Code Explanation:

Table of Contents

Mux 2x1

This code describes a simple 2-to-1 multiplexer:

- A multiplexer selects one of the several input signals and forwards the selected input to a single output line.
- In this 2-to-1 multiplexer:
  - There are two inputs (`I0` and `I1`).
  - There is one select line (`S`).
  - There is one output (`O`).

The functionality is as follows:

- When the select line `S` is `'0'`, the output `O` will be equal to `I0`.
- When the select line `S` is `'1'`, the output `O` will be equal to `I1`.

This behavior is implemented using an `if-else` statement within a VHDL process, which ensures that the output `O` is updated immediately when there is any change in the inputs `I0`, `I1`, or the select line `S`.

4x1 Mux

This code describes a 4-to-1 multiplexer using three 2-to-1 multiplexers:

- The first level consists of two 2-to-1 multiplexers (`Mux2x1_0` and `Mux2x1_1`). These multiplexers select between pairs of the input signals (`I0`, `I1` and `I2`, `I3`) based on the select signal `S0`.
- The second level consists of one 2-to-1 multiplexer (`Mux2x1_final`). This multiplexer selects between the outputs of the first level (`mux2x1_0_out` and `mux2x1_1_out`) based on the select signal `S1`.

The 4-to-1 multiplexer works as follows:

- `S0` determines which of the input pairs (`I0`, `I1` or `I2`, `I3`) are selected in the first level.
- `S1` determines which output from the first level is passed to the final output `O`.

In other words:

- When `S1 = 0`, the final output `O` depends on the selection made by `S0` between `I0` and `I1`.
- When `S1 = 1`, the final output `O` depends on the selection made by `S0` between `I2` and `I3`.

By cascading two levels of 2-to-1 multiplexers, this design effectively creates a 4-to-1 multiplexer.

Full adder:

This code describes the behavior of a Full Adder:

- A Full Adder adds three bits: two significant bits `A` and `B`, and a carry-in bit `Cin`.
- It produces a sum bit `Sum` and a carry-out bit `Cout`.

Logic for `Sum`

```
Sum  <= (A xor B) xor Cin;
```

```
The sum output Sum is calculated using the XOR operation.
```

- `(A xor B)` gives the intermediate sum of `A` and `B` without considering the carry-in.
- `(A xor B) xor Cin` then includes the carry-in bit to produce the final sum.

  Logic for `Cout`

  Cout <= (A and B) or (Cin and (A xor B));

  he carry-out output `Cout` is calculated using the AND and OR operations.

- `(A and B)` produces a carry when both `A` and `B` are `1`.

- `Cin and (A xor B)` produces a carry when either the intermediate sum `(A xor B)` is `1` and the carry-in `Cin` is `1`.
- The OR operation combines these conditions to produce the final carry-out.

Summary

- **Inputs**: `A`, `B`, and `Cin` are the inputs to the Full Adder.
- **Outputs**: `Sum` is the sum output, and `Cout` is the carry-out output.
- **Logic**:
    - `Sum` is the result of the XOR operation involving `A`, `B`, and `Cin`.
    - `Cout` is the result of the OR operation combining the AND of `A` and `B` with the AND of `Cin` and the intermediate sum `(A xor B)`.

This code implements a basic Full Adder, which is a fundamental building block in arithmetic logic units (ALUs) and is used in various digital circuits for binary addition.

ALU1:

The ALU1 VHDL code describes a 1-bit ALU (Arithmetic Logic Unit) named `ALU1`. This ALU performs basic logical and arithmetic operations based on the inputs and control signals provided. Let's break down the code to understand its structure and functionality.

Library and Use Clause:

- `library IEEE;` and `use IEEE.STD_LOGIC_1164.ALL;` include the IEEE library and the `STD_LOGIC_1164` package, providing the standard logic types and operations.
- **Entity `ALU1`**:
- Defines the name of the module as `ALU1`.
- **Ports**:
    - `A`, `B`: 1-bit inputs (`std_logic`).
    - `A_inv`, `B_inv`: 1-bit inversion control inputs (`std_logic`).
    - `CarryIn`: 1-bit carry input (`std_logic`).

- `OP0`, `OP1`: 1-bit operation select inputs (`std_logic`).
- `CarryOut`: 1-bit carry output (`std_logic`).
- `Result`: 1-bit result output (`std_logic`).

Architecture `Behavioral`:

- Defines the implementation details of the `ALU1` entity.
- The architecture is named `Behavioral`, indicating it describes the behavior of the ALU.

- **Component Declarations**:

- **MUX2to1**: A 2-to-1 multiplexer.
- **MUX4x1**: A 4-to-1 multiplexer.
- **FullAdder**: A full adder.

- **Signal Declarations**:

- `not_A`, `not_B`: Signals to hold the inverted values of `A` and `B`.
- `mux2x1_A`, `mux2x1_B`: Signals to hold the outputs of the first level of multiplexers.
- `and_w`, `or_w`, `adder_w`: Signals to hold intermediate results for AND, OR, and ADD operations respectively.

Inversion Logic:

- `not_A <= not A;`: Inverts `A` and assigns it to `not_A`.
- `not_B <= not B;`: Inverts `B` and assigns it to `not_B`.

2. **First Level Multiplexers**:
   - **Mux1**: A 2-to-1 multiplexer that selects between `A` and `not_A` based on `A_inv`, outputting to `mux2x1_A`.
   - **Mux2**: A 2-to-1 multiplexer that selects between `B` and `not_B` based on `B_inv`, outputting to `mux2x1_B`.
3. **Logical Operations**:
   - `and_w <= mux2x1_A and mux2x1_B;`: Performs bitwise AND on `mux2x1_A` and `mux2x1_B`.
   - `or_w <= mux2x1_A or mux2x1_B;`: Performs bitwise OR on `mux2x1_A` and `mux2x1_B`.
4. **Full Adder**:
   - **Adder**: A full adder that sums `mux2x1_A`, `mux2x1_B`, and `CarryIn`. The sum is assigned to `adder_w` and the carry-out is assigned to `CarryOut`.
5. **Final Multiplexer**:
   - **mux4_1**: A 4-to-1 multiplexer that selects between `and_w`, `or_w`, `adder_w`, and `0` based on `OP0` and `OP1`, outputting to `Result`.

Explanation

This 1-bit ALU performs the following operations based on the control signals `A_inv`, `B_inv`, `OP0`, and `OP1`:

- **Inversion Control**:
    - `A_inv`: When set to `1`, inverts `A` before further processing.
    - `B_inv`: When set to `1`, inverts `B` before further processing.
- **Operations**:
    - **AND** (`OP1 = 0, OP0 = 0`): `Result` = `mux2x1_A AND mux2x1_B`.
    - **OR** (`OP1 = 0, OP0 = 1`): `Result` = `mux2x1_A OR mux2x1_B`.
    - **ADD** (`OP1 = 1, OP0 = 0`): `Result` = `mux2x1_A + mux2x1_B + CarryIn`.
    - **Zero** (`OP1 = 1, OP0 = 1`): `Result` = `0`.

This ALU module can be used as a building block for more complex multi-bit ALUs by cascading multiple instances of this 1-bit ALU, each handling one bit of the multi-bit operation.

ALU8:

The provided VHDL code describes an 8-bit ALU (Arithmetic Logic Unit) named `ALU8`. This ALU is built by instantiating eight 1-bit ALU components (ALU1) to handle 8-bit operations.

Library and Use Clause:

- `library IEEE;` and `use IEEE.STD_LOGIC_1164.ALL;` include the IEEE library and the `STD_LOGIC_1164` package, providing the standard logic types and operations.
- **Entity `ALU8`**:
- Defines the name of the module as `ALU8`.
- **Ports**:
    - `A`, `B`: 8-bit inputs (`std_logic_vector(7 downto 0)`).
    - `A_in`, `B_inv`: 1-bit control inputs (`std_logic`) for inverting `A` and `B`.
    - `CarryIn`: 1-bit carry input (`std_logic`).
    - `OP`: 2-bit operation select input (`std_logic_vector (1 downto 0)`).
    - `CarryOut`: 1-bit carry output (`std_logic`).
    - `Result`: 8-bit result output (`std_logic_vector(7 downto 0)`).

Architecture `Behavioral`:

- Defines the implementation details of the `ALU8` entity.
- The architecture is named `Behavioral`, indicating it describes the behavior of the ALU.
- **Component Declaration**:
- **ALU1**: A 1-bit ALU component which is instantiated eight times, one for each bit of the 8-bit ALU.
- **Signal Declaration**:

- `carry`: A 9-bit signal (`std_logic_vector(8 downto 0)`) to hold intermediate carry signals between each 1-bit ALU.
- **Initial Carry Input**:
- `carry(0) <= CarryIn;`: Sets the initial carry input to the value of `CarryIn`.

Instantiating ALU1 Components

The ALU8 is constructed by instantiating eight ALU1 components, each handling one bit of the 8-bit inputs.

```
-- Instantiate ALU1 components for each bit
ALU1_0: ALU1 port map (
    A       => A(0),
    A_inv   => A_in,
    B       => B(0),
    B_inv   => B_inv,
    CarryIn  => carry(0),
    OP0     => OP(0),
    OP1     => OP(1),
    CarryOut => carry(1),
    Result   => Result(0)
);

ALU1_1: ALU1 port map (
    A       => A(1),
    A_inv   => A_in,
    B       => B(1),
    B_inv   => B_inv,
    CarryIn  => carry(1),
    OP0     => OP(0),
    OP1     => OP(1),
    CarryOut => carry(2),
    Result   => Result(1)
);

ALU1_2: ALU1 port map (
    A       => A(2),
    A_inv   => A_in,
    B       => B(2),
    B_inv   => B_inv,
    CarryIn  => carry(2),
    OP0     => OP(0),
    OP1     => OP(1),
    CarryOut => carry(3),
    Result   => Result(2)
);

ALU1_3: ALU1 port map (
    A       => A(3),
    A_inv   => A_in,
    B       => B(3),
```

```
    B_inv    => B_inv,
    CarryIn  => carry(3),
    OP0      => OP(0),
    OP1      => OP(1),
    CarryOut => carry(4),
    Result   => Result(3)
);

ALU1_4: ALU1 port map (
    A        => A(4),
    A_inv    => A_in,
    B        => B(4),
    B_inv    => B_inv,
    CarryIn  => carry(4),
    OP0      => OP(0),
    OP1      => OP(1),
    CarryOut => carry(5),
    Result   => Result(4)
);

ALU1_5: ALU1 port map (
    A        => A(5),
    A_inv    => A_in,
    B        => B(5),
    B_inv    => B_inv,
    CarryIn  => carry(5),
    OP0      => OP(0),
    OP1      => OP(1),
    CarryOut => carry(6),
    Result   => Result(5)
);

ALU1_6: ALU1 port map (
    A        => A(6),
    A_inv    => A_in,
    B        => B(6),
    B_inv    => B_inv,
    CarryIn  => carry(6),
    OP0      => OP(0),
    OP1      => OP(1),
    CarryOut => carry(7),
    Result   => Result(6)
);

ALU1_7: ALU1 port map (
    A        => A(7),
    A_inv    => A_in,
    B        => B(7),
    B_inv    => B_inv,
    CarryIn  => carry(7),
    OP0      => OP(0),
    OP1      => OP(1),
```

```
    CarryOut => carry(8),
    Result   => Result(7)
);
```

Each instance of `ALU1` is connected to a corresponding bit of the 8-bit inputs `A` and `B`. The inversion control signals `A_in` and `B_inv`, and the operation select signals `OP(0)` and `OP(1)` are common to all instances. The carry-out of each `ALU1` instance is connected to the carry-in of the next `ALU1` instance in sequence.

Final Carry Output

```
-- Final carry output
CarryOut <= carry(8);
```

The final carry output of the ALU8 is taken from `carry(8)`.

Explanation

1. **Initialization**:
    - The initial carry input is provided by `CarryIn`.
2. **Bit-wise ALU Operations**:
    - Each `ALU1` instance performs operations on one bit of `A` and `B`, with carry propagation handled by the `carry` signal.
    - The results of these operations are collected in the `Result` signal.
3. **Carry Propagation**:
    - The carry generated by each `ALU1` instance is propagated to the next bit, ensuring correct arithmetic operations across the 8 bits.
4. **Final Carry Out**:
    - The final carry output after all 8 bits have been processed is provided as `CarryOut`.

This modular approach allows for easy scalability and clear separation of logic for each bit, facilitating debugging and extension to larger bit-width ALUs if needed.