ALU Varication with Cocotb

This project demonstrates testing an Arithmetic Logic Unit (ALU) using a coroutine-based testbench written in Cocotb. The ALU is a sequential design implemented in SystemVerilog, and the testbench is responsible for verifying its correct functionality under different operations. Each arithmetic or logical operation is tested independently inside its coroutine.

Simulation

Simulation EDA Playground

Design Overview

ALU Module (RTL)

The ALU supports the following operations:

- **Addition** (`A + B`)
- **Subtraction** (`A - B`)
- **AND** (`A & B`)
- **OR** (`A | B`)
- **XOR** (`A ^ B`)
- **Default Operation**: Outputs `0`

```
module Sequential_ALU(
    input clk,
    input rst_n,
    input [3:0] opcode,
    input  [7:0] A, B,
    output reg [8:0] Result
);

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        Result <= 8'b0;
    end else begin
        case(opcode)
            4'b0000: Result <= A + B;
            4'b0001: Result <= A - B;
            4'b0010: Result <= A & B;
            4'b0011: Result <= A | B;
            4'b0100: Result <= A ^ B;
            default: Result <= 8'b0;
        endcase
    end
end

endmodule
```

Testbench Overview

The testbench for the ALU uses the **Cocotb** Python framework.

Testbench Implementation

The testbench consists of six key tests:

1. **Addition Test**
2. **Subtraction Test**
3. **AND Test**
4. **OR Test**
5. **XOR Test**
6. **Default Operation Test**

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
import random

pass_count = 0
fail_count = 0

@cocotb.coroutine
async def generate_clock(dut):
    clock = Clock(dut.clk, 10, units="ns")  # 100MHz clock
    cocotb.fork(clock.start())

@cocotb.coroutine
async def reset(dut):
    dut.rst_n <= 0
    await FallingEdge(dut.clk)
    dut.rst_n <= 1
    await FallingEdge(dut.clk)

# Example: Addition Test
@cocotb.coroutine
async def add_test(dut, A, B):
    global pass_count
    global fail_count
    dut.opcode <= 0b0000  # Opcode for ADD
    dut.A <= A
    dut.B <= B
    await FallingEdge(dut.clk)
    expected = A + B
    if dut.Result.value == expected:
        print(f"ADD Test PASS: {A} + {B} = {dut.Result.value}")
        pass_count += 1
    else:
        print(f"ADD Test FAIL: {A} + {B} != {dut.Result.value}, expected
{expected}")
        fail_count += 1
```

Running the Tests

Cocotb (**xcelium**).provided Makefile to automate the testing process.

```
TOPLEVEL_LANG ?= verilog
SIM ?=xcelium
PWD=$(shell pwd)

ifeq ($(TOPLEVEL_LANG),verilog)
VERILOG_SOURCES = $(PWD)/design.sv
else ifeq ($(TOPLEVEL_LANG),vhdl)
  VHDL_SOURCES = $(PWD)/design.vhdl
else
    $(error A valid value (verilog or vhdl) was not provided for
TOPLEVEL_LANG=$(TOPLEVEL_LANG))
endif
TOPLEVEL := Sequential_ALU # Module name
MODULE := tb               # Python testbench file

include $(shell cocotb-config --makefiles)/Makefile.sim
```

Executing the tests:

```
make SIM=xcelium TOPLEVEL_LANG=verilog
```

Test Results

After running the testbench, results are printed indicating the number of passed and failed tests for each operation.

```
ADD Test PASS: 112 + 18 = 010000010
SUB Test PASS: 112 - 18 = 001011110
AND Test PASS: 112 & 18 = 000010000
OR Test PASS: 112 | 18 = 001110010
XOR Test PASS: 112 ^ 18 = 001100010
Default Test PASS: 112 ? 18 = 000000000
...
NO. Pass = 600
NO. Fail = 0
```