ALU Project Documentation

Varun G - 6122

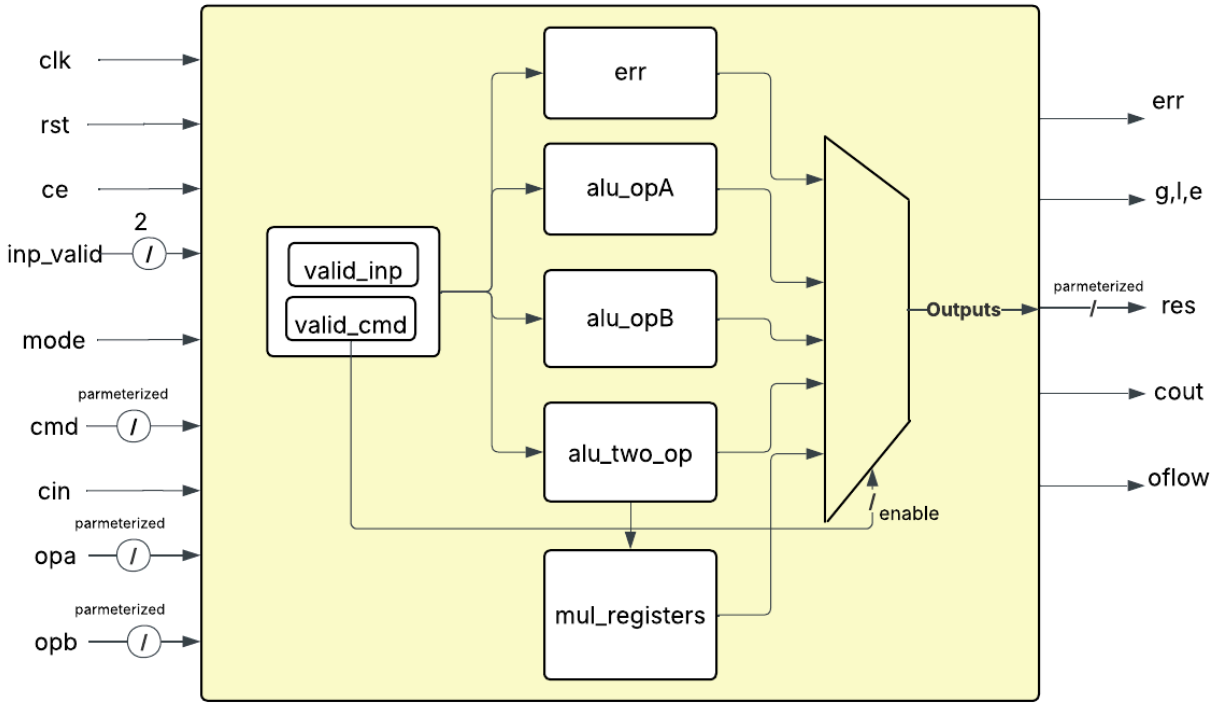
Introduction

The Arithmetic Logic Unit (ALU) is a fundamental combinational digital circuit designed to perform arithmetic and logical operations on binary data. This project presents a scalable, modular ALU implemented in Verilog that supports a variety of operations. The ALU is designed to be parameterised in width and includes error detection and pipelined multiplication support. The ALU is structured into operand-type-specific submodules to evaluate efficiently and maintain modularity.

Objectives

- Implement a Verilog-based ALU supporting multiple arithmetic and logic operations.
- Create modular subblocks based on operand usage (OPA, OPB, both).
- Support optional multiplication with wider output result using a compile-time macro.
- Incorporate input and command validation with error signalling.
- Simulate and verify functional correctness using waveform inspection and flag outputs.
- Ensure no redundant code using code coverage.

Architecture



The architecture is divided into:

- **Inputs**: clk, rst, ce, cmd, cin, mode, opa, opb, and inp_valid
- **Submodules**:
    - alu_opA: Handles single-operand operations using opa
    - alu_opB: Handles single-operand operations using opb
    - alu_two_op: Handles operations requiring both opa and opb
- **Control Logic**:
    - Checks if the command is valid for the current mode
    - Verifies if the correct operand availability (inp_valid) is given

    - Enables the correct submodule

- **Output Logic**:
    - Multiplexes the result from the active submodule
    - Generates flags: cout, oflow, g, l, e, err
- **Pipelining**:
    - Multiplication operations introduce one extra cycle
    - Intermediate results are latched to preserve output timing

---

Working

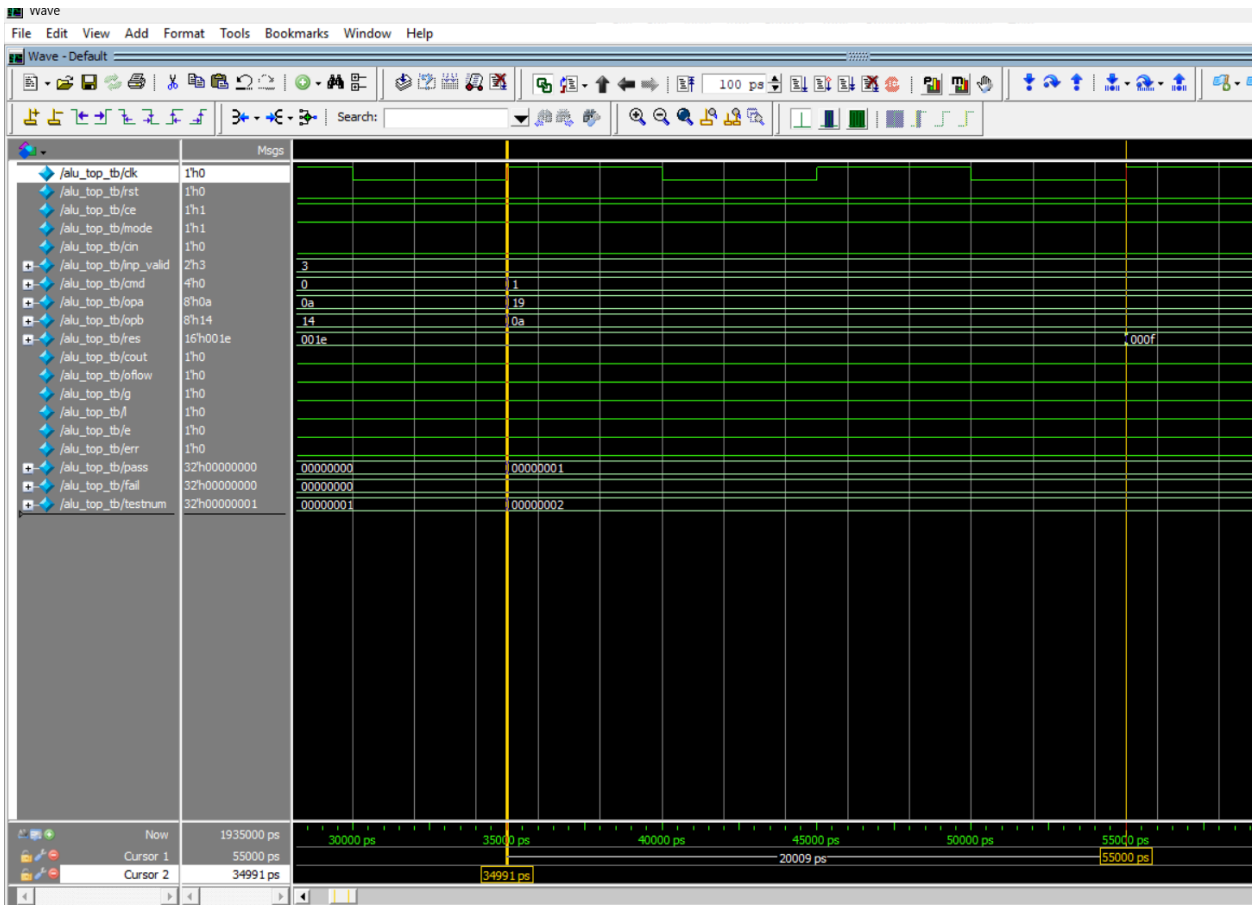
- The operation is determined by the opcode (cmd) and execution mode.
- inp_valid is used to indicate whether OPA, OPB, or both are valid for the current operation.
- The command validator and operand validator activate only the relevant submodule using enable.
- Based on mode:
    - **Arithmetic operations** (mode = 1): ADD, SUB, INC_MUL, SHL_MUL, etc.
    - **Logical operations** (mode = 0): AND, XOR, SHR1, ROL, etc.
- For rotation operations (ROL, ROR), the MSBs of opb are checked to ensure the rotate amount is valid using $clog2(OP_WIDTH).
- Outputs are muxed from the correct submodule.
- If cmd or inp_valid is mismatched or undefined, err is asserted and other flags/results are cleared.

Result

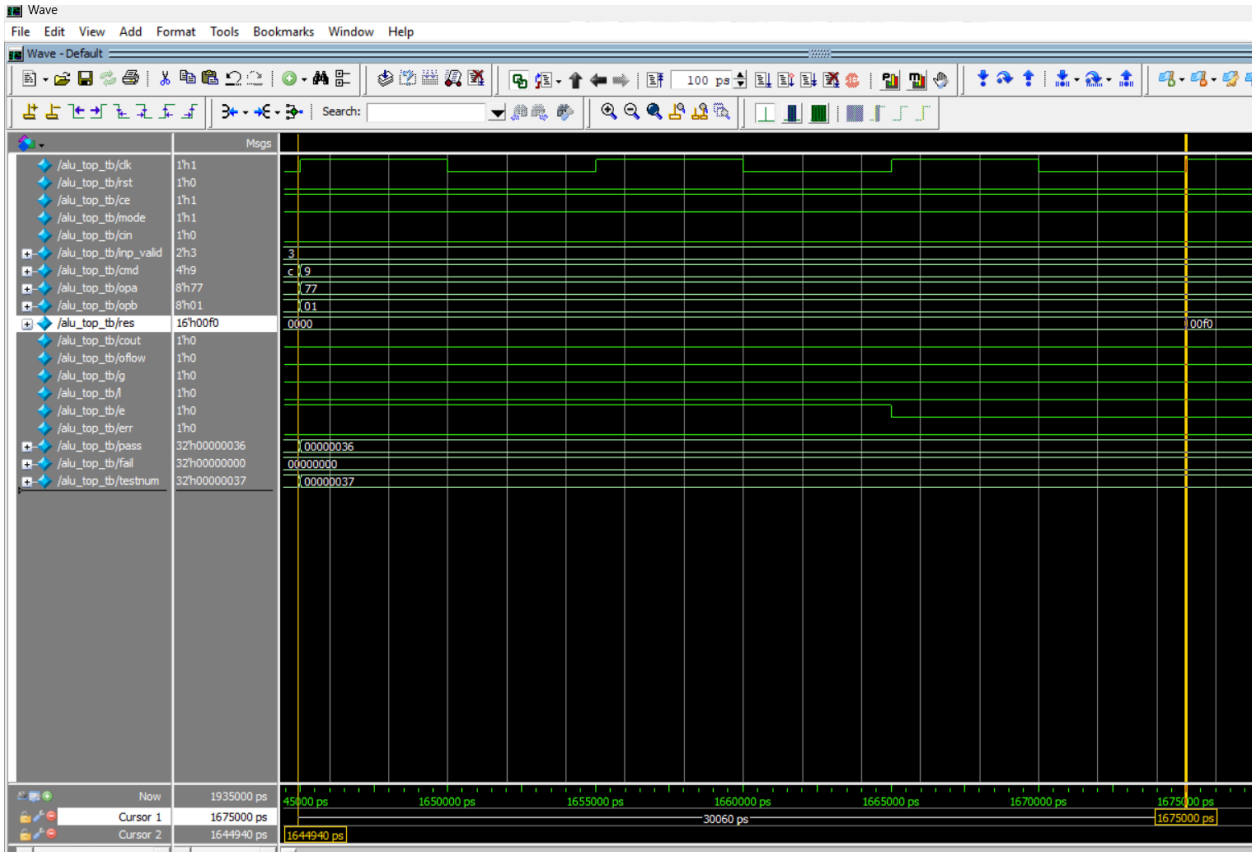
The ALU has been successfully simulated with a wide variety of test cases covering:

- Correct execution for all arithmetic and logic instructions
- Overflow and carry conditions
- Error detection when invalid inp_valid is provided
- Verification of rotation limits
- Output flags (cout, oflow, g, l, e, err) showing correct values
- Waveforms confirm correct behavior across all modules

2 cycle Delay:



3 cycle Delay:




```
# [PASS]          DEC_B_OVF
# [PASS]        INC_A_COUT
# [PASS]        INC_B_COUT
# [PASS]           ROL_ERR
# [PASS]           ROR_ERR
# [PASS]       SUB_CIN_EQ1
# [PASS]       SUB_CIN_EQ0
# [PASS]     SUB_CIN_UNDER
# [PASS]      ADD_SIGN_NEG
# [PASS]       ADD_SIGN_EQ
# [PASS]       SUB_SIGN_EQ
# [PASS]      INC_MUL_MAXA
# [PASS]      INC_MUL_MAXB
# [PASS]       INC_MUL_BIG
# [PASS]     ADD_SIGN_TEST
# [PASS]     SUB_SIGN_TEST
# [PASS]      INC_MUL_TEST
# [PASS]      SHL_MUL_TEST
# [PASS]           ERR_RST
# Total: 62   Passed: 62   Failed: 0
```

Functional correctness is demonstrated via simulation log and waveform traces.

Conclusion

This ALU is:

- Fully modular and parameterized
- Synthesizable across different widths and configurations
- Error-aware with proper flag generation
- Well-structured to allow easy expansion or integration

The top module alu_top.v orchestrates all control and datapath decisions, while operand-type-specific submodules (alu_opA, alu_opB, alu_two_op) perform the actual computation.

Future Improvement

To further enhance the design:

- **Support additional operations**.
- **Apply formal verification** techniques for exhaustive correctness.
- **Optimize for area/power** for FPGA/ASIC synthesis.