TEAM VERILOG GITHUB TASK

Team Members: Salah, Soliman, Khaled, Saeed, Lotfy

Overview

This is your collaborative GitHub onboarding assignment. You will work together as a team to design, implement, test, and document a complete Verilog hardware project using industry-standard Git workflows and repository structure.

Your objective: Build a fully integrated, tested, and documented 4-bit ALU through collaborative GitHub development. Each team member has mandatory individual responsibilities that cannot be delegated.

Repository Structure (MANDATORY)

Your repository **must** follow this exact industry-standard structure:

```
project-root/
├── README.md
├── .gitignore
├── LICENSE
├── rtl/
│   ├── alu_4bit.v
│   ├── adder_subtractor.v
│   ├── logic_unit.v
│   └── shifter.v
├── tb/
│   ├── alu_4bit_tb.v
│   ├── adder_subtractor_tb.v
│   ├── logic_unit_tb.v
│   └── shifter_tb.v
├── docs/
│   ├── architecture.md
│   ├── module_specifications.md
│   ├── test_plan.md
│   ├── verification_report.md
│   └── integration_guide.md
├── sim/
│   ├── results/
│   │   └── [waveforms and logs]
│   └── scripts/
│       └── run_tests.sh
└── .github/
    └── pull_request_template.md
```

Reference repositories with similar structure:

- https://github.com/ultraembedded/biriscv (professional RISC-V core)
- https://github.com/ultraembedded/core_uriscv (clean RTL organization)
- https://opencores.org (browse any core project for standard structure)

- OpenCores HDL Guidelines: https://cse.usf.edu/~haozheng/teach/cda4253/doc/opencores_coding_guidelines.pdf

Folder purposes:

- `rtl/` - **ONLY** synthesizable Verilog RTL code (no testbench constructs)
- `tb/` - **ONLY** testbench code (simulation-only)
- `docs/` - **ALL** documentation (Markdown format)
- `sim/` - Simulation outputs, scripts, waveforms
- `.github/` - GitHub workflow templates

Project: 4-Bit ALU with Modular Architecture

System Specifications

Top-level module: `alu_4bit.v`

Interface:

```
module alu_4bit(
    input [3:0] a,           // First operand
    input [3:0] b,           // Second operand
    input [2:0] op,          // Operation select
    output reg [3:0] result, // ALU result
    output reg carry,        // Carry/borrow flag
    output reg zero          // Zero flag
);
```

Operation Encoding:

op[2:0]	Operation	Module Responsible
000	ADD	adder_subtractor
001	SUB	adder_subtractor
010	AND	logic_unit
011	OR	logic_unit
100	XOR	logic_unit
101	NOT (of a)	logic_unit
110	SHL (a<<1)	shifter
111	SHR (a>>1)	shifter

Flag Specifications:

- `carry`: Set on arithmetic overflow/underflow (ADD/SUB) or shift bit out (SHL/SHR)
- `zero`: Set when `result == 4'b0000`

Modular Design Requirements

Your ALU **must** be built from three sub-modules that are instantiated in the top-level:

1. **adder_subtractor.v** - Handles ADD and SUB operations
2. **logic_unit.v** - Handles AND, OR, XOR, NOT operations
3. **shifter.v** - Handles SHL and SHR operations

The top-level `alu_4bit.v` instantiates these modules and uses multiplexers to select the appropriate output based on `op`.

Individual Mandatory Assignments

Salah's Responsibilities

1. RTL Module: `adder_subtractor.v`

Requirements:

- Module interface:

```
module adder_subtractor(    input [3:0] a,    input [3:0] b,    input sub,
          // 0=add, 1=subtract    output [3:0] result,    output carry);
```

- Implement 4-bit addition and subtraction
- Use full-adder logic (can be ripple-carry or any method)
- For subtraction: compute `a - b` using 2's complement method (`a + (~b) + 1`)
- `carry` output must correctly indicate overflow for ADD and borrow for SUB
- **Must include module header comment with: author name, date, description**

2. Testbench: `adder_subtractor_tb.v`

Requirements:

- Self-checking testbench with automatic pass/fail
- Minimum 20 test cases covering:
  - All zero inputs
  - All one inputs
  - Maximum positive values (4'hF + 4'hF)
  - Edge cases for carry/borrow
  - Random combinations
- Print: "TEST PASSED" or "TEST FAILED: [description]" for each case
- Final summary: "X/Y tests passed"

3. Documentation: Section in `docs/module_specifications.md`

- Detailed description of adder_subtractor module
- Truth tables or equation for add/subtract logic
- Carry flag behavior explanation
- Block diagram (ASCII art or description acceptable)

4. Git Requirements:

- Create branch: `feature/adder-subtractor`
- Minimum 3 commits with clear messages
- Submit PR with description of implementation
- Review at least 2 other PRs with meaningful technical comments

---

Soliman's Responsibilities

1. RTL Module: `logic_unit.v`

Requirements:

- Module interface:

```
module logic_unit(    input [3:0] a,    input [3:0] b,    input [1:0] op,
    // 00=AND, 01=OR, 10=XOR, 11=NOT    output [3:0] result);
```

- Implement all four bitwise logical operations
- NOT operation operates only on input `a` (ignore `b`)
- Use continuous assignment or combinational always block
- **Must include module header comment with: author name, date, description**

2. Testbench: `logic_unit_tb.v`

Requirements:

- Self-checking testbench
- Minimum 16 test cases (4 operations × 4 test vectors each)
- Must test:
  - All zeros
  - All ones
  - Alternating patterns (4'b1010, 4'b0101)
  - Random values
- Print pass/fail status for each test
- Final summary line

3. Documentation: Section in `docs/module_specifications.md`

- Description of logic_unit module
- Truth table for each operation (at bit level)
- Example calculations
- Design decisions

4. Git Requirements:

- Create branch: `feature/logic-unit`
- Minimum 3 commits with clear messages
- Submit PR with description
- Review at least 2 other PRs

Khaled's Responsibilities

1. RTL Module: `shifter.v`

Requirements:

- Module interface:

```
module shifter(    input [3:0] a,    input dir,              // 0=left,
1=right    output [3:0] result,    output shift_out        // bit shifted
out);
```

- Implement logical shift left and logical shift right (by 1 position)
- Shift left: `result = {a[2:0], 1'b0}`, `shift_out = a[3]`
- Shift right: `result = {1'b0, a[3:1]}`, `shift_out = a[0]`
- **Must include module header comment with: author name, date, description**

2. Testbench: `shifter_tb.v`

Requirements:

- Self-checking testbench
- Minimum 10 test cases (5 for left shift, 5 for right shift)
- Test patterns that verify:
  - Correct bit shifting
  - Shift_out captures correct bit
  - Zero filling in vacant positions
- Print pass/fail for each test
- Final summary

3. Documentation: Section in `docs/module_specifications.md`

- Description of shifter module
- Diagram showing shift operations with example
- Explanation of shift_out flag
- Use cases

4. Git Requirements:

- Create branch: `feature/shifter`
- Minimum 3 commits with clear messages
- Submit PR with description
- Review at least 2 other PRs

Saeed's Responsibilities

1. RTL Module: `alu_4bit.v` (Top-level Integration)

Requirements:

- Top-level module that instantiates all three sub-modules
- Interface as specified in System Specifications above
- Use multiplexer logic or case statement to select correct sub-module output based on `op[2:0]`
- Connect carry flag appropriately:
  - For ADD/SUB: use adder_subtractor carry output
  - For SHL/SHR: use shifter shift_out
  - For logical ops: carry = 0
- Implement zero flag: `zero = (result == 4'b0000)`
- **Must include detailed module header with: author, date, description, pin list**
- **Code must be clean, well-commented, and follow consistent style**

2. Testbench: `alu_4bit_tb.v` (Complete System Test)

Requirements:

- Comprehensive self-checking testbench for entire ALU
- Minimum 50 test vectors covering:
  - Every operation (8 operations × 6+ vectors each)
  - Flag verification for each test
  - Edge cases (overflow, underflow, all zeros, all ones)
  - Sequential operation changes
- Must verify:
  - Correct result for each operation
  - Correct carry flag behavior
  - Correct zero flag behavior
- Print detailed results: operation name, inputs, expected vs actual output
- Final summary: "XX/50 tests passed"
- **This is the most critical testbench - it validates the entire system**

3. Documentation: `docs/architecture.md` (Complete file)

Requirements:

- System overview and block diagram
- Description of modular architecture
- Explanation of how sub-modules are integrated
- Data flow description
- Interface specification table
- Design decisions and rationale

4. Documentation: `docs/integration_guide.md` (Complete file)

Requirements:

- How to instantiate the ALU in another design
- Port connection examples
- Timing information (purely combinational, mention this)

- Usage examples with code snippets
- Common pitfalls and recommendations

5. Git Requirements:

- Create branch: `feature/alu-top-level`
- Wait for other modules to be available (or create stubs)
- Minimum 5 commits (this is integration work)
- Submit PR with detailed integration testing results
- Review at least 2 other PRs

Lotfy's Responsibilities

1. Repository Setup and Structure

Requirements:

- Create the GitHub repository
- Add all team members as collaborators
- Set up complete directory structure as specified
- Create

  ```
  .gitignore
  ```

  file for Verilog projects:

  ```
  # Simulation files*.vcd*.vvp*.log*.out# Tool-
  specificwork/transcriptvsim.wlf*.wlf# OS files.DS_StoreThumbs.db
  ```

- Create `LICENSE` file (choose MIT or Apache 2.0)
- Create

  ```
  .github/pull_request_template.md
  ```

  :

  ```
  ## Description[Describe what this PR does]## Type of Change- [ ] New feature-
  [ ] Bug fix- [ ] Documentation update## Testing Done[Describe testing
  performed]## Checklist- [ ] Code follows project style guidelines- [ ] Self-
  review completed- [ ] Comments added for complex logic- [ ] Documentation
  updated- [ ] Tests pass locally
  ```

2. Documentation: `README.md` (Complete file)

Requirements:

- Project title and description
- Team member names and their responsibilities (list what each person implemented)
- Directory structure explanation
- Prerequisites (tools needed: iverilog, gtkwave, etc.)

- Quick start guide:
  - How to clone
  - How to run simulations
  - How to view waveforms
- Build/simulation commands (specific commands for each testbench)
- Project status / completion checklist
- Links to detailed documentation in `docs/`

3. Documentation: `docs/test_plan.md` (Complete file)

Requirements:

- Testing strategy overview
- Description of each testbench and what it validates
- Test coverage matrix (which tests cover which operations)
- How to run all tests
- Expected results
- Regression testing procedure

4. Documentation: `docs/verification_report.md` (Complete file)

Requirements:

- Summary of all testing performed
- Test results for each module
- Test results for integrated system
- Code coverage analysis (which lines/conditions tested)
- Known issues or limitations
- Sign-off statement when all tests pass

5. Simulation Scripts: `sim/scripts/run_tests.sh`

Requirements:

- Bash script that runs all testbenches automatically
- Compiles each module and testbench
- Runs simulation
- Collects results
- Prints summary of pass/fail for all tests
- Example structure:

```
#!/bin/bashecho "Running ALU Test Suite..."# Test adder_subtractoriverilog -o
sim/results/adder_sub.vvp rtl/adder_subtractor.v tb/adder_subtractor_tb.vvvp
sim/results/adder_sub.vvp# [repeat for other modules]echo "All tests
complete."
```

6. Git Requirements:

- Create branch: `feature/documentation-and-scripts`
- Update documentation as other modules are completed
- Minimum 5 commits (documentation is iterative)
- Submit PR with complete documentation

- Review at least 2 other PRs

Git Workflow Requirements (ALL MEMBERS)

Branch Strategy

Mandatory rules:

1. **NEVER commit directly to `main` branch**
2. All work happens on feature branches
3. Branch naming: `feature/<your-component>` or `fix/<issue-description>`
4. One branch per major responsibility
5. Delete branch after successful merge

Commit Message Standards

Every commit must follow this format:

```
<type>: <short description>

<detailed explanation if needed>
```

Types:

- `feat:` - New feature
- `fix:` - Bug fix
- `docs:` - Documentation only
- `test:` - Test additions or fixes
- `refactor:` - Code restructuring without behavior change

Examples:

```
feat: implement 4-bit adder with carry logic

Added full adder implementation using ripple carry architecture.
Handles both addition and subtraction via 2's complement.
test: add edge case tests for logic unit

Added tests for all-zero and all-one inputs across all
logic operations to verify correct bitwise behavior.
```

Pull Request Requirements

Every PR must include:

1. **Title:** Clear, descriptive (e.g., "Implement adder_subtractor module with testbench")
2. **Description with these sections:**

```
## What was implemented
[List of files added/modified]

## Testing performed
[What tests were run, results summary]

## Notes for reviewers
[Anything specific to look at]
```

3. **Reviewers:** Assign at least 2 team members
4. **Requirements before merge:**
    - At least 2 approvals from OTHER team members
    - All review comments addressed
    - No merge conflicts with main
    - Code follows project standards

Code Review Requirements

When reviewing a PR, you must:

1. Actually read and understand the code
2. Check that it follows Verilog best practices:
    - Proper use of blocking vs non-blocking assignments
    - No simulation constructs in RTL files
    - Proper module hierarchy
    - Clean, readable code with comments
3. Verify testbench is self-checking and comprehensive
4. Check that documentation is clear and complete
5. Leave at least 2 meaningful comments (questions, suggestions, or approvals)

Review comment quality examples:

❌ Bad: "Looks good" ✅ Good: "The carry logic in line 23 correctly implements overflow detection for 4-bit addition. However, consider adding a comment explaining the 2's complement method for subtraction."

❌ Bad: "LGTM" ✅ Good: "Testbench covers all required cases. Suggestion: add a test for when both inputs are maximum value (4'hF) to verify carry flag behavior."

Coding Standards (ALL RTL CODE)

Module Header Template

Every `.v` file must start with:

```
//============================================================================
// Module: [module_name]
// Description: [brief description of functionality]
// Author: [Your name]
// Date: [Creation date]
//============================================================================
// Inputs:
//   [port_name] - [description]
// Outputs:
//   [port_name] - [description]
//============================================================================
```

Verilog Style Requirements

1. **Indentation:** 4 spaces (no tabs)
2. Naming:
   - Modules: lowercase with underscores (`adder_subtractor`)
   - Signals: lowercase with underscores (`shift_out`)
   - Parameters/Constants: UPPERCASE (`WIDTH`)
3. Always blocks:
   - Use `always @(*)` for combinational logic
   - Use non-blocking assignments (`<=`) only for sequential logic
   - Use blocking assignments (`=`) for combinational logic in always blocks
4. Comments:
   - Every module needs header comment
   - Complex logic needs inline comments
   - One-line comments use `//`
   - Multi-line comments use `/* */`

What NOT to include in RTL files

❌ `$display`, `$monitor`, `$finish` (testbench only) ❌ `#delay` statements (testbench only) ❌ `initial` blocks (testbench only) ❌ `$random` or other system tasks (testbench only)

Testbench Requirements (ALL TESTBENCHES)

Testbench Structure Template

```
//============================================================================
// Testbench: [module_name]_tb
// Description: Self-checking testbench for [module_name]
// Author: [Your name]
// Date: [Date]
//============================================================================

`timescale 1ns/1ps

module [module_name]_tb;
    // Declare testbench signals
    reg [inputs];
    wire [outputs];
```

```
    // Instantiate module under test
    [module_name] uut (
        .port(signal),
        // ...
    );

    // Test variables
    integer passed = 0;
    integer failed = 0;

    // Test procedure
    initial begin
        $display("========================================");
        $display("Testing [module_name]");
        $display("========================================");

        // Test case 1
        [inputs] = [values];
        #10;  // Wait for propagation
        if ([outputs] == [expected]) begin
            $display("PASS: Test 1 - [description]");
            passed = passed + 1;
        end else begin
            $display("FAIL: Test 1 - Expected %b, Got %b", [expected],
[outputs]);
            failed = failed + 1;
        end

        // [More test cases...]

        // Final summary
        $display("========================================");
        $display("Tests Passed: %0d/%0d", passed, passed+failed);
        $display("========================================");
        $finish;
    end

    // Optional: Waveform dump
    initial begin
        $dumpfile("sim/results/[module_name].vcd");
        $dumpvars(0, [module_name]_tb);
    end
endmodule
```

Documentation Standards (ALL DOCUMENTATION)

Markdown Requirements

1. Use proper heading hierarchy (`#`, `##`, `###`)
2. Include code blocks with syntax highlighting:

   ```
   ```verilogmodule example(input a, output b);```
   ```

3. Use tables for specifications:
   ```

```
| Signal | Width | Description ||--------|-------|-------------|| a      | 4
    | First input |
```

4. Include diagrams (ASCII art acceptable):

```
  ┌──────────────┐  | Adder/Sub   |─┤            ├─  |           |
└──────────────┘
```

Required Documentation Sections

Each documentation file must include:

- Title
- Overview/Purpose
- Detailed content (specific to file purpose)
- Examples (where applicable)
- Author information (who wrote which section)

Integration and Merge Order

Recommended merge sequence:

1. **First:** Lotfy's documentation structure and README (establishes framework)
2. **Second:** Individual module PRs (Salah, Soliman, Khaled) - can happen in parallel
3. **Third:** Saeed's integration PR (after all modules available)
4. **Final:** Updated documentation with final test results (Lotfy)

Before final submission:

- All PRs merged to `main`
- All tests passing
- All documentation complete
- README has accurate team member contributions

Final Deliverables Checklist

Your repository must have:

Code (rtl/ and tb/)

- [ ] `adder_subtractor.v` with proper header and comments
- [ ] `logic_unit.v` with proper header and comments
- [ ] `shifter.v` with proper header and comments
- [ ] `alu_4bit.v` (top-level) with proper header and comments
- [ ] `adder_subtractor_tb.v` (self-checking, 20+ tests)
- [ ] `logic_unit_tb.v` (self-checking, 16+ tests)
- [ ] `shifter_tb.v` (self-checking, 10+ tests)
- [ ] `alu_4bit_tb.v` (self-checking, 50+ tests)

Documentation (docs/)

- [ ] `architecture.md` (complete system description)
- [ ] `module_specifications.md` (all three sub-modules documented)
- [ ] `test_plan.md` (testing strategy and procedures)
- [ ] `verification_report.md` (all test results)
- [ ] `integration_guide.md` (usage instructions)

Root Files

- [ ] `README.md` (complete with quick start guide)
- [ ] `.gitignore` (proper Verilog exclusions)
- [ ] `LICENSE` (open source license)

Scripts (sim/scripts/)

- [ ] `run_tests.sh` (automated test execution)

Git History

- [ ] All work done on feature branches
- [ ] All PRs properly reviewed (2+ approvals each)
- [ ] Clean commit history with proper messages
- [ ] All branches merged and deleted

Success Criteria

Your submission is complete when:

1. **All modules are implemented and tested** - each sub-module and top-level work correctly
2. **All testbenches pass** - self-checking tests show 100% pass rate
3. **Documentation is comprehensive** - someone unfamiliar with the project can understand and use it
4. **Git history shows collaboration** - proper branching, PRs, reviews, and commits
5. **Repository is professionally organized** - follows industry standards

---

Important Notes

Individual Accountability

- **Each person's work will be evaluated separately through Git history**
- Your commits, PRs, and code reviews prove your contribution
- Cannot delegate your assigned module to others
- Must complete all your assigned responsibilities

Collaboration Requirements

- Help each other learn Git, Verilog, and GitHub workflow
- Review each other's code constructively
- Communicate about interfaces between modules
- Resolve issues together
- **But:** Each person must complete their own assigned code and documentation

Quality Over Speed

- Well-tested, documented code is better than fast, broken code
- Take time to write proper commit messages
- Write meaningful code reviews
- Test thoroughly before submitting PR

Learning Objectives

By completing this project, you will:

- Understand professional Git workflows (branching, PRs, reviews)
- Learn Verilog RTL design and testbench development
- Experience collaborative hardware design
- Create a portfolio-worthy project
- Practice industry-standard documentation

The final state of your `main` branch will be reviewed. Your Git history, code quality, testing, documentation, and collaboration practices will all be evaluated.

Good luck, and remember: professional hardware teams work exactly like this.