CS6230 - CAD for VLSI**: Project 1**

Multiply-Accumulate (MAC) Unit

Arun Prabakhar CS23D012
Karthikeyan R CS23S039

DESIGN

MAC (Multiply Accumulate) Module

Module implements integer and floating point arithmetic, with different modules handling the multiplication and addition operations.
The MAC module contains an interface, `Ifc_mac` which defines a function `mac.`

- `s1_or_s2` : (1 bit) Decides whether to use integer or floating point arithmetic.
- `a` and `b` : 16 bit inputs to be multiplied.
- `c`: 32 bits value to be accumulated with the multiplication result.
- Returns a 32-bit output.

Depending on the s1_or_s2 flag. The MAC operation can be switched between using either integer arithmetic or floating point arithmetic.

MAC module instantiates multiplier and adder modules for both integer and floating point operations.

- `mkMultiplier8Bit`: Multiplies two 8-bit integers.
- `mkAdder32Bit:` Adds two 32-bit integers.
- `mkFpMultiplier`: Multiplies two floating-point numbers.
- `mkFpAdder`: Adds two floating-point numbers.

PIPELINED MAC (Multiply Accumulate) Module

The design is divided into two stages:
STAGE 1: Perform multiplication (integer or floating point) based on a control signal (`s1_or_s2`).

- If `s1_or_s2==1,` multiplies the lower 8 bits of a and b using the 8-bit multiplier.
- If `s1_or_s2==0`, multiplies the 16 bit floating-point values `a` and `b` using the floating-point multiplier.
- The intermediate multiplication result is then queued into the FIFO for Stage 2 `(mac_intermediate_AB),` along with the value of `c` and the control signal `s1_or_s2.`

STAGE 2: Perform the accumulation using the results from Stage 1.

- The design dequeues the multiplication result, the control signal, and `c` from their respective FIFOs.

- If `s1_or_s2==1`, it adds the integer multiplication result to `c`.
- If `s1_or_s2==0`, it adds the floating-point multiplication result to `c`.
- The result of this addition is stored in a wire which is returned when `get_result` is called.

Packages:

1.Integer multiplier :

The interface `Ifc_multiplier` defines a method `mul` that performs multiplication on two inputs x and y, each of width n, and returns the product with a bit-width of 2n.
We define two modules in this package.
Signed Multiplier (`mkMultiplierSigned`) module:

This module performs signed multiplication for inputs `a` and `b` of width `n`.

Unsigned Multiplier (mkMultiplierUnsigned) module:

It is similar to the signed multiplier, but for unsigned multiplication.

- Both the multipliers use an adder module (`Ifc_adder`) to add partial products during the multiplication process.
- This allows the design to modularly reuse the adder for both signed and unsigned multipliers.

2.Integer adder :

The interface `Ifc_adder` defines a method add that takes two n-bit inputs x and y, and a control bit `add_or_sub`. This control bit decides whether to perform addition (`add_or_sub == 0`) or subtraction (`add_or_sub == 1`).
The result is of length n+1, meaning the result contains sum and carry out.

Adder Module (`mkAdder`):

- This module implements an `n`-bit ripple-carry adder/subtractor, using the full adder for each bit position.
- If `add_or_sub` is set to `1`, the second operand `(b)` is bitwise negated `(~b)`, enabling two's complement subtraction when combined with the `add_or_sub` bit as the initial carry-in.

Ripple-Carry Logic: For each bit position:

- The first bit position uses the `add_or_sub` value as the initial carry-in.
- Subsequent bits use the carry from the previous bit position.

The final carry-out is stored in `carry_out`, and the result is returned with the format.

3.Floating point adder :

The package includes a 32-bit floating-point adder/subtractor module (`mkFpAdder`) following the IEEE-754 format. It includes helper modules for addition and subtraction of mantissas and exponents.

Floating-Point Adder Interface (`Ifc_fpadder`):

The interface has one method, `fp_add`, which takes two 32-bit floating-point numbers as input and returns a 32-bit floating-point result.

Floating-Point Adder Module (`mkFpAdder`):

- This module performs floating-point addition and subtraction (`fp_add method`) :
  - The highest magnitude operand is assigned to the operand `op1` to simplify alignment.
  - The result's sign is based on the sign of `op1`, while `add_or_sub` controls whether addition or subtraction is performed based on the sign of both operands.
  - The difference between `exponent1` and `exponent2` is calculated using an 8-bit adder. The smaller operand's mantissa is then right-shifted by this difference to align the exponents.
  - Mantissa Addition and Subtraction:
    - If the operands have the same sign, mantissas are added.
    - If the signs differ, subtraction is performed.
    - After either operation, normalization adjusts the result mantissa and exponent if necessary.

4.Floating point multiplier :

The package includes a 16-bit floating-point multiplier (`mkFpMultiplier`) that follows the IEEE-754 floating-point standard. It performs floating-point multiplication using two 16-bit floating-point inputs, producing a 32-bit result.

Floating-Point multiplier Interface (`Ifc_fpmultiplier`):

The interface defines a single method `fp_mul` which multiplies two 16-bit floating-point numbers and returns a 32-bit floating-point result.

Floating-Point Multiplier Module (`mkFpMultiplier`):

- The sign of the result is determined by XORing the signs of the inputs (`a` and `b`).
- Both the input's exponent is extracted, and then they are summed using multiple 8-bit adders. The result is then adjusted based on IEEE-754 biasing and normalization.

- The implicit bit is added based on whether the number is normalized, and the mantissas are multiplied using an 8-bit multiplier.
- The intermediate mantissa result is checked for normalization and adjusted if necessary. Rounding is performed based on the least significant bits, using an adder for rounding overflow control.

VERIFICATION

PATTERNS

DIRECTED TESTS:

For the inputs A,B and C, we have tested for the following patterns of inputs,

Walking Ones and Walking Zeros:

- Sequentially sets one bit to 1 (for walking ones) or to 0 (for walking zeros) and shifts it across the width of n bits.

  Example for n=4:

  0001, 0010, 0100, 1000.

  1110, 1101, 1011, 0111

Alternating Zeros and Ones:

- Generates a fixed pattern of alternating 0s and 1s across the bit width.

  Example for n=4: 1010, 0101

Sliding Ones and Sliding Zeros:

- Creates a window of ones or zeros that "slides" across n bits. The window size is customizable.

  Example for n=4, window_size=2:

  0011, 0110, 1100

Powers of Two Minus One:

- Generates patterns that include powers of two minus one, filling bits with sequential ones until n bits are reached.

  Example for n=4:

  0001, 0011, 0111, 1111

RANDOM TESTS:

- We have generated some random test cases for A,B and C.
- Produces a fully randomized binary string of length n.

COVER POINTS AND CROSS COVERAGE

The coverage is tracked by defining coverage points and cross-coverage'

- **CoverPoints**: Each CoverPoint represents a unique input vector for `A`, `B`, and `C`. The `bins` specify the range of values checked (currently set as `range(32)`.
- **CoverCross**: The `CoverCross` object `top.cross_cover` monitors cross-combinations across `A`, `B`, `C`, and the `s1_or_s2` flag to ensure all possible interactions are tested.

TESTBENCH

Testbench supports both integer and floating-point operations:

- Floating-Point Mode :
  - Defines `EXPONENT_SIZE = 8` and two different mantissa sizes (`MANTISSA_SIZE_BF16 = 7` and `MANTISSA_SIZE_FP32 = 23`) to represent BF16 and FP32 formats.
  - For each sign bit (`0` for positive, `1` for negative), the testbench generates combinations of exponent and mantissa values. Patterns are generated for A,B and C and random test cases are also generated.
- Integer Mode :
  - Defines `MULTIPLICAND_SIZE = 16` and `ADDEND_SIZE = 32`, representing two operands for integer mode.
  - Pattern generation and test case structure are similar to floating point mode.

In **floating-point mode**, exponent edge cases are handled. Specific patterns that may cause underflows or overflows, known as exponent exception cases, are checked and skipped to avoid invalid operations.

APPLYING TEST CASES:

Non-Pipelined Mode:

- Reset the MAC module by setting RST_N low, then high, and enabling the module with EN_mac.

- Apply A, B, and C as inputs to mac_a, mac_b, and mac_c, respectively.
- Check the MAC output against the expected result.

Pipelined Mode:

- Values are continuously fed into the MAC input stages, and each cycle's result is checked.
- After an initial delay, outputs are expected to match the correct pipeline stage results.