`, and `C`), selecting the mode, and retrieving the MAC result. This approach enables seamless switching between integer and floating-point arithmetic, allowing it to handle various operations within a single design.

The module's interface, `MAC_ifc`, provides methods to set operands (`A`, `B`, and `C`), select the mode, and retrieve the 32-bit MAC result. Internally, the module uses 16-bit registers (`regA` and `regB`) to store primary operands, a 32-bit register (`regC`) for accumulation, and a 1-bit register (`select`) to toggle modes. Rules dictate data flow: `rl_mac_int` handles integer operations by slicing `regA` and `regB` into 8-bit segments, packing them with `regC` into a structure (`MACinputsint`), and passing them to `MAC_int`. Meanwhile, `rl_mac_fp` manages floating-point operations by packing `regA`, `regB`, and `regC` into a structure (`MACinputsfp`) and sending them to `MAC_fp`. Input values are set using the methods `get_A`, `get_B`, and `get_C`, while mode selection is controlled by `set_S1_or_S2`. The final MAC result is retrieved through `get_MAC`, which dynamically selects the output from `MAC_int` or `MAC_fp` based on the current mode. Integrating sub-modules, rules, and registers ensures efficient handling of integer and floating-point operations, making the `Mac` module versatile and adaptable for diverse digital signal processing applications.

Interface (`MAC_ifc`)

The `MAC_ifc` interface provides the following methods:

- **`get_A`**: Sets the operand `A`.
- **`get_B`**: Sets the operand `B`.
- **`get_C`**: Sets the operand `C` for accumulation.
- **`set_S1_or_S2`**: Sets the mode of operation (0 for integer, 1 for floating-point).
- **`get_MAC`**: Retrieves the 32-bit MAC result based on the current mode.

Module Body (`mkMac`)

The `mkMac` module implements the `MAC_ifc` interface, instantiating both `MAC_int` and `MAC_fp` sub-modules and providing the logic to switch between them based on the `select` signal.

1. **Sub-module Instantiation**:
    - The integer and floating-point MAC sub-modules (`mac_int` and `mac_fp`) are instantiated using `mkMAC_int` and `mkMAC_fp,` respectively.
2. **Register Declarations**:
    - Registers `regA`, `regB`, and `regC` store inputs, and `select` determines the active MAC mode.
3. **Rules**:
    - **`rl_mac_int`**: Executes when `select` is `0`.
        - Slices `regA` and `regB` into 8-bit portions for integer multiplication.
        - Creates a `MACinputsint` structure containing these sliced values along with `regC`.
        - Passes this structure to `mac_int.get_IntInputs`, initiating the integer MAC computation.
    - **`rl_mac_fp`**: Executes when `select` is `1`.
        - Packs `regA`, `regB`, and `regC` into a `MACinputsfp` structure.
        - Passes this structure to `mac_fp.get_FpInputs`, initiating the floating-point MAC computation.

Module operation flow:

1. **Setting Inputs**:
    - Input values for `A`, `B`, and `C` are provided via the `get_A`, `get_B`, and `get_C` methods. These values are stored in `regA`, `regB`, and `regC` for later use.
2. **Mode Selection**:
    - The mode of operation is set by calling `set_S1_or_S2`.
    - When `select` is `0`, integer MAC mode is chosen.
    - When `select` is `1`, floating-point MAC mode is chosen.
3. **MAC Calculation**:
    - Depending on the `select` signal:
        - If in integer mode (`select == 0`), the sliced values of `regA` and `regB`, along with `regC`, are passed to `macint`.
        - If in floating-point mode (`select == 1`), the total values of `regA`, `regB`, and `regC` are passed to `macfp`.
4. **Retrieving the Result**:
    - The `get_MAC` method provides the final 32-bit MAC result. It calls either `macint.get_MACint` (for integer) or `macfp.get_MACfp` (for floating-point), based on the mode

MAC SUB-MODULES

A. MAC_INT

The module uses a signed 8-bit multiplier and a 32-bit ripple-carry adder to perform the MAC operation, where it multiplies two integer inputs, adds a third, and stores the result. This design includes several key features and components to manage the control flow and handle signed arithmetic. The main components of this sub-module are listed below:

1. **`MACinputsint` Struct**:
    - A struct `MACinputsint` is defined to package the three inputs needed for the MAC operation. It includes:
        - `a`: Signed 8-bit integer (first operand of the multiplication).
        - `b`: Signed 8-bit integer (second operand of the multiplication).
        - `c`: Signed 32-bit integer (to be added to the product).
    - The struct derives `Bits` and `Eq`, allowing easy manipulation and comparison of its instances.
2. **Interface `MAC_int_ifc`**:
    - Defines the interface for the MAC module with two main methods:
        - `get_IntInputs(MACinputsint inputs)`: Receives the inputs for the MAC operation.
        - `get_MACint()`: Returns the final MAC result as a signed 32-bit integer.
3. **Module `mkMAC_int`**:
    - This is the main module that implements the `MAC_int_ifc` interface. It synthesizes the MAC operation's logic by integrating the adder and multiplier sub-modules.

The module uses several internal registers:

- **`regA`** and **`regB`**: Store the 8-bit signed inputs for multiplication.
- **`regC`**: Stores the 32-bit signed input for accumulation.
- **`macOut`**: Holds the final 32-bit result of the MAC operation.
- **`rg_sent_inputs`**: A flag indicating that the inputs have been sent to the multiplier.
- **`rg_add_complete`**: A flag indicating that the addition is complete.

The `mkMAC_int` module has three main rules that define the MAC operation flow:

1. **`rl_startMAC`**:
   - Initiates the MAC operation by sending `regA` and `regB` as inputs to the multiplier.
   - `regA` and `regB` are packaged into a struct `GetMulInp` and passed to `mul.get_Inputs`.
   - Sets `rg_sent_inputs` to `True` to indicate that the multiplication process has started.
2. **`rl_intermediateMAC`**:
   - Triggered after `rg_sent_inputs` is set to `True`.
   - Retrieves the multiplication result (`product`) from `mul.get_Mul`, sign-extends it to 32 bits, and sends it along with `regC` to the adder module.
   - Starts the adder operation by calling `rca.start` with `product` and `regC` as inputs.
   - Sets `rg_sent_inputs` to `False` and `rg_add_complete` to `True` to indicate the addition process has started.
3. **`rl_getMAC`**:
   - Triggered after `rg_add_complete` is set to `True`.
   - Retrieves the final addition result (`sum`) from `rca.get_add()` and stores it in `macOut`.
   - Resets `rg_add_complete` to `False`, indicating the MAC operation is complete.

The module has two primary methods:

1. **`get_IntInputs(MACinputsint inputs)`**:
   - This method takes the `MACinputsint` struct as input and assigns its fields to `regA`, `regB`, and `regC`.
   - This sets up the values needed for the MAC operation.
2. **`get_MACint()`**:
   - Returns the value stored in `macOut`, which contains the final MAC result after completing multiplication and addition.

Module Operation Flow

1. **Input Loading**:
   - Inputs are set via `get_IntInputs`, which stores values in `regA`, `regB`, and `regC`.
2. **Multiplication**:
   - `rl_startMAC` sends `regA` and `regB` to the multiplier, initiating the multiplication process.
   - `rg_sent_inputs` is set to `True` to signal that the multiplier inputs have been provided.
3. **Addition**:
   - `rl_intermediateMAC` is triggered when the multiplier result is ready.
   - It retrieves the signed 32-bit product and starts the addition with `regC` using the ripple-carry adder.
   - `rg_add_complete` is set to `True` to indicate that the addition has started.
4. **Result Retrieval**:
   - `rl_getMAC` retrieves the addition result and stores it in `macOut`.
   - The MAC result can now be retrieved via `get_MACint()`.

B. Adder_int : For addition of int32 inputs

The `mkRipplecarryadder` module, part of the `adder_int` package in Bluespec SystemVerilog (BSV), implements a 32-bit signed Ripple Carry Adder (RCA) using a bit-by-bit ripple carry mechanism to add two 32-bit signed integers. It includes an overflow flag to detect when overflow occurs during addition. The core addition logic is encapsulated in the `ripple_carry_addition` function, ensuring clear functionality. The module provides a simple interface to set the inputs and retrieve the result, making it easy to integrate and reuse across various designs. The key components are as follows:

1. **`Adderresult` Struct**:
   - The `Adderresult` struct encapsulates the result of the addition operation. It has:
     - `sum`: a 32-bit signed integer storing the addition result.
     - `overflow`: a 1-bit flag indicating if overflow occurred (set to 1 if true).
2. **Interface `RCA_ifc`**:
   - Defines the interface for the ripple carry adder module:
     - `start(Int#(32) a, Int#(32) b)`: loads the inputs for addition.
     - `get_add()`: retrieves the result of the addition (an `Adderresult` struct).
3. **Registers**:
   - `rg_inp1` and `rg_inp2`: Registers to store the 32-bit signed integer inputs.
   - `rg_inp_valid`: A delayed register (DReg) indicating the validity of input data.
   - `rg_out`: Stores the result of the addition operation in the form of an `Adderresult` struct.
   - `rg_out_valid`: A delayed register to flag the validity of the output result.

4. **Ripple Carry Addition Function (`ripple_carry_addition`)**:
    - This function implements the bitwise addition logic. It computes the sum and carry for each bit in a sequential manner (one bit at a time), with each bit's carry flowing to the next bit position.
    - **Carry Propagation**: Uses a 33-bit `carry` register to handle carries between bits and check for overflow in the final carry-out bit.
    - The function returns an `Adderresult` struct, which includes the final `sum` and the overflow flag (set based on `carry[32]`, the final carry-out bit).
5. **Rule (`rl_rca`)**:
    - `rl_rca` triggers the addition by calling `ripple_carry_addition` on `rg_inp1` and `rg_inp2`, with a carry-in (`cin`) of 0.
    - The result is stored in `rg_out`, and the output validity flag (`rg_out_valid`) is set to `True`, indicating the result is ready.
6. **Methods**:
    - `start(Int#(32) a, Int#(32) b)`: Allows external modules to set the inputs `a` and `b` for the addition.
    - `get_add()`: Provides access to the `Adderresult` struct stored in `rg_out`, which contains the computed sum and overflow status.

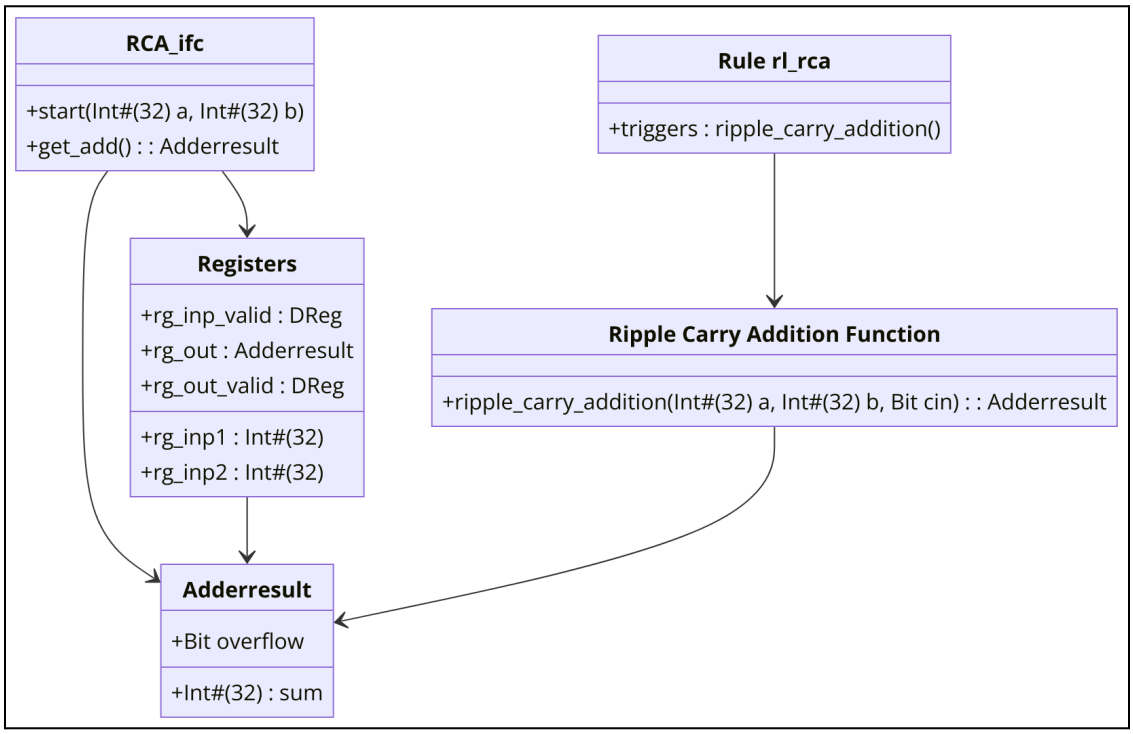


Fig2: Adderresult Struct and RCA Interface

C. Multiplier_int : For multiplication of int8 inputs

The `mkMult` module is designed as an iterative, bitwise multiplier using a sequential shift-and-add approach. The multiplication is performed over multiple clock cycles, controlled by the `rl_compute` rule. The key components of the microarchitecture are as follows:

1. Input Handling: The `get_Inputs` method loads the 8-bit inputs into the `d` (multiplicand) and `r` (multiplier) registers, resetting `product` to `0` and `done` to `False`.
2. Shift-and-Add Loop: The `rl_compute` rule is the core of the multiplication process, firing if `r` is non-zero and multiplication is incomplete. This rule checks the least significant bit of `r`. If it is `1`, `d` is added to `product`. The rule then shifts `d` left by one bit (doubling it) and shifts `r` right by one bit (halving it), effectively handling one bit of the multiplier in each cycle.
3. Completion Check: When `r` reaches `1`, the `done` flag is set to `True`, signaling the end of the multiplication process.

Key Components

1. **Struct `GetMulInp`:** Holds the signed 8-bit inputs, `a` (multiplicand) and `b` (multiplier).
2. **Interface `Mult_ifc`:** Defines methods for setting inputs and retrieving the result, **ensuring** standardized interaction with the module.
3. **Module `mkMult`:** Implements the `Mult_ifc` interface and contains the multiplier logic.
4. **Registers:**
    - `product`: A 16-bit register to store the final result.
    - `d`: A 16-bit register for the extended multiplicand.
    - `r`: An 8-bit register for the multiplier.
    - `done`: A Boolean flag indicating the completion of multiplication.

The design uses a shift-and-add algorithm, processing one bit of the multiplier at a time:

- Bitwise Addition: Adds the multiplicand to `product` if the current bit of `r` is `1`.
- Shifting:
    - Left-shift the multiplicand (`d`), effectively doubling its value.
    - Right-shift the multiplier (`r`), moving to the next bit.

This iterative process continues until all bits of `r` are processed, with `done` set to `True` upon completion. The data flow ensures efficient accumulation of partial products and proper alignment of bits.

D. MAC_FP module

The `mkMAC_fp` module in the `mac_fp` package implements a floating-point Multiply-Accumulate operation using a sequential architecture. It integrates separate floating-point multiplier and adder modules, processing the inputs in stages. The data flow is register-based and state-controlled, following the IEEE 754 floating-point arithmetic standards. The key components are :

1. **Interface (`MAC_fp_ifc`):**
    - The interface defines two methods:
        - `get_FpInputs(MACinputsfp inputs)`: Receives three inputs—two 16-bit floating-point numbers (`a` and `b`) and one 32-bit floating-point number (`c`).
        - `get_MACfp()`: Returns the 32-bit MAC output after computation.
2. **Inputs:**
    - The struct `MACinputsfp` holds three inputs:
        - `A` (16-bit): First input for multiplication.
        - `b` (16-bit): Second input for multiplication.
        - `c` (32-bit): Input to be added to the product.
3. **Control Signals:**
    - Two Boolean registers:
        - `rg_sent_inputs`: Indicates if inputs are sent to the multiplier.
        - `rg_add_complete`: Indicates when the addition is complete.

4. **Submodules:**

    - `mkFP_multiplier`: Performs floating-point multiplication.
    - `mk_Adder_fp`: Performs floating-point addition.

The two modules are connected sequentially to perform a multiply-accumulate operation.

5. **Register:**

    - `regA` (16-bit): Holds the multiplicand.
    - `regB` (16-bit): Holds the multiplier.
    - `regC` (32-bit): Holds the value to be added.
    - `macFpOut` (32-bit): Stores the final MAC result.

Operation Flow:

1. Input Handling:
    - The method `get_FpInputs()` sets the registers (`regA`, `regB`, `regC`) with the respective inputs.
    - It initiates the MAC operation by loading these inputs into the multiplier.
2. Sequential Operations:
    - The operation is divided into three main rules:
        1. `rl_startMACfp`: Initiates the multiplication by sending inputs (`regA` and `regB`) to the multiplier interface.
        2. `rl_intermediateMACfp`:

        - Fetches the 32-bit product from the multiplier and sends it, along with `regC`, to the adder interface.
        - Sets `rg_sent_inputs` to `False` and `rg_add_complete` to `True`, indicating the transition to the addition stage.
    3. `rl_getMACfp`:
        - Retrieves the 32-bit sum from the adder and stores it in `macFpOut`.
        - Resets `rg_add_complete` to `False`, marking the end of the MAC operation.
3. MAC Calculation:
    - The MAC operation involves two main steps:
        1. Multiplication:
            - Multiplies inputs `regA` and `regB`, producing a 32-bit floating-point product.
        2. Addition:
            - Adds the product to `regC` using the floating-point adder.
    - The results are stored in `macFpOut`, which can be retrieved using the `get_MACfp()` method.

D. `multiplier_fp` module for floating point multiplication

The `mkFP_multiplier` module in the `multiplier_fp` package implements a sequential floating-point multiplier following IEEE 754 standards. It processes 16-bit bfloat16 inputs and produces a 32-bit FP32 output using a register-based, rule-driven architecture. The data flow is designed to handle inputs sequentially, performing core multiplication, normalization, and rounding to generate an accurate FP32 result. The key components are as follows:

1. Registers for Input and Output:
    - `rg_A` (16-bit): Stores the first operand of the multiplication.
    - `rg_B` (16-bit): Stores the second operand of the multiplication.
    - `rg_Product` (32-bit): Stores the resulting 32-bit floating-point product.
2. Control Registers:
    - `rg_inp_valid`: Indicates whether valid inputs are available for computation.
    - `rg_Product_valid`: Indicates whether the computed product is valid and ready for retrieval.

The data flow in the module involves the following steps:

1. Input Handling:
    - The `start` method is invoked with two 16-bit inputs (`a` and `b`).
    - The inputs are stored in registers (`rg_A` and `rg_B`), and the `rg_inp_valid` register is set to `True` to indicate that valid inputs are available for processing.
2. Floating-Point Conversion:
    - Inputs are interpreted in bfloat16 format, which includes a 1-bit sign, an 8-bit exponent, and a 7-bit fraction.
    - The 7-bit fraction is extended to 23 bits to comply with FP32 format by padding with 16 zeros.

3. Computation:
    - The core multiplication operation occurs in the `fp_multiply` function, which performs the following steps:
        - Sign Calculation: The result's sign is determined by XORing the signs of the inputs.
        - Exponent Addition: The exponents of the inputs are added, adjusting for bias as per IEEE 754.
        - Fraction Multiplication:
            - The fractions of the two inputs are multiplied using a bitwise shift-and-add approach.
            - The fractional multiplication is extended to 16 bits to accommodate intermediate results.
        - Normalization: After multiplication, the result is checked for leading 1s and normalized accordingly by right-shifting the fraction and incrementing the exponent.
        - Rounding: The result is rounded to the nearest value, taking into account guard, round, and sticky bits to ensure compliance with IEEE 754 rounding rules.
4. Output Handling:
    - After computation, the 32-bit product is stored in `rg_Product`, and the `rg_Product_valid` register is set to `True` to indicate that the result is ready.
    - The result can be retrieved using the `get_Product()` method.
5. Rule-Based Execution:
    - The `rl_compute_product` rule is triggered when `rg_inp_valid` is `True`, indicating that inputs are available for multiplication.
    - It processes the multiplication, sets the result in `rg_Product`, and updates the control registers to indicate the operation's completion.

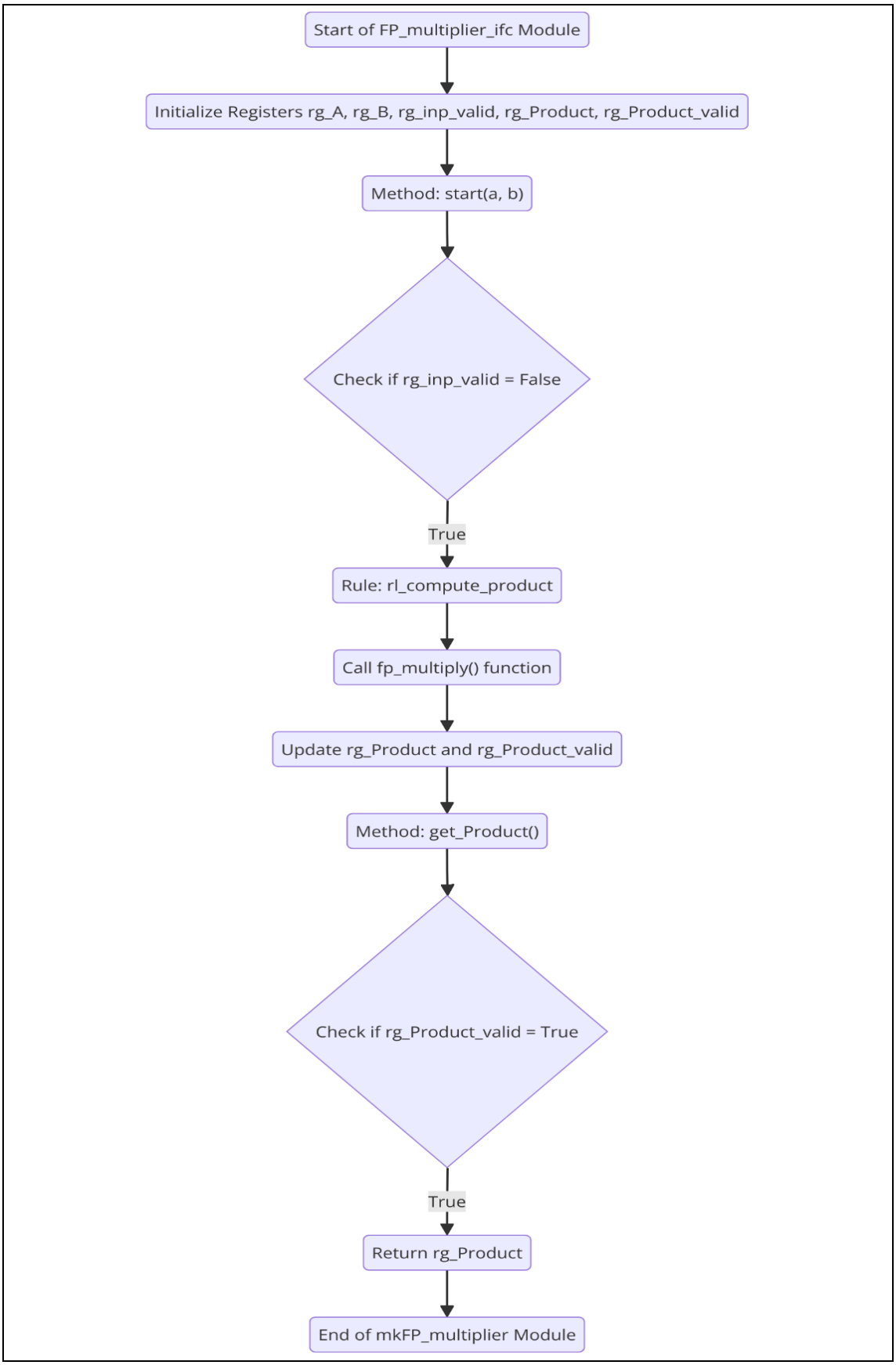


Fig3: General data flow of multplier_fp module

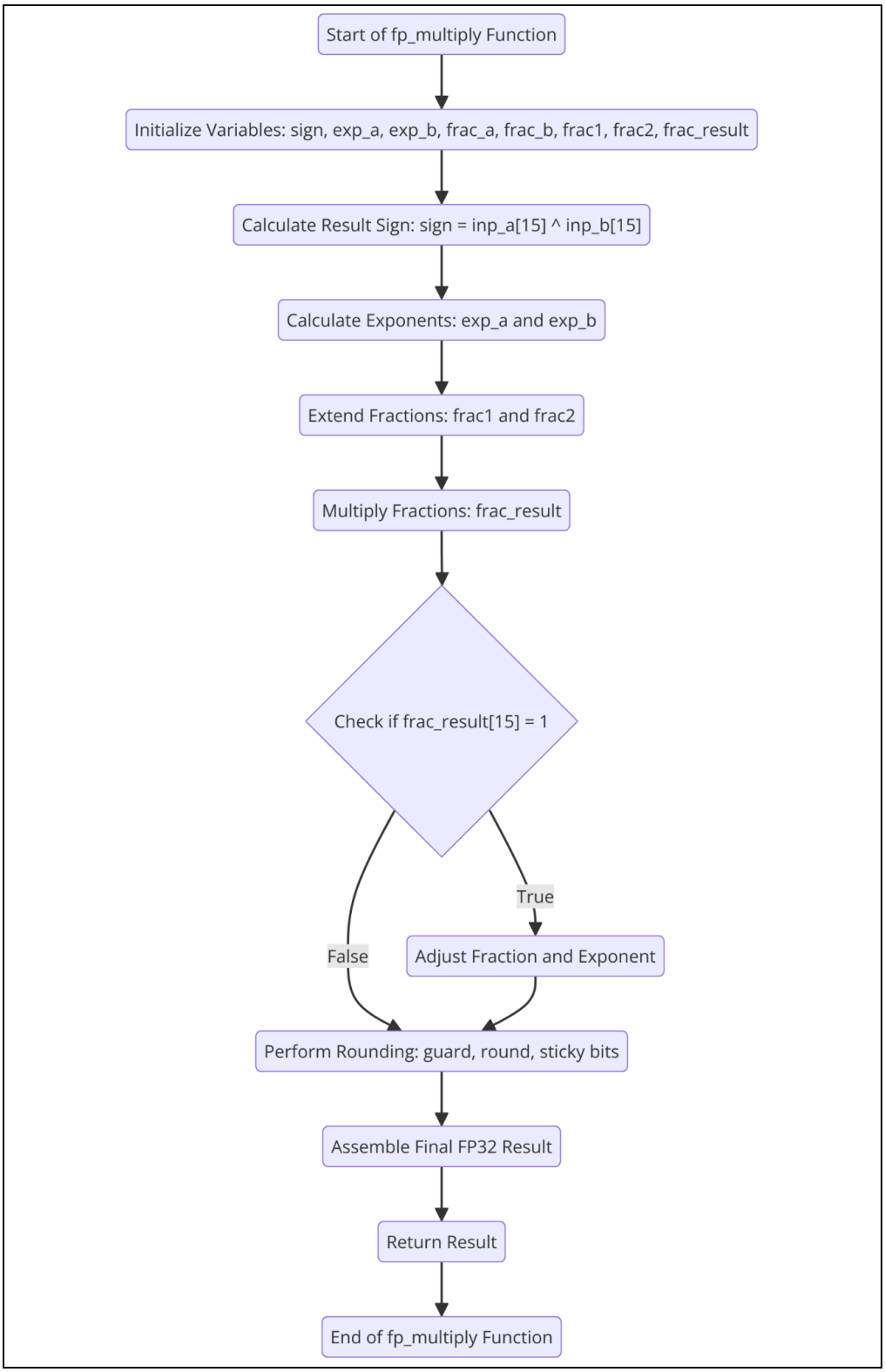


Fig4: Flowchart of fp_multiply function

D. `adder_fp` module for floating-point addition

The `mk_Adder_fp` module in the `adder_fp` package implements IEEE 754-compliant floating-point addition for two 32-bit FP32 inputs. It aligns the exponents, performs addition or subtraction of the significands, normalizes and rounds the result, and handles overflow or underflow. The design uses a register-based, rule-driven architecture to ensure accurate floating-point operations.

1. **Registers for Input and Output:**
    - `rg_fp_inp1` (32-bit): Stores the first operand of the addition.
    - `rg_fp_inp2` (32-bit): Stores the second operand of the addition.
    - `rg_fp_out` (32-bit): Stores the result of the floating-point addition.
2. **Control Registers:**
    - `rg_fp_inp_valid`: Indicates whether valid inputs are available for computation.
    - `rg_fp_out_valid`: Indicates whether the computed sum is valid and ready for retrieval**.**

3. Addition Rule (`rl_fp_add`):

- Triggered when `rg_fp_inp_valid` is True, indicating valid inputs are ready.
- Invokes the `fp_addition` function to process floating-point addition on `rg_fp_inp1` and `rg_fp_inp2`.
- Stores the resulting 32-bit sum in `rg_fp_out`.
- Sets `rg_fp_out_valid` to True, signaling the output is ready for retrieval via `get_add()`.
- Updates `rg_fp_inp_valid` to False, marking the end of the current cycle and preparing for the next operation.
- Ensures sequential execution, performing addition only with valid inputs to prevent unintended calculations.
- The result can be accessed using `get_add()`, which checks `rg_fp_out_valid` to ensure the output is correct and validated.

4. Data Flow:

1. Input Handling:
    - The `start` method is invoked with two 32-bit inputs (`a` and `b`).
    - The inputs are stored in the registers (`rg_fp_inp1` and `rg_fp_inp2`), and the `rg_fp_inp_valid` register is set to `True`, indicating that valid inputs are available for computation.

2. Floating-Point Addition:
    - The core addition operation occurs in the `fp_addition` function, which follows these steps:

Steps Involved in Floating-Point Addition

- Unpack Inputs:
    - Extract the sign, exponent, and mantissa from the two 32-bit FP32 inputs:
        - Sign: Bit 31.
        - Exponent: Bits 30 to 23.
        - Mantissa: Bits 22 to 0, with an implicit leading 1 added to represent normalized values.
- Align Exponents:
    - Calculate the exponent difference between the two inputs:
        - `exp_Diff = |exp_A - exp_B|`
    - Shift the mantissa of the smaller exponent to the right by the exponent difference, ensuring that the exponents are aligned for addition or subtraction.
- Add or Subtract Mantissas:
    - If the signs of the two inputs are the same, add the aligned mantissas.
    - If the signs are different, subtract the smaller mantissa from the larger one.
    - The result is stored in a 25-bit variable (`mantissaSum`) to accommodate potential overflow.
- Normalize the Result:
    - If there is an overflow in the mantissa (i.e., the most significant bit is set), right-shift the mantissa by one bit and increment the exponent.
    - If normalization is needed, left-shift the mantissa and decrement the exponent until the leading bit becomes 1.
- Round the Result:
    - Use guard, round, and sticky bits for rounding:
        - Guard bit: The bit after the mantissa bits.
        - Round bit: The bit after the guard bit.
        - Sticky bit: The OR of all remaining lower bits.
    - Apply round-to-nearest-even rounding based on these bits.
- Check for Overflow or Underflow:
    - If the exponent exceeds the maximum (255), set the result to infinity.
    - If the exponent becomes zero or negative, set the result to zero or a denormalized value.
- Assemble the Final Result:
    - Combine the sign, exponent, and mantissa to form the final 32-bit FP32 result.
    - Store the result in the `rg_fp_out` register, and set the `rg_fp_out_valid` flag to `True`.
- Output Handling:
    - The `get_add()` method retrieves the 32-bit addition result from `rg_fp_out` when `rg_fp_out_valid` is `True`.

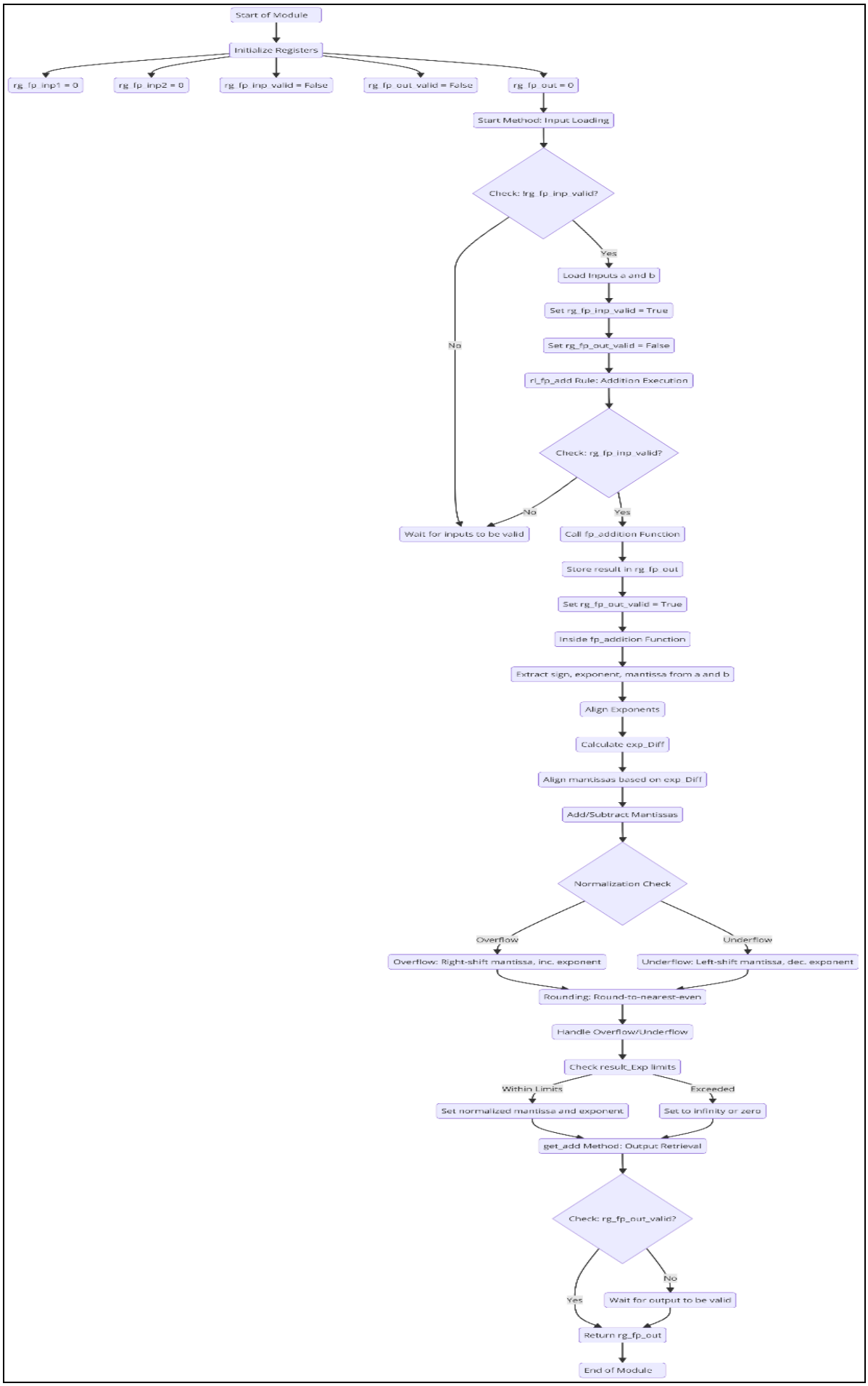


Fig5:Flowchart of adder_fp module

b.Verification methodology

The `model_mac` function is a Python-based reference model designed to verify the behavior of a hardware Multiply-Accumulate (MAC) module implemented in BSV (Bluespec SystemVerilog). It accepts three operands (`a`, `b`, `c`) and a `select` signal, simulating both integer and floating-point MAC operations based on the value of `select`. When `select` is `0`, it performs integer multiplication and addition, while a value of `1` triggers floating-point arithmetic. The function classifies the inputs as 'integer', 'fractional', 'negative', or 'other' for coverage analysis, ensuring comprehensive testing across different scenarios. The computed result is returned and logged, facilitating direct comparison with the BSV MAC module's output. This model plays a crucial role in validating the functionality of the hardware implementation and identifying discrepancies during testing, making it an integral part of the verification process.

The Python test model test_model.py leverages Cocotb to simulate, control, and verify a hardware MAC module. It performs the following key tasks:

1. Initializes the clock and resets the MAC module.
2. Loads inputs from files for integer and floating-point testing.
3. Tests integer MAC operations by setting the select signal to 0, processing the inputs, and comparing the results with the reference model.
4. Tests floating-point MAC operations similarly, with select set to 1.
5. Uses assertions to validate outputs, ensuring that the hardware implementation matches the expected behavior.
6. Exports coverage data for further analysis.

Coverage points are critical in hardware verification, as they help ensure that all meaningful input scenarios are tested, thus increasing confidence in the correctness and robustness of the design.

1. Cover Points for Inputs (`a`, `b`, `c`)

`mac.a` and `mac.b`

- Bins: `[0, 1, 32767, -32768, 16384, -16384]`
    - 0: Represents the simplest case where `a` or `b` is zero, ensuring that the MAC output is solely dependent on `c`.
    - 1: Represents a minimal non-zero positive value, useful for checking boundary conditions close to zero.
    - 32767: The largest positive value for a signed 16-bit integer. It tests the upper limit of positive values for `a` and `b`, which can cause maximum possible product values in integer MAC.
    - -32768: The smallest negative value for a signed 16-bit integer, testing the lower bound for `a` and `b`.
    - 16384 and -16384: Represent mid-range values that are often used to ensure the design works correctly for typical values in the middle of the range.

`mac.c`

- Bins: `[0, 1, 2147483647, -2147483648, 1073741824, -1073741824]`
    - 0: Ensures that the MAC output is purely the result of the product of `a` and `b`, without any contribution from `c`.
    - 1: Tests the smallest non-zero addition to the product of `a` and `b`, covering boundary conditions.
    - 2147483647: The largest positive value for a signed 32-bit integer, testing the upper limit of `c`.
    - -2147483648: The smallest negative value for a signed 32-bit integer, testing the lower bound for `c`.
    - 1073741824 and -1073741824: Mid-range values for a 32-bit signed integer, ensuring correct behavior for typical values that fall in between the upper and lower limits.

2. Cover Points for Select Signal (`select`)

`mac.select`

- Bins: `[0, 1]`
    - 0: Represents integer MAC mode, ensuring that the module performs correctly when operating in integer mode.
    - 1: Represents floating-point MAC mode, ensuring that the module performs correctly when operating in floating-point mode.

This coverage ensures that the MAC module is thoroughly tested for both integer and floating-point operations.

3. Cover Points for Output (`output`)

`mac.output`

- Bins: `[0, 1, 2147483647, -2147483648]`
    - 0: Ensures that the MAC module can correctly produce zero output when inputs are zero.
    - 1: Tests the smallest non-zero output.
    - 2147483647: The largest positive output, checking for correct handling of positive overflow conditions.
    - -2147483648: The smallest negative output, checking for correct handling of negative overflow conditions.

4. Cross-Coverage

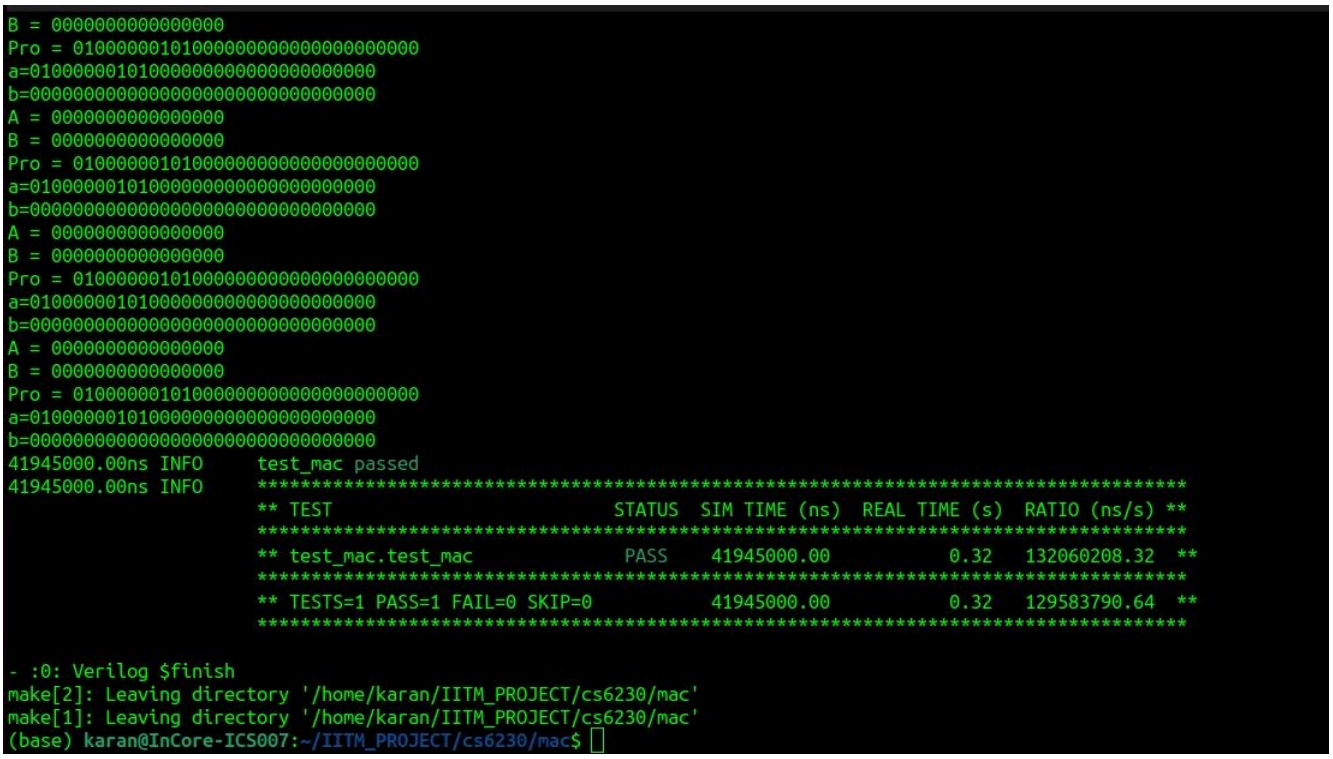
Cross-coverage checks the interaction between multiple input signals. It ensures that the MAC module behaves correctly not only for individual inputs but also for combinations of different inputs, which might reveal corner cases or unexpected behaviors.

CrossCoverage Definitions

- `cross_a_b`: Checks how different values of `a` interact with different values of `b`.
    - This is important because the MAC operation involves the multiplication of `a` and `b`, making it critical to verify all combinations to detect possible issues like overflow or incorrect handling of signed arithmetic.
- `cross_a_b_c`: Checks interactions between `a`, `b`, and `c`.
    - This covers the full MAC operation, ensuring that the module performs correctly across a range of input combinations for multiplication and addition.
- `cross_a_b_select`: Checks the interaction between `a`, `b`, and the `select` signal.
    - This ensures that the module correctly switches between integer and floating-point modes based on the `select` signal for all values of `a` and `b`.
- `cross_c_select`: Checks how different values of `c` interact with the `select` signal.
    - This verifies the module's behavior when switching modes while handling various values of `c` during accumulation.

The cover points are crucial for comprehensive testing of the MAC module, as they include critical test cases like boundary values, typical mid-range values, and zero, which help identify potential errors or edge cases. Testing all combinations of inputs through cross-coverage ensures the module's robustness by detecting subtle issues that may only arise with specific input combinations. Additionally, covering extreme values, such as maximum and minimum 16-bit and 32-bit integers, helps identify overflow or underflow scenarios that might otherwise be overlooked. Finally, by including different values of the select signal, the verification process ensures correct mode switching between integer and floating-point operations.

Test Results:



```
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0100110000100011110101110000 1011
[MAC Model] a=8522825728.0, b=6.65625, c=65011.71484375, select=1, result=56730123763.71484, type=fractional
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0100110000100011110101110000 1011
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0100011101111101111100111010 0111
[MAC Model] a=1125281431552.0, b=7147094016.0, c=115964120.0, select=1, result=8.042492185761329e+21, type=fractional
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0100011101111101111100111010 0111
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0100110011011101001011110001 1011
[MAC Model] a=0.8984375, b=65024.0, c=93323.265625, select=1, result=151743.265625, type=fractional
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0100110011011101001011110001 1011
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0100011110110110010001011010 0010
[MAC Model] a=104689827840.0, b=96.5, c=3236962500608.0, select=1, result=13339530887168.0, type=fractional
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0100011110110110010001011010 0010
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0101010000111100011010101000 0000
[MAC Model] a=8.1875, b=23040.0, c=38654708.0, select=1, result=38843348.0, type=fractional
A = 0000000000000000
B = 0000000000000000
Pro = 0100000010100000000000000000000000
a=0100000010100000000000000000000000
b=0101010000111100011010101000 0000
81935000.00ns INFO     test_mac passed
81935000.00ns INFO     ****************************************************************************************
                       ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                       ****************************************************************************************
                       ** test_mac.test_mac               PASS    81935000.00           0.66   124959792.71  **
                       ****************************************************************************************
                       ** TESTS=1 PASS=1 FAIL=0 SKIP=0           81935000.00           0.67   123197812.60  **
                       ****************************************************************************************

- :0: Verilog $finish
make[2]: Leaving directory '/home/karan/IITM_PROJECT/cs6230/mac'
make[1]: Leaving directory '/home/karan/IITM_PROJECT/cs6230/mac'
(base) karan@InCore-ICS007:~/IITM_PROJECT/cs6230/mac$
```