**Module: Multiplication**

The module takes two 8-bit input operands, **IN1** and **IN2**, and produces an 8-bit output **New\_Num**, which represents their product.

The multiplication is implemented by generating partial products for each bit of the multiplier and multiplicand. The multiplication algorithm used here is a simple binary multiplication algorithm, where each bit of the multiplier is multiplied with each bit of the multiplicand.

The module uses an **always** block, sensitive to changes in **IN1**, **IN2**, or **New\_Num**, to perform the multiplication. Before the actual computation, a delay of 2 time units is introduced using **#2** to synchronize the operation with other parts of the circuit.

The partial products are generated by performing bitwise AND operations between the bits of **IN1** and **IN2**. Each partial product is assigned to the corresponding element in the **partial\_products** array. The indexing is based on the position of the corresponding bit in the multiplier.

After generating the partial products, the final product is computed by summing up all the partial products using the **+** operator. The result is assigned to the output signal **New\_Num**.

**Module : Arithmetic Right Shift**

The module takes a 4-bit input **SHIFT** and an 8-bit input **Number**. It produces an 8-bit output **New\_Num**, which represents the result of the arithmetic right shift operation on **Number**.

The arithmetic right shift operation shifts the bits of **Number** to the right by a specified number of positions, filling the leftmost positions with the sign bit (the most significant bit) of **Number**. The value of **SHIFT** determines the number of positions to shift.

Inside the **always** block, a delay of 1 time unit is introduced using **#1** to ensure proper shifting.

The **case** statement is used to select the appropriate shift operation based on the value of **SHIFT**. Each case represents a specific shift amount, ranging from 0 to 7. In each case, the **New\_Num** value is assigned based on the specified shift amount.

For example:

* When **SHIFT** is 4'd0 (0 in decimal), the **New\_Num** is set equal to **Number** without any shifting.
* When **SHIFT** is 4'd1 (1 in decimal), the **New\_Num** is set to the concatenation of the sign bit of **Number** and the remaining bits shifted by one position to the right.

The **default** case is included to handle any unexpected values of **SHIFT**. In this case, the **New\_Num** is set to the same value as the arithmetic right shift by 7 positions, which keeps the sign bit intact.

**Module : Rotate Right**

The module takes a 4-bit input **SHIFT** and an 8-bit input **Number**. It produces an 8-bit output **New\_Num**, which represents the result of the right rotation operation on **Number**.

A right rotation operation shifts the bits of **Number** to the right by a specified number of positions, wrapping the rightmost bit to the leftmost position. The value of **SHIFT** determines the number of positions to rotate.

Inside the **always** block, a delay of 1 time unit is introduced using **#1** to ensure proper rotation.

The **case** statement is used to select the appropriate rotation operation based on the value of **SHIFT**. Each case represents a specific rotation amount, ranging from 0 to 7. In each case, the **New\_Num** value is assigned based on the specified rotation amount.

For example:

* When **SHIFT** is 3'd0 (0 in decimal), the **New\_Num** is set equal to **Number** without any rotation.
* When **SHIFT** is 3'd1 (1 in decimal), the **New\_Num** is set to the concatenation of the rightmost bit of **Number** with the remaining bits rotated one position to the right.

The **default** case is included to handle any unexpected values of **SHIFT**. In this case, the **New\_Num** is set to 8'b0, meaning all bits are set to 0.

**Module : Shift**

The module takes a 4-bit input SHIFT, an 8-bit input Number, a 1-bit input Choice to select between left and right shift, and produces an 8-bit output New\_Num, which represents the result of the shift operation.

Inside the always block, the case statement is used to select the appropriate shift operation based on the value of Choice. If Choice is 1'b0, it indicates a left shift operation. If Choice is 1'b1, it indicates a right shift operation.

For left shift:

* A delay of 1 time unit is introduced using #1 to ensure proper shifting.
* Inside the nested case statement, the SHIFT value determines the number of positions to shift.
* Each case represents a specific shift amount, ranging from 0 to 7.
* The New\_Num value is assigned based on the specified shift amount, by concatenating the appropriate number of zeros (0) at the right side of Number.

For right shift:

* Similar to the left shift, a delay of 1 time unit is introduced using #1.
* Inside the nested case statement, the SHIFT value determines the number of positions to shift.
* Each case represents a specific shift amount, ranging from 0 to 7.
* The New\_Num value is assigned based on the specified shift amount, by concatenating the appropriate number of zeros (0) or the sign bit (Number[7]) at the left side of Number.

**ALU Opcodes**

|  |  |
| --- | --- |
| **Opcode** | **Operation** |
| 000 | mov, loadi |
| 001 | add |
| 010 | bitwise and |
| 011 | bitwise or |
| 100 | multiplication |
| 101 | arithmetic right shift |
| 110 | rotate |
| 111 | shift |