SemiColab Tile User Logic Guide

Version 1.0

Contents

List of Figures

List of Tables

Glossary

- **MSB (Most Significant Bit)** — The bit in a binary number with the highest positional value. For example, in an 8-bit value, bit 7 is the MSB.
- **LSB (Least Significant Bit)** — The bit in a binary number with the lowest positional value. For the same 8-bit value, bit 0 is the LSB.
- **Tile Wrapper** — A standard interface layer which connects an user-defined RTL module to the broader system. It implements consistent communication, control, and integration within a chip's architecture.
- **Tile** — A modular RTL block that implements a specific functionality and follows a standardized interface for integration into larger chip architectures. Designed for reuse and scalability in SoC designs.
- **Register** — A storage element in hardware that holds a value across clock cycles. Registers are used to store inputs, outputs, intermediate results, and control signals.
- **CSR (Control and Status Register)** — A register used for sending control signals to a hardware module or reading back status information. They monitor operation progress.
- **FSM (Finite State Machine)** — A logic-based control structure that transitions between a finite set of states based on inputs and internal conditions.
- **ALU (Arithmetic Logic Unit)** — A digital circuit that performs arithmetic and logical operations such as addition, subtraction, and bitwise logic. They form the core of the datapath in processors and are controlled via input signals that select the operation and return status flags.
- **High Impedance (Z State)** — A condition where an output signal line is electrically disconnected. In simulation, it is represented as 'Z'. It allows multiple devices to share a connection without interfering, common in tri-state buffers and bus architectures.

1 Introduction

This document provides a comprehensive guide for the correct usage and integration of user-defined RTL modules using the SemiColab User Tile standard interface. The primary goal is to establish a common structure for integrating user-defined RTL modules within a reusable and scalable chip design framework. Two integration examples are provided: a Finite State Machine (FSM) showcasing a sequential logic integration, and an Arithmetic Logic Unit (ALU) representing a combinational logic implementation. Both designs are fully adapted to the User Tile interface.

This FSM performs a bitwise shifting operation, processing data serially from either data_reg_a or data_reg_b depending on user selection, and producing the result on data_reg_c. All of the FSM control signals are managed through Control and Status Registers (CSR).

The FSM module establishes a reusable reference for designers participating in SemiColab project. The user's logic, CSR mapping conventions, and inputs/outputs selection can be adapted to a variety of RTL components while preserving compatibility with the SemiColab's integration flow.

One of the key motivations for following the SemiColab User Tile conventions is to enable seamless interaction between the user's RTL module and a graphical user interface (GUI) implemented in Python, which communicates directly with the FPGA. By adapting the RTL to the defined port mappings and tile templates, designers ensure that their modules can be easily tested, configured, and visualized in real time through the Python GUI.

The following sections will describe the interface specifications, testbench architecture, and the implementation of both user modules. The goal is to guide designers through the full integration flow, from RTL development to verification using the SemiColab infrastructure.

All the templates required to implement the user tile standard, and this user guide are available in the official GitHub repository:

`github.com/Mifral-Tech/semicolab-cocyten2025`

Users are encouraged to clone the repository and follow the structure provided to implement and test their own RTL modules.

2 User Tile Port Description

2.1 Port Conventions

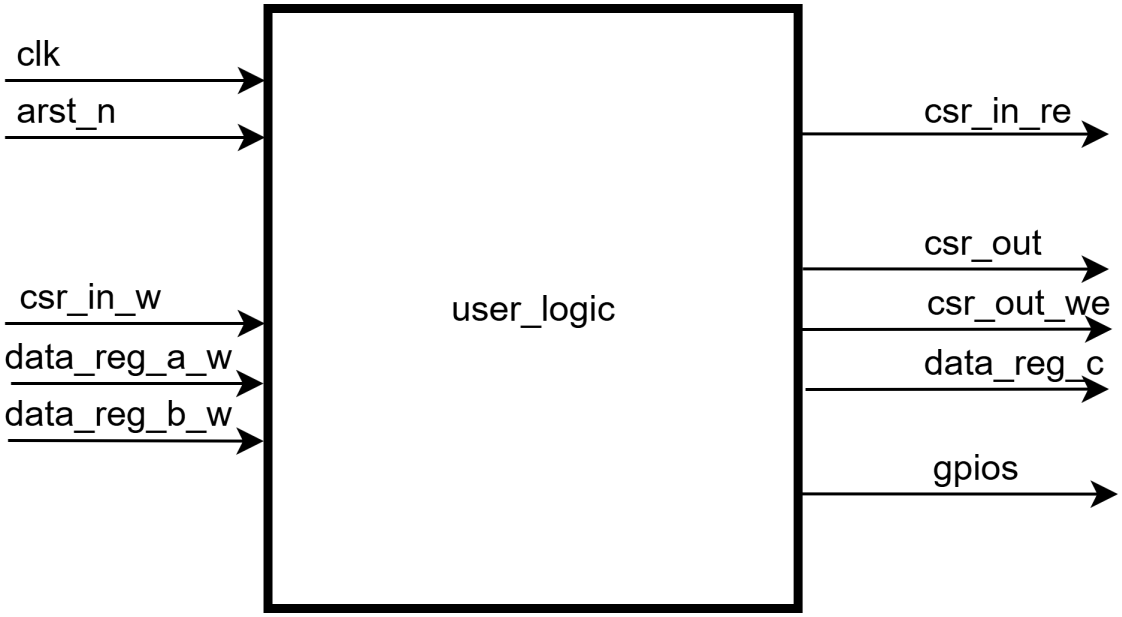

Figure 1: User Tile I/O.

Port	Input/Output	Description
`clk`	Input	Clock signal
`arst_n`	Input	Reset signal
`csr_in`	Input	Control status register (4 pulse bits, 8 stable bits, 4 clear on read bits)
`data_reg_a`	Input	Input data register for the module, 32 bits wide
`data_reg_b`	Input	Input data register for the module, 32 bits wide
`csr_out`	Output	Control status register (12 stable bits, 4 clear on read bits)
`csr_out_we`	Output	Enables writing on `csr_out`
`csr_in_re`	Output	Enables reading of csr_in
`data_reg_c`	Output	Output register for the module, 32 bits wide

Table 1: Description of the ports available in the user tile.

2.2 csr_in Bits Conventions

Bits	Type	Description
[15:12]	Single pulse	The four most significant bits of csr_in, single pulse meaning these bits remain asserted for only one clock cycle and are automatically cleared on the next cycle, after set when the next clock cycle starts bus is cleared, ideal to set instructions.
[11:4]	Stable	Stable meaning they can be set with the same value with no alteration. Can be utilized to set conditions or values for the instruction to execute.
[3:0]	Clear on read	The four least significant bits, clear on read meaning only when circuit reads the input data correctly, these bits will be set to zero, they work as input data but more as a control signal (initial conditions, orders).

Table 2: Description of the csr_in bits of the user tile.

2.3 csr_out Bits Conventions

Bits	Type	Description
[15:4]	Stable	The twelve most significant bits, stable meaning they hold values like normal registers, these can serve as flags or as output values.
[3:0]	Clear on read	The four least significant bits, clear on read meaning only when counterpart reads the input data correctly, these bits will be set to zero, these work as an output but more as an indicator (flags).

Table 3: Description of the csr_out bits of the user tile.

The `csr_in` and `csr_out` registers might initially seem complex, so additional details are provided below.

CSR stands for *Control Status Register*. These signals govern the overall behavior of the design, enabling both control inputs and status outputs.

To understand the role of CSR, it is important to recall that there is a master-slave relationship between the external system and the tile. The tile acts as the slave, responding to commands and data from the master. However, this relationship is bidirectional: the slave not only receives control signals but also returns status information to the master via CSR outputs.

The operation and interaction of these registers will be explained step by step and illustrated with waveform diagrams to facilitate understanding.

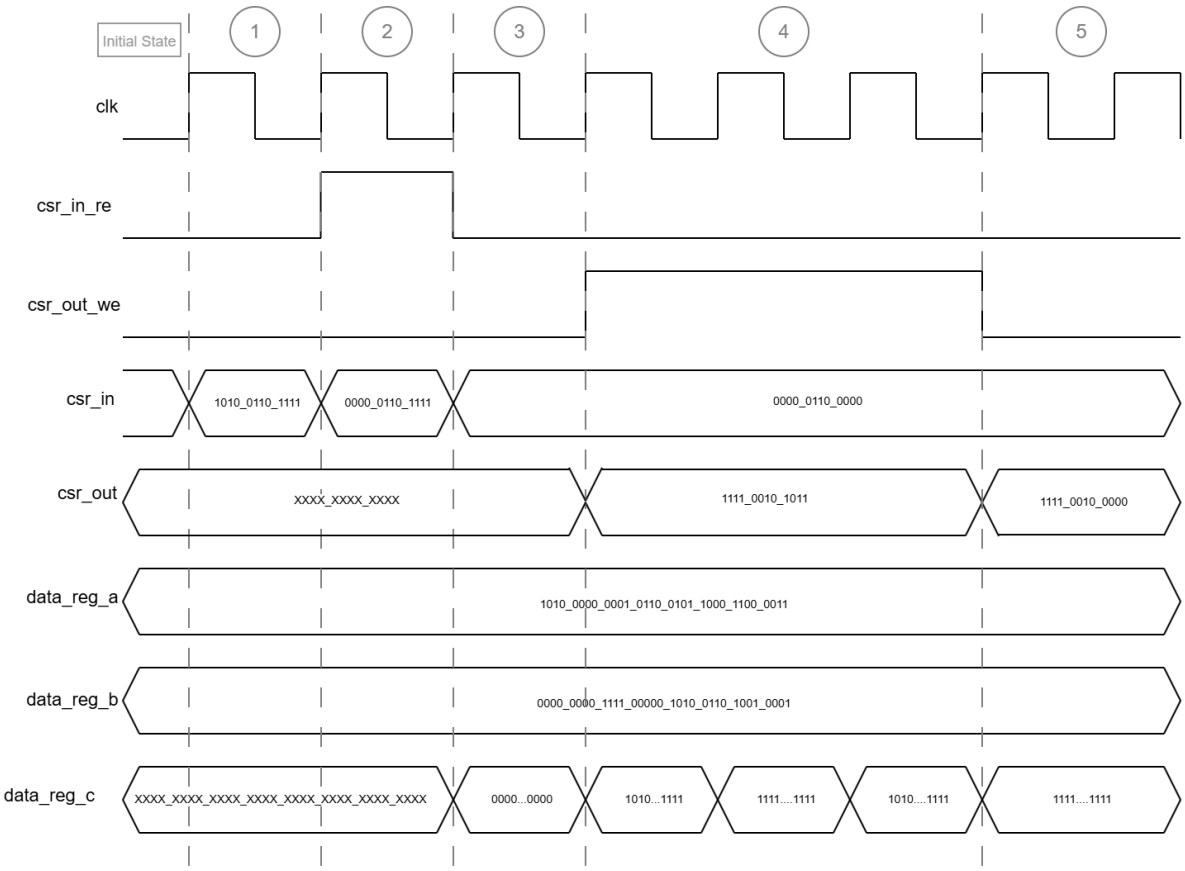

Figure 2: User tile registers sequence.

- **Initial State:** Arbitrary data is initially loaded into `data_reg_a` and `data_reg_b`. On the other hand, `csr_out` and `data_reg_c` remain undefined (`XXXX_XXXX..._XXXX`) as the system has not yet performed any operation.
- **Step 1:** Arbitrary data (`1010_0110_1111`) is written to `csr_in` in order to trigger a system action.
- **Step 2:** Since bits `[15:12]` of `csr_in` are single-pulse control bits, they are intended to be cleared on the next clock cycle (the tile does not clear these bits automatically; the external system must ensure they are deasserted after one cycle). Simultaneously, the `csr_in_re` signal is set high, indicating that the system is reading `csr_in` to begin executing the specified action.
- **Step 3:** After `csr_in` is read, the clear-on-read bits `[3:0]` are cleared. The defined operation is then executed. Note that this process may take an arbitrary number of clock cycles to complete; during this time, `data_reg_c` may exhibit intermediate values. At this stage, `csr_out` is also updated with some status (1111_0110_0000).
- **Step 4:** After `csr_in[3:0]` are cleared and the operation has started, the `csr_out_we` signal is asserted, indicating that data is being written to `csr_out` while the operation is in progress. (Note that the duration of the `csr_out` signal being high does not necessarily indicate that the operation has completed; it only enables status updates to be written to `csr_out`.)
- **Step 5:** After the external system reads `csr_out`, the clear-on-read bits `[3:0]` are automatically reset to zero, completing the transaction.

These signals may seem unnecessary or useless. However their implementation is focused in the communication whit an external master, imagine a computer being connected, with these signals a continuous sequence can be implemented or individual orders and the status of the circuit can be cuncurrently evaluated.

3 Testbench Overview for SemiColab User Tile

As part of SemiColab User Guide, a standardized testbench has been developed to verify user logic modules integrated into the User Tile template. This testbench is designed to be modular, reusable and compatible with any RTL module that respects the SemiColab port conventions.

The testbench includes the following components:

- **Clock and Reset Generation:** Use the RTL tile template, based on the ports given by the SemiColab ip tile wrapper, define your logic, you can implement a variety of RTL modules as long as the port conventions are being followed.
- **CSR Logic:** A model for csr_in and csr_out is implemented to replicate the typical SemiColab behavior, including:
  - Single pulse signals: Automatically cleared one cycle after being asserted.
  - Clear on read signals: Automatically cleared when read.
  - Stable signals: Remain stable until overwritten.
- **Interface Tile:** Task and functions are provided through an interface module to simplify connection with the DUT. These include operations such as writing data to input registers, applying the CSR logic commands, and reading module outputs such as data_reg_c and csr_out.
- **User Module Adaptability:** The same testbench infrastructure can be reused across multiple modules. This promotes consistency, scalability, and reusability, while ensuring all modules conform to the SemiColab verification protocol.
- **Waveforms:** The testbench supports full waveform generation and signal verification through simulations tools such as Vivado, allowing detailed analysis of the user module.

To solidy module verification, a dedicated interface has been designed to interact with the user DUT. This interface includes a set of reusable tasks and functions that simplify the interaction with the user tile. This unified testbench structure ensures:

- **Modularity:** New user modules can be tested with minimal modifications.
- **Reusability:** The same interface logic and testbench can support multiple designs, promoting consistency.
- **Scalability:** Additional tests or RTL components can be included while maintaining the testbench and interface architecture.

This testbench architecture not only facilitates module verification, but also ensures readiness for future hardware interaction through the SemiColab Python GUI connected to the FPGA platform.

The folowwing sections describe in detail the interface module, followed by the practical implementation and verification of multiple RTL designs, including both combinational and sequential examples.

4 Interface

An Interface in SystemVerilog, is used to encapsulate all the signals into a block. All the signals within the block are grouped together to form the interface, this becomes very useful when we work with multiple projects because the same interface can be re-used for other projects. It also becomes easier to connect with the DUT and our Testbench.

4.1 Port Directions

Interface signals can be used with many verification components as well as the DUT, within an interface we can define signal directions, meaning that different port directions can be passed to different components. This allows us to define different input-output directions for each component if necessary.

4.2 Advantages of Using an Interface

Just like a testbench, an interface in SystemVerilog can encapsulate tasks, functions, parameters, variables, functional coverage, and assertions. This feature allows for a smooth and maintainable connection to the design, as all the relevant information and behaviors are centralized within the interface itself.

- Significantly reduces repetitive code by avoiding the need to declare individual signals one by one.
- Improves code legibility and organization.
- Interfaces can be reused across different modules and testbenches, promoting modularity.
- Facilitates signal maintenance and modification, since all related signals are centralized within the interface.
- Allows the definition of modports, which can adapt the interface to different module contexts (e.g., DUT, Testbench, Monitor).

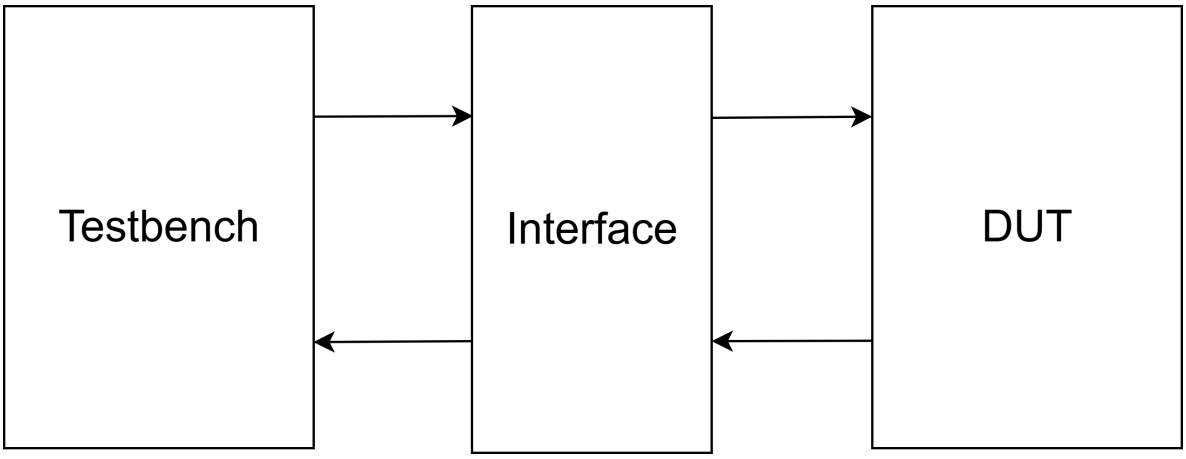

Figure 3: Interface Connection

4.3 How to Connect the Interface

```
interface interface_ip_tile(input logic clk, input logic arst_n);

        // CSR parameters
        parameter CSR_IN_WIDTH = 16;
        parameter CSR_OUT_WIDTH = 16;
        parameter REG_WIDTH = 32;

        // Declaration of signals used by user tile
  bit [CSR_IN_WIDTH - 1 : 0] csr_in;
  bit [REG_WIDTH - 1 : 0] data_reg_a;
  bit [REG_WIDTH - 1 : 0] data_reg_b;
  logic [CSR_OUT_WIDTH - 1 : 0] csr_out;
  logic csr_in_re;
  logic csr_out_we;
  logic [REG_WIDTH - 1 : 0] data_reg_c;

        // Declaration of signals used by testbench only (can only be accessed
            by interface tasks/functions)
  logic [CSR_OUT_WIDTH - 1 : 0] csr_out_r;
  bit csr_in_we;
        bit [CSR_IN_WIDTH - 1 : 0] csr_in_wdata;
  bit csr_out_re;
```

Listing 1: Interface definition

Within the interface, we centralize all the signals that will be used by the modules we intend to instantiate. This eliminates redundancy by avoiding the need to declare connections repeatedly across different testbenches or designs. Additionally, by defining parameters inside the interface, we enable global configuration changes from a single location, which simplifies maintenance and improves design scalability.

4.4 Interface Instantiation

Once the interface has been defined, it can be used like any port of a module. For the implementation on the FSM testbench, the coding style should be as follows:

```
module ip_tile_fsm_bitwise_shifter_tb;

        parameter CSR_IN_WIDTH = 16;
        parameter CSR_OUT_WIDTH = 16;
        parameter REG_WIDTH = 32;


        bit clk;
        bit arst_n;
    // Interface instance for DUT signal grouping and testbench control
        interface_ip_tile intf(clk, arst_n);
```

Listing 2: Interface Declaration and Instantiation

This interface will be used to implement some tasks and functions, which will drive our DUT's inputs to specific values and also monitor outputs.

4.5 List of Tasks and Functions

The list of tasks and functions implemented inside the interface is presented next.

Name in Testbench	Type	Description
write_data_reg_a()	task	Writes a 32-bit value into data_reg_a.
write_data_reg_b()	task	Writes a 32-bit value into data_reg_b.
read_data_reg_c()	function	Returns the result from data_reg_c.
write_csr_in()	task	Writes a 16-bit value into csr_in.
read_csr_out()	task	Reads the value of csr_out once csr_out_we is active.
*wait_done()	task	Waits for read_csr_out[0] (DONE flag) to fall to read csr_out_r calling read_csr_out() task.

Table 4: Description of interface tasks and functions to interact with the RTL module.

Doing this allows us to make multiple instances of the interface to test different aspects of the design, making the code simpler only needing to organize the instances of the interface and the tasks to be called.

NOTE: The `wait_done()` task is exclusive to the `bitwise_shifter` testbench. It was developed specifically to evaluate the `bitwise_shifter` design and is not included in the standard SemiColab testbench.

4.6 Accessing Interface Signals, Tasks and Functions

In order to access the tasks and functions defined within the interface, the user must specify the name of the interface instance before the task and function.

Inside the testbench, we must use the name of the interface instance to access the signals, tasks, and functions defined within it. Since these elements are encapsulated inside the interface, they are only accessible via the instance (e.g., `intf.write_data_reg_a(...)`).

The DUT itself only interacts with the signals that are wired through the interface, not with the tasks or functions. Therefore, using the interface instance in the testbench is essential for applying stimulus, checking results, or coordinating test sequences. An example illustrating this concept is shown below.

```
module ip_tile_fsm_shifter_tb;

        parameter CSR_IN_WIDTH = 16;
        parameter CSR_OUT_WIDTH = 16;
        parameter REG_WIDTH = 32;

        bit clk;
        bit arst_n;

        // Instantiation of the user-defined interface
    // This encapsulates all DUT signals and provides tasks/functions for
        testbench interaction
        interface_ip_tile intf(clk, arst_n);

        always #5ns clk = !clk;
        assign #20ns arst_n = 1'b1;

    // DUT instantiation
    // Each DUT port is connected to the corresponding signal within the
        interface instance
    // This allows the testbench to drive inputs and observe outputs via the
        interface
    ip_tile_fsm_shifter DUT(
      .clk(clk),
      .arst_n(arst_n),
      .csr_in(intf.csr_in),
      .data_reg_a(intf.data_reg_a),
```

```
25        .data_reg_b(intf.data_reg_b),
26        .csr_out(intf.csr_out),
27        .csr_out_we(intf.csr_out_we),
28        .data_reg_c(intf.data_reg_c),
29        .csr_in_re(intf.csr_in_re)
30      );
31
32    initial begin
33      clk = 0;
34      arst_n = 0;
35      @(posedge arst_n);  // wait for reset release
36
37      // Arbitrary Test 1 to show interface use: Shift data_reg_a (0xA5A5A5A5)
            right 32 bits
38      intf.write_data_reg_a(32'hA5A5A5A5);
39      intf.write_csr_in(16'b1000_0001_1111_0101);
40      intf.wait_done();
41
42      // Arbitrary Test 2 to show interface use: Shift data_reg_b (0xF00000000)
            left 4 bits
43      intf.write_data_reg_b(32'hF000_0000); // binary 1010
44      intf.write_csr_in(16'b1000_0000_0011_1010);
45      intf.wait_done();
46      $display("Test2 result (shift left 4 bits reg_b=0xA): 0x%08h", intf.
            read_data_reg_c());
```

Listing 3: Accessing Interface Tasks and Functions

5 Combinational RTL Example: 8-bit ALU with 16 Operations Using the Semicolab User Tile

5.1 Module Overview

As part of the integration examples for the SemiColab User Logic Tile, this section presents a combinational Arithmetic Logic Unit (ALU). The module is fully adapted to the user tile port conventions previously described.

The ALU receives its operands from the `data_reg_a` and `data_reg_b` input ports. Operand A is encoded in the lower 8 bits `[7:0]` of `data_reg_a`, while Operand B is extracted from the lower 8 bits `[7:0]` of `data_reg_b`. The operation selector is defined by the four most significant bits `[31:28]` of `data_reg_a`.

The result of the computation is provided through the output port `data_reg_c`, specifically in its least significant 8 bits `[7:0]`. Additionally, the ALU reports operational flags through the five most significant bits `[31:27]` of `data_reg_c`. These flags indicate conditions such as addition or multiplication overflow, division by zero, and zero result. Detailed descriptions of each flag are presented in the following subsections.

It is important to note that not all tile ports need to be utilized by every user module. However, compliance with the Tile Wrapper interface is mandatory to ensure interoperability and seamless system integration. This ALU example demonstrates how to implement a fully functional combinational logic design while respecting the signal conventions defined by the SemiColab Tile architecture.

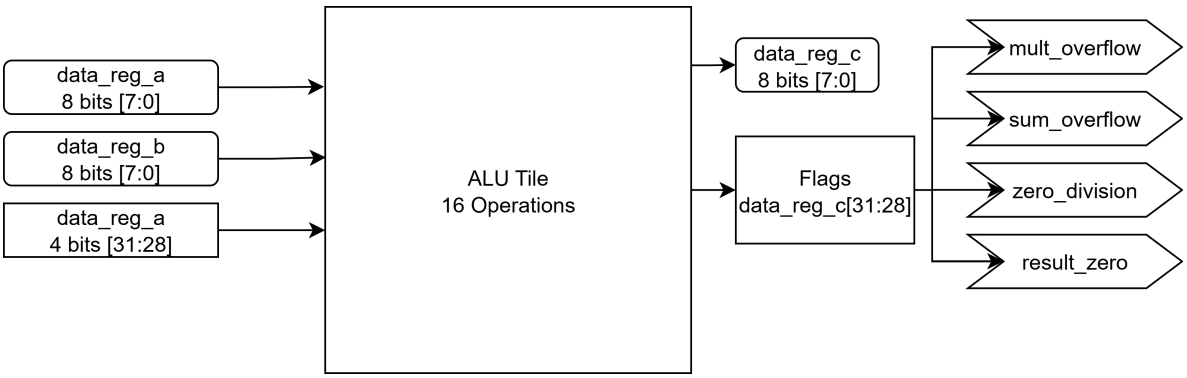

Figure 4: Combinational ALU Tile Block Diagram.

5.2 data_reg_a Bits Conventions for the ALU module

Bits	Type	Usage in this module
`[31:28]`	Input	Operation selector for the ALU
`[27:8]`	Input	Not in use in this module
`[7:0]`	Input	Input data for the ALU operand A

Table 5: Description of data_reg_a in the ALU module.

5.3 data_reg_b Bits Conventions for the ALU module

Bits	Type	Usage in this module
[31:8]	Input	Not in use in this module
[7:0]	Input	Input data for the ALU operand B

Table 6: Description of data_reg_b in the ALU module.

5.4 data_reg_c Bits Conventions for the ALU module

Bits	Type	Usage in this module
[31:28]	Output	Flags to provide information related to the ALU operation result
[7:0]	Output	Result from the ALU selected operation

Table 7: Description of data_reg_b in the ALU module.

5.5 RTL Implementation

```
1
2  `timescale 1ns / 1ps
3
4  module ip_tile_alu_8bit_16op #(
5  parameter REG_WIDTH = 32,
6  parameter CSR_IN_WIDTH = 16,
7  parameter CSR_OUT_WIDTH = 16)
8
9  (
10 input wire clk,
11 input wire arst_n,
12 input wire [CSR_IN_WIDTH-1:0] csr_in,
13 input wire  [REG_WIDTH - 1:0] data_reg_a,
14 input wire  [REG_WIDTH - 1:0] data_reg_b,
15 output wire [REG_WIDTH - 1:0] data_reg_c,
16 output wire [CSR_OUT_WIDTH-1:0] csr_out,
17 output wire csr_in_re,
18 output wire csr_out_we
19     );
20
21 // localparam for operation selector
22 localparam ADD  = 4'b0000,
23            SUB  = 4'b0001,
24            MUL  = 4'b0010,
25            DIV  = 4'b0011,
26            AND  = 4'b0100,
27            OR   = 4'b0101,
28            NOT  = 4'b0110,
29            XOR  = 4'b0111,
30            XNOR = 4'b1000,
31            LEFT_SHIFT  = 4'b1001,
32            RIGHT_SHIFT = 4'b1010,
33            INCREMENT  = 4'b1011,
34            DECREMENT  = 4'b1100,
35            INVERSE_SUB = 4'b1101,
36            AR_RIGHT_SHIFT_A = 4'b1110,
37            AR_RIGHT_SHIFT_B = 4'b1111;
38
```

```
// alu operation selector from a 4 MSBs
wire [3:0] alu_op = data_reg_a[31:28];
wire [7:0] operand_a = data_reg_a[7:0];
wire [7:0] operand_b = data_reg_b[7:0];
wire [3:0] flags;

// wires for internal flags and operations
wire [15:0] full_mult;
wire [15:0] full_sum;
wire zero_division;
wire mult_overflow;
wire sum_overflow;
wire result_zero;

// registers for internal output values
reg [7:0] data_reg_c_r;
reg csr_out_we_reg;
reg csr_in_re_reg;

// ALU //
always @(*) begin
    csr_in_re_reg   = 1'b0;
    csr_out_we_reg  = 1'b0;
    data_reg_c_r    = 8'd0;
    case(alu_op)
        ADD: data_reg_c_r  = operand_a + operand_b;
        SUB: data_reg_c_r  = operand_a - operand_b;
        MUL: data_reg_c_r  = operand_a * operand_b;
        DIV: begin
            if(operand_b == 0)
                data_reg_c_r = 8'b11111111;
            else
                data_reg_c_r = operand_a / operand_b;
        end
        AND: data_reg_c_r  = operand_a & operand_b;
        OR:  data_reg_c_r  = operand_a | operand_b;
        NOT: data_reg_c_r  = ~operand_a;
        XOR: data_reg_c_r  = operand_a ^ operand_b;
        XNOR: data_reg_c_r = ~(operand_a ^ operand_b) ;
        LEFT_SHIFT: data_reg_c_r = operand_a << 1;
        RIGHT_SHIFT: data_reg_c_r = operand_a >> 1;
        INCREMENT: data_reg_c_r = operand_a + 1;
        DECREMENT: data_reg_c_r = operand_a - 1;
        INVERSE_SUB: data_reg_c_r = (operand_b - operand_a);
        AR_RIGHT_SHIFT_A: data_reg_c_r = operand_a >>> 1 ;
        AR_RIGHT_SHIFT_B: data_reg_c_r = operand_b >>> 1 ;
        default: data_reg_c_r = 8'd0;
    endcase
end

// assign for internal signals used for flags
assign full_sum = operand_a + operand_b; // internal sum operation with 16
    bits to detect overflow
assign full_mult = operand_a * operand_b; // internal multiplication operation
     with 16 bits to detect overflow
assign sum_overflow = |(full_sum[15:8]) && (alu_op == ADD); // flag to
    indicate that the SUM has overflow
assign mult_overflow = |(full_mult[15:8]) && (alu_op == MUL); // flag to
    indicate that the MULTIPLICATION has overflow
assign zero_division = (operand_b == 0) && (alu_op == DIV); // flag to
    indicate that the DIVISION is invalid (division by 0)
```

```
95 assign result_zero = (data_reg_c_r == 0); // flag to indicate that the
       operation result is 0
96
97 // assign to the module outputs
98 assign csr_out_we = csr_out_we_reg;
99 assign csr_in_re = csr_in_re_reg;
100
101 // assign flag values and concatenate operation result with flags in
       data_reg_c output
102 assign flags = {mult_overflow, sum_overflow, zero_division, result_zero};
103 assign data_reg_c = {flags, 20'd0, data_reg_c_r};
104
105 endmodule
```

Listing 4: ALU RTL implementation

5.6 Testbench

The implemented testbench is designed to verify the behavior of the ip_tile_alu_8bit_16op module. Given that this ALU module is purely combinational and does not rely on control or synchronization signals such as csr_in, csr_out, or internal state machines, the testbench is kept simple and direct. The verification consists of assigning operands and operation selectors through data_reg_a and data_reg_b, and checking the resulting output and flags in data_reg_c.

5.7 Signal list

Name in Testbench	Description
clk	Clock signal, although not strictly required for a combinational block
arst_n	Asynchronous active-low reset signal, not strictly required for a combinational block
data_reg_a	Input data register carrying Operand A (bits [7:0]) and ALU operation selector (bits [31:28])
data_reg_b	Input data register carrying Operand B, (bits [7:0])
data_reg_c	Output result from ALU operation, stored in bits [7:0] and output status flags stored in bits [31:27]

Table 8: Description of testbench signals for interacting with the RTL module.

Although the module does not rely on a specific clock edge, a simple clock generator and reset initialization are included to maintain consistency with the SemiColab testbench environment.

5.8 DV plan

A total of 16 individual test cases are applied to the module, one for each supported ALU operation. These include basic arithmetic operations (ADD, SUB, MUL, DIV), logic operations (AND, OR, XOR, XNOR, NOT), and shift/bitwise utilities (SHIFT LEFT, SHIFT RIGHT, etc.). The testbench assigns operand and operation selection values directly to the inputs and evaluates the result of data_reg_c by inspecting both the result and the flags.

Each test case uses a specific operation selector encoded in the top 4 bits of data_reg_a[31:28], along with operands A in data_reg_a[7:0] and B in data_reg_b[7:0].

5.9 Initial Stimulus Block

The following initial block contains the complete sequence of test applied to the ALU.

```
1
2  initial begin
3      clk = 0;
4      arst_n = 0;
5      #10;
6      arst_n = 1;
7
8      // ---------- ADD ----------
9      intf.write_data_reg_a({4'b0000, 20'd0, 8'd10}); // opcode ADD, A = 10
10     intf.write_data_reg_b({24'd0, 8'd5});           // B = 5
11     #10;
12
13     // ---------- SUB ----------
14     intf.write_data_reg_a({4'b0001, 20'd0, 8'd20}); // opcode SUB, A = 20
15     intf.write_data_reg_b({24'd0, 8'd8});           // B = 8
16     #10;
17
18     // ---------- MUL ----------
19     intf.write_data_reg_a({4'b0010, 20'd0, 8'd7});  // opcode MUL, A = 7
20     intf.write_data_reg_b({24'd0, 8'd6});           // B = 6
21     #10;
22
23     // ---------- DIV ----------
24     intf.write_data_reg_a({4'b0011, 20'd0, 8'd40}); // opcode DIV, A = 40
25     intf.write_data_reg_b({24'd0, 8'd5});           // B = 5
26     #10;
27
28     // ---------- DIV /0 ----------
29     intf.write_data_reg_a({4'b0011, 20'd0, 8'd15}); // opcode DIV, A = 15
30     intf.write_data_reg_b({24'd0, 8'd0});           // B = 0
31     #10;
32
33     // ---------- AND ----------
34     intf.write_data_reg_a({4'b0100, 20'd0, 8'b10101010});
35     intf.write_data_reg_b({24'd0, 8'b11001100});
36     #10;
37
38     // ---------- OR ----------
39     intf.write_data_reg_a({4'b0101, 20'd0, 8'b10100010});
40     intf.write_data_reg_b({24'd0, 8'b00001111});
41     #10;
42
43     // ---------- NOT ----------
44     intf.write_data_reg_a({4'b0110, 20'd0, 8'b11110000}); // Solo usa A
45     intf.write_data_reg_b(32'd0);
46     #10;
```

```
47
48      // ---------- XOR ----------
49      intf.write_data_reg_a({4'b0111, 20'd0, 8'b11110000});
50      intf.write_data_reg_b({24'd0, 8'b00001111});
51      #10;
52
53      // ---------- XNOR ----------
54      intf.write_data_reg_a({4'b1000, 20'd0, 8'b10101010});
55      intf.write_data_reg_b({24'd0, 8'b10101010});
56      #10;
57
58      // ---------- SHIFT LEFT ----------
59      intf.write_data_reg_a({4'b1001, 20'd0, 8'b00011111});
60      intf.write_data_reg_b(32'd0);
61      #10;
62
63      // ---------- SHIFT RIGHT ----------
64      intf.write_data_reg_a({4'b1010, 20'd0, 8'b11100000});
65      intf.write_data_reg_b(32'd0);
66      #10;
67
68      // ---------- INCREMENT ----------
69      intf.write_data_reg_a({4'b1011, 20'd0, 8'd99});
70      intf.write_data_reg_b(32'd0);
71      #10;
72
73      // ---------- DECREMENT ----------
74      intf.write_data_reg_a({4'b1100, 20'd0, 8'd100});
75      intf.write_data_reg_b(32'd0);
76      #10;
77
78      // ---------- INVERSE SUB ----------
79      intf.write_data_reg_a({4'b1101, 20'd0, 8'd30});
80      intf.write_data_reg_b({24'd0, 8'd50});
81      #10;
82
83      // ---------- ARITH SHIFT RIGHT A ----------
84      intf.write_data_reg_a({4'b1110, 20'd0, 8'b11110000});
85      intf.write_data_reg_b(32'd0);
86      #10;
87
88      // ---------- ARITH SHIFT RIGHT B ----------
89      intf.write_data_reg_a({4'b1111, 20'd0, 8'b00000000});
90      intf.write_data_reg_b({24'd0, 8'b11110000});
91      #10;
92
93      $finish;
94  end
```

Listing 5: ALU Testbench Initial Stimulus Block

5.10 Simulation Results

To verify the correct behavior of the ALU module, a simulation testbench was created using Vivado. The following waveform captures illustrate the expected behavior for some of the implemented operations.

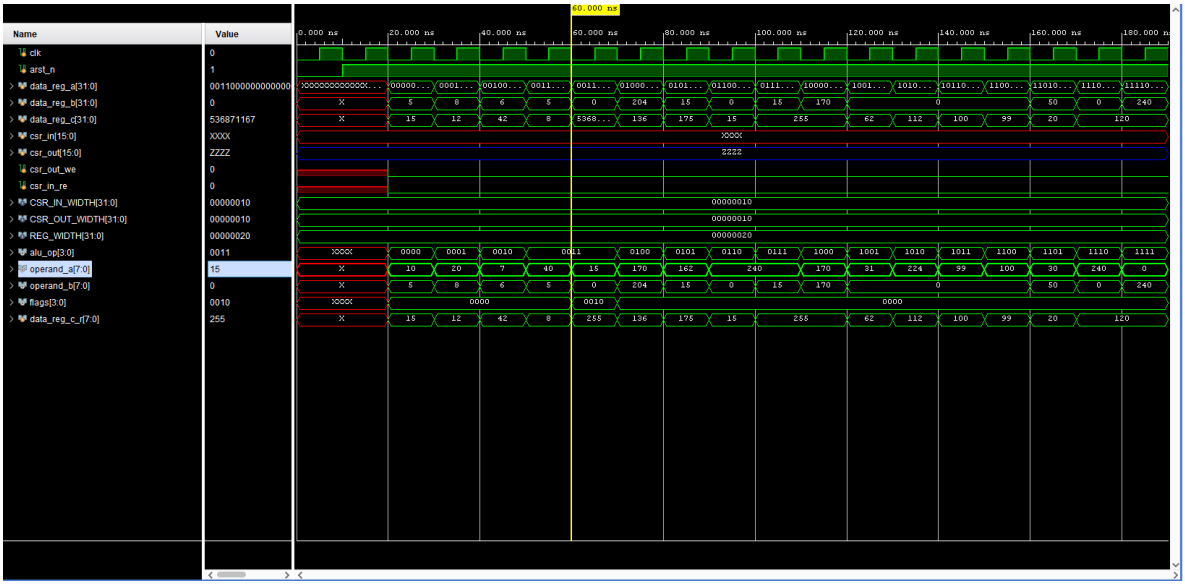

Figure 5: Waveform showing the complete stimulus for the ALU.

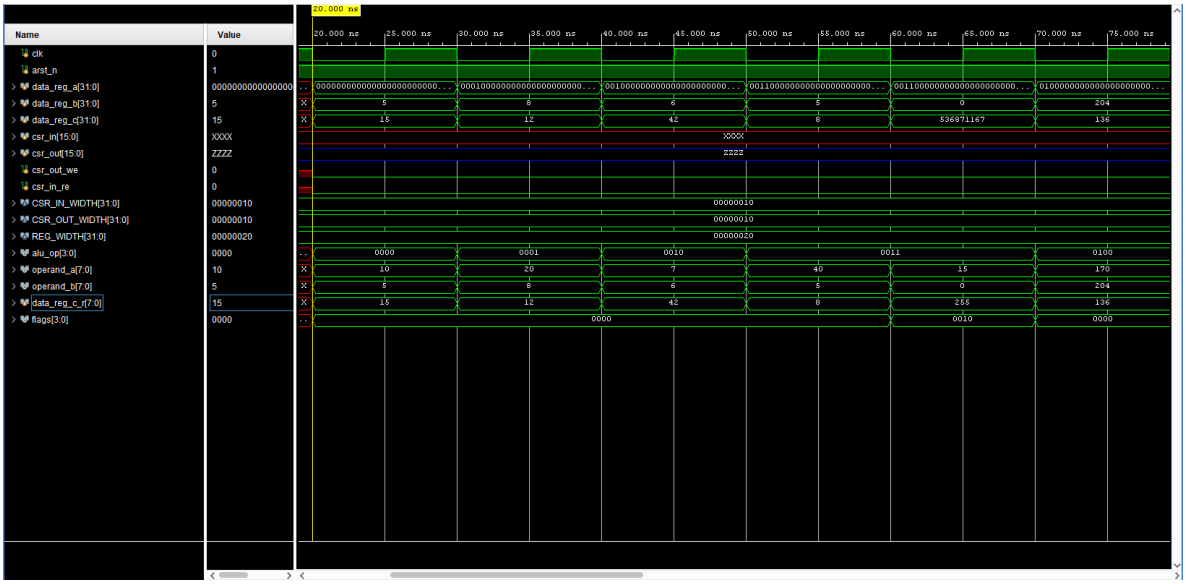

Figure 6: Waveform showing the result from the first 4 operations in the ALU.

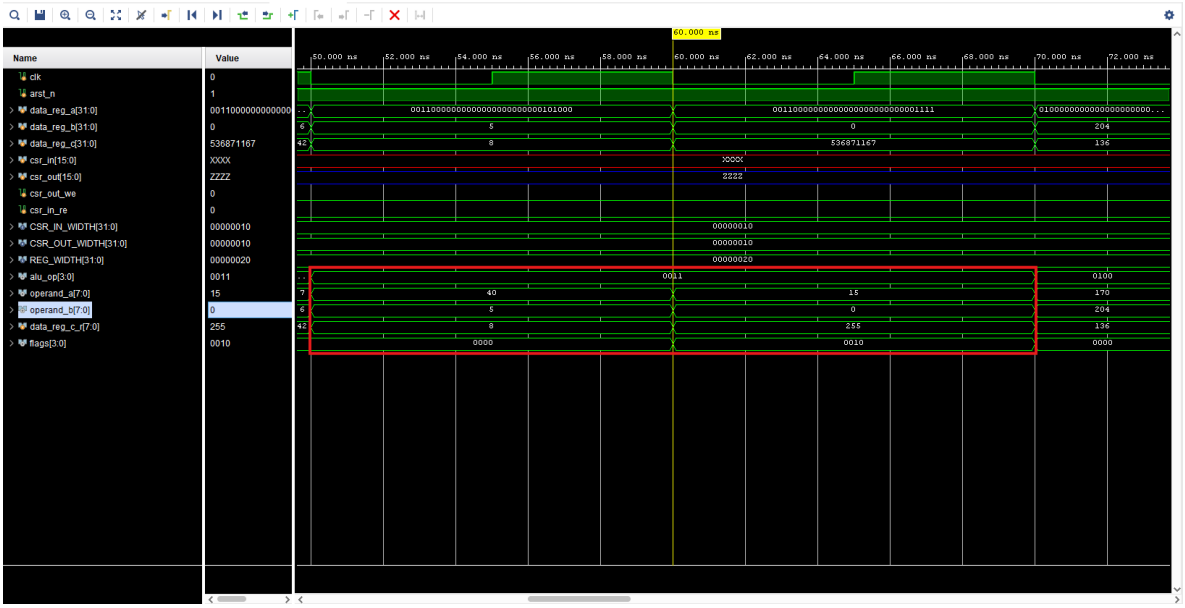

Figure 7: Waveform showing the result from the division, indicating a division by zero error in the ALU flags output.

NOTE: The `csr_in` is always XXXX (undefined) due to the fact that it is declared as an input port but is never driven by any source—i.e., it is not connected to anything in the top-level module, so it will not have a defined value. Something similar happens with `csr_out` it is also declared as a port, an output this time, and is not connected to any destination either. As a result, its value remains ZZZZ (high impedance).

6 Sequential RTL Example: FSM Bitwise Shifter Using the Semicolab User Tile

The RTL implementation for this arbitrary tile consists of a simple four-state Finite State Machine (FSM) designed to perform a configurable bitwise shifting operation. Based on user-defined control signals received through `csr_in`, the FSM selects an input source (`data_reg_a` or `data_reg_b`), determines the shift direction (left or right), and applies a serial shift for a specified number of cycles (defined by the number of shifts set in `csr_in`). Once the operation is complete, the result is presented on `data_reg_c`, and the module signals completion through `csr_out`.

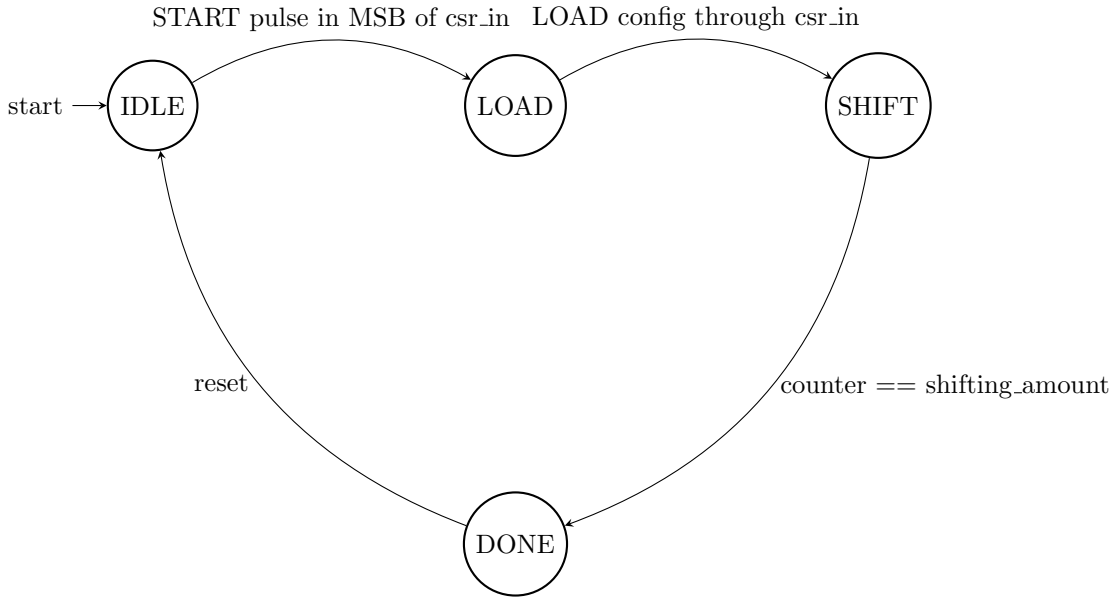

Figure 8: FSM used in the RTL bitwise shifter module.

6.1 csr_in Bits Conventions for bitwise_shifter module

Bits	Type	Usage in this module
[15]	Single pulse	Start pulse in MSB of csr_in
[8:4]	Stables	Shifting amount in csr_in stable bits
[3:0]	Clear on read	[3] Shift left, [2] Shift right, [1] Select data_reg_a, [0] Select data_reg_b

Table 9: Description of the csr_in bits used in the fsm_bitwise_shifter module.

6.2 csr_out Bits Conventions for bitwise_shifter module

Bits	Type	Usage in this module
[15]	Stable	Busy signal: indicates shifting operation is on going
[0]	Clear on read	Done signal: indicates shifting operation is done

Table 10: Description of the csr_out bits used in this RTL module.

6.3 bitwise_shifter FSM States

State	Description
IDLE	IDLE state waiting for a start pulse
LOAD	Load CSR_IN configuration and input data
SHIFT	Performs the bitwise shifting operation
DONE	Indicates that the operation has been finished

Table 11: Implemented states for the FSM in the bitwise_shifter module.

The user writes control data into the csr_in register to initiate the shifting process. The control signals include:

- A start pulse to trigger the operation
- The shift direction (left or right)
- The shifting amount
- The input data source selection (data_reg_a or data_reg_b)

Once the `csr_in` has been successfully read by the module, the FSM begins the bitwise shifting operation, processing one bit per clock cycle. Finally, the module activates `csr_out_we` and sets the `done` flag to high in the `csr_out` register to indicate that the operation has finished. The final result is then output through the `data_reg_c` port of the wrapper.

6.4 RTL Implementation

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Mifral
// Engineer:
//
// Design Name: semicolab
// Module Name: ip_tile_fsm_bitwise_shifter
//
//////////////////////////////////////////////////////////////////////////////////

// ===================== INPUT REGISTERS ======================

// DATA_REG_A - 32-bit data input port.
// This register provides the primary input data to be shifted by the FSM
// when selected through CSR_IN.

// DATA_REG_B - 32-bit alternative data input port.
// This serves as a second data source. The FSM will use this register if selected
// through CSR_IN.

// DATA_REG_C - 32-bit output port.
// This is the final output of the FSM once the shifting operation has completed.
// The value is valid when the DONE flag is HIGH (csr_out[0] = 1).

// ===================== CSR_IN BREAKDOWN (16 bits) ======================

// CSR_IN CLEAR-ON-READ[1:0] - Data input selection.
// [0] = 1 to select DATA_REG_A as the input to be shifted.
```

```
// [1] = 1 to select DATA_REG_B as the input to be shifted.
// If both are zero, the module will use zero as the input (null data).
// These bits are "clear-on-read", meaning they are automatically cleared
// by the FSM once read.

// CSR_IN PULSE[15] - Operation start pulse.
// Writing '1' to this bit signals the FSM to begin a shifting operation.
// This is a pulse bit, and must only be high for one cycle.

// CSR_IN CLEAR-ON-READ[3:2] - Shift direction control.
// [2] = 1 selects a right shift operation.
// [3] = 1 selects a left shift operation.
// These are clear-on-read bits and must be re-written for each new operation.
// If both bits are zero, no shifting occurs. If both are one, left has
    priority.

// CSR_IN STABLES[8:4] - Shift amount.
// Using 5 stable bits from csr_in indicating how many bits to shift.
// Range: 0 to 31 (for 32-bit registers).
// This value remains valid until overwritten.

// ===================== CSR_OUT BREAKDOWN (16 bits) ======================

// CSR_OUT[15] - BUSY flag (stable).
// Set to '1' while the FSM is performing the shifting operation.
// Cleared automatically when the FSM reaches the DONE state.

// CSR_OUT[0] - DONE flag (clear-on-read).
// Set to '1' for one clock cycle when the operation has completed.
// This is a "clear-on-read" flag: it is automatically cleared by reading
// from csr_out.



module ip_tile_fsm_bitwise_shifter #(
parameter REG_WIDTH = 32,
parameter CSR_IN_WIDTH = 16,
parameter CSR_OUT_WIDTH = 16)

(
input wire clk,
input wire arst_n,
input wire [CSR_IN_WIDTH-1:0] csr_in,
input wire  [REG_WIDTH - 1:0] data_reg_a,
input wire  [REG_WIDTH - 1:0] data_reg_b,
output wire [REG_WIDTH - 1:0] data_reg_c,
output wire [CSR_OUT_WIDTH-1:0] csr_out,
output wire csr_in_re,
output wire csr_out_we
    );


`define CSR_OUT_BUSY csr_out[15] // DEFINE BUSY SIGNAL ON CSR_IN STABLE BIT TO
    INDICATE THAT THE FSM IS SHIFTING THE DATA
`define CSR_OUT_DONE csr_out[0] // DEFINE DONE SIGNAL ON CSR_IN CLEAR ON READ
    BIT TO INDICATE THAT THE FSM HAS FINISHED THE SHIFTING OPERATION
`define CSR_IN_START csr_in[15] // START PULSE SIGNAL TO LOAD DATA AND START
    SHIFTING OPERATION
`define CSR_IN_SHIFT_AMOUNT csr_in[8:4] // DEFINE SHIFT AMOUNT IN CSR_IN
    STABLE BITS (MAX SHIFT AMOUNT = 32 BITS)
`define CSR_IN_RIGHT_SHIFT_DIRECTION csr_in[2] // DEFINE RIGHT SHIFT DIRECTION
    IN CSR_IN CLEAR ON READ BIT
```

```
85 `define CSR_IN_LEFT_SHIFT_DIRECTION csr_in[3] // DEFINE LEFT SHIFT DIRECTION
       IN CSR_IN CLEAR ON READ BIT
86 `define CSR_IN_INPUT_DATA_REG_A_SELECTION csr_in[0] // DEFINE DATA_REG_A
       SELECTION IN CSR_IN CLEAR ON READ BIT
87 `define CSR_IN_INPUT_DATA_REG_B_SELECTION csr_in[1] // DEFINE DATA_REG_B
       SELECTION IN CSR_IN CLEAR ON READ BIT
88
89 reg [1:0] state_reg, state_nxt; // STATES
90 reg [4:0] shift_counter; // COUNTER TO SHIFT THE DATA THE AMOUNT DEFINED BY
       THE CSR_INPUT
91 reg [REG_WIDTH - 1:0] data_input_r; // REGISTER FOR INPUT DATA_REG_A/B
       SELECTED
92 reg [REG_WIDTH - 1:0] shifting_result_r; // REGISTER FOR SHIFTING OPERATION
       OUTPUT
93 reg csr_in_re_r; // REGISTER FOR CSR_IN_RE
94 reg csr_out_we_r; // REGISTER FOR CSR_OUT_WE
95 wire busy; // INDICATES THAT THE FSM IS PERFORMING THE SHIFTING OPERATION
96 wire done; // INDICATES THAT THE FSM HAS FINISHED THE SHIFTING OPERATION
97
98 reg [4:0] shift_amount_r; // REGISTER TO STORE THE SHIFT AMOUNT SET IN CSR_IN
99 reg use_reg_a, use_reg_b; // REGISTER TO STORE THE DATA INPUT SELECTED IN
       CSR_IN
100 reg shift_left_r, shift_right_r; // REGISTER TO STORE THE SHIFTING DIRECTION
       SELECTED IN CSR_IN
101
102 localparam IDLE  = 2'b00,
103            LOAD  = 2'b01,
104            SHIFT = 2'b10,
105            DONE  = 2'b11;
106
107 always @(*) begin
108     case (state_reg)
109         IDLE:  state_nxt = `CSR_IN_START ? LOAD : IDLE; // IF THERE IS A START
                    PULSE IN CSR_IN PULSE BIT, WE TRANSITION TO LOAD STATE
110         LOAD:  state_nxt = SHIFT; // ONCE THE DATA IS LOADED WE TRANSITION TO
                   SHIFT STATE
111         SHIFT: state_nxt = (shift_counter == shift_amount_r) ? DONE : SHIFT;
                   // WE SHIFT THE DATA UNTIL THE SHIFT AMOUNT HAS BEEN REACHED
112         DONE:  state_nxt = IDLE; // DONE STATE INDICATED BY CSR_OUT PULSE BIT
                   - DONE SIGNAL
113         default: state_nxt = IDLE;
114     endcase
115 end
116
117
118 always @(posedge clk or negedge arst_n) begin
119     if (!arst_n) begin
120         state_reg       <= IDLE;
121         shift_counter   <= 5'd0;
122         data_input_r        <= {REG_WIDTH{1'b0}};
123         shifting_result_r <= {REG_WIDTH{1'b0}};
124         csr_in_re_r <= 1'b0;
125         csr_out_we_r <= 1'b0;
126         shift_amount_r <= 5'd0;
127         use_reg_a <= 1'b0;
128         use_reg_b <= 1'b0;
129         shift_left_r <= 1'b0;
130         shift_right_r <= 1'b0;
131     end else begin
132         state_reg <= state_nxt;
133         csr_out_we_r <= 1'b0;
134         csr_in_re_r <= 1'b0;
```

```
135
136         case (state_reg)
137             IDLE: begin
138                 if(`CSR_IN_START) begin // TRIGGER THE OPERATION IF A START
                        PULSE IS DETECTED IN CSR_IN PULSE BIT
139                     // SAVE ALL THE CSR_IN SIGNALS IN REGISTERS TO PERFORM THE
                            OPERATION //
140                     shift_amount_r <= (`CSR_IN_SHIFT_AMOUNT > REG_WIDTH) ?
                            REG_WIDTH[4:0] : `CSR_IN_SHIFT_AMOUNT;
141                     use_reg_a <= `CSR_IN_INPUT_DATA_REG_A_SELECTION;
142                     use_reg_b <= `CSR_IN_INPUT_DATA_REG_B_SELECTION;
143                     shift_left_r <= `CSR_IN_LEFT_SHIFT_DIRECTION;
144                     shift_right_r <= `CSR_IN_RIGHT_SHIFT_DIRECTION;
145                     csr_in_re_r <= 1'b1; // SET CSR_IN_RE TO 1 IN ORDER TO
                            CONFIRM THAT WE HAVE RECEIVED THE CSR_IN COMMAND
146                     shift_counter <= 5'd0; // RESET THE COUNTER
147                     shifting_result_r <= {REG_WIDTH{1'b0}};  // WE CLEAN THE
                            OUTPUT
148                 end
149             end
150             LOAD: begin
151                 if (use_reg_a) begin
152                     data_input_r <= data_reg_a; // USE DATA_REG_A AS OUR DATA
                            INPUT
153                 end else if (use_reg_b) begin
154                     data_input_r <= data_reg_b; // USE DATA_REG_B AS OUR DATA
                            INPUT
155                 end else
156                     data_input_r <= {REG_WIDTH{1'b0}};  // ALL 0'S IF INPUT
                            DATA IS NOT SELECTED
157             end
158             SHIFT: begin
159                 csr_out_we_r <= 1'b1; // ACTIVATING THE CSR_OUT_WE TO WRITE
                        THE BUSY SIGNAL IN CSR_OUT STABLE BIT
160                 shift_counter <= shift_counter + 1; // ADDING 1 TO THE SHIFT
                        COUNTER
161
162                 if (shift_left_r) begin
163                     shifting_result_r <= {shifting_result_r[REG_WIDTH-2:0],
                            data_input_r[REG_WIDTH-1]};  //     MSB FROM INPUT TO
                            LSB OF OUTPUT
164                     data_input_r        <= {data_input_r[REG_WIDTH-2:0], 1'b0
                            };  //     SHIFT INPUT LEFT
165                 end else if (shift_right_r) begin
166                     shifting_result_r <= {data_input_r[0], shifting_result_r[
                            REG_WIDTH-1:1]};  //     LSB FROM INPUT TO MSB OF
                            OUTPUT
167                     data_input_r        <= {1'b0, data_input_r[REG_WIDTH
                            -1:1]};  //     SHIFT INPUT RIGHT
168                 end
169             end
170             DONE: begin
171                 csr_out_we_r <= 1'b0; // ENABLE WRITING TO CSR_OUT TO INDICATE
                        THE DONE FLAG STATUS
172             end
173             default: begin
174             end
175         endcase
176     end
177 end
178
179
```

```
180
181
182 assign busy = (state_reg == SHIFT); // ASSIGN BUSY SIGNAL TO HIGH IF THE FSM
        IS SHIFTING, CURRENT STATE = BUSY
183 assign done = (state_reg == DONE);  // ASSIGN DONE SIGNAL TO HIGH IF THE FSM
        HAS FINISHED THE OPERATION, CURRENT STATE = DONE
184 assign csr_out_we = csr_out_we_r; // ASSIGN THE CSR_OUT_WE_INTERNAL SIGNAL
        USED IN THE LOGIC TO CSR_OUT_WE OUTPUT
185 assign csr_out = {busy, 14'd0, done}; // CONCATENATE BUSY SIGNAL IN THE STABLE
         BIT AND DONE SIGNAL IN THE PULSE BIT OF CSR_OUT
186 assign data_reg_c = shifting_result_r; // ASSIGN THE RESULT FROM THE SHIFTING
        OPERATION TO DATA_REG_C OUTPUT
187 assign csr_in_re = csr_in_re_r; // ASSIGN THE CSR_IN_RE_INTERNAL SIGNAL USED
        IN THE LOGIC TO CSR_IN_RE OUTPUT
188
189 endmodule
```

Listing 6: FSM Bitwise Shifter RTL Module

6.5 Testbench

The implemented testbench is designed to verify the behavior of the `ip_tile_fsm_bitwise_shifter` module. It generates the clock (`clk`) and asynchronous reset (`arst_n`) signals, and handles the operating and control signals. The CSR signals (including enable) are implemented in a way that execute the described behavior for `clear_on_read`, `single pulse`, and `stable` types, for both input and output control. This behavior is achieved through two `always_ff` blocks.

The testbench assigns values to the `data_reg_a` and `data_reg_b` inputs, as well as to `csr_in`, using individual `tasks`. To read `csr_out`, a `task` is used, and to read `data_reg_c` (the shifted result), a `function` is implemented.

As part of the SemiColab framework, this testbench is freely available for any tile designer to use when verifying their own designs, this tb can be found in the introduction section through a link to a github repository, right before Figure 1. Overall, the template includes the standard tile ports, CSR logic, and the necessary tasks to drive and evaluate the tile's behavior.

6.6 Signal list

Name in Testbench	Description
`clk`	Clock signal for synchronization
`arst_n`	Asynchronous active-low reset signal
`csr_in`	Control/status register input bus, 16 bits wide
`data_reg_a`	Input data register A for shift operations, 32 bits wide
`data_reg_b`	Input data register B for shift operations, 32 bits wide
`csr_in_we`	Write enable signal for `csr_in` (indicates write operation)
`csr_in_wdata`	Data to be written into `csr_in` on write enable, 16 bits wide
`csr_in_re`	Read enable signal for `csr_in` (indicates read operation from module)
`csr_out`	Control/status register output bus (from module), 16 bits wide
`csr_out_r`	Registered version of `csr_out` with read-clear behavior, 16 bits wide
`csr_out_re`	Read enable signal for `csr_out` (indicates reading output CSR)
`csr_out_we`	Write enable for `csr_out` register (controls updating `csr_out_r` from module)
`data_reg_c`	Output data register holding the result of the shift operation, 32 bits wide

Table 12: Description of testbench signals for interacting with the RTL module.

These signals will be performing the connection with the module to perform the driving inputs of operands and csr_in and also take the values of the output result of the shifting and the csr_out to perform the explained logic implemented in semicolab.

To perform the testing we must instance the module, in this case, by the name of DUT (Device Under Test).

```
1         ip_tile_fsm_bitwise_shifter DUT(
2           .clk(clk),
3           .arst_n(arst_n),
4           .csr_in(csr_in),
```

```
5           .csr_in_re(csr_in_re),
6           .data_reg_a(data_reg_a),
7           .data_reg_b(data_reg_b),
8           .csr_out(csr_out),
9           .csr_out_we(csr_out_we),
10          .data_reg_c(data_reg_c)
```

Listing 7: CSR implementation

6.7 Control Status Register Logic

As explained before, this testbench replicates the Control Status Register logic implemented in semicolab, performing the clear on read behaviour in the previously presented bits, as well as the single pulse inputs. This is achieved by an individual always_ff block for the csr_in and another for the csr_out. This csr logic will be connected to the shifter module, this will demonstrate how csr signals dictate the instructions to operate the inputs and evaluate the state of the module.

```
1  always_ff@(posedge clk, negedge arst_n) begin // CSR_OUT BLOCK
2    if(!arst_n) begin
3      csr_out_r <= {CSR_OUT_WIDTH{1'b0}};
4        end else begin
5          if(csr_out_we) // Writing is taking priority over clear on read
6            csr_out_r <= csr_out;
7          else if(csr_out_re)
8            csr_out_r[3 : 0] <= 4'd0; // 4 Least Significant Bits are clear on
                 read, so we have to clear when reading
9        end
10   end
11
12 always@(posedge clk, negedge arst_n) begin // CSR_IN BLOCK
13   if(~arst_n) begin
14     csr_in <= {CSR_IN_WIDTH{1'b0}};
15   end else begin
16     csr_in[CSR_IN_WIDTH - 1 : CSR_IN_WIDTH - 4] <= 4'd0; // 4 Most
         Significant Bits are single pulse, so we have to clear every clock
         cycle except when we write
17     if(csr_in_we) // Writing is taking priority over clear on read
18       csr_in <= csr_in_wdata;
19     else if(csr_in_re)
20       csr_in[3 : 0] <= 4'd0; // 4 Least Significant Bits are clear on read,
           so we have to clear when reading
21   end
22 end
```

Listing 8: CSR implementation

6.8 DV plan

To test the behaviour of the module interacting with the semicolab logic a total of eight tests were implemented. Proposed tests drive different values for the inputs `data_reg_a` & `data_reg_b` and the control signal `csr_in` and evaluate the output `data_reg_c` & control signal `csr_out`. Simulation will be done in Xilinx's Vivado and will be displayed (through waveform) and later displayed. List of implemented tests is shown next.

Test Name	Description
Test 1	Shift right data_reg_a containing 0xA5A5A5A5 by 32 bits, data_reg_c should output same value
Test 2	Shift left data_reg_b containing 0xF0000000 by 4 bits, data_reg_c should output 'hF
Test 3	Shift left data_reg_a containing 0xF0F0F0F0 by 16 bits, data_reg_c should output 'hF0F0
Test 4	Shift right data_reg_b containing 0xFFFFFFFF by 32 bits. All bits shifted out.
Test 5	Shift right data_reg_b containing 0xBBBBBBBB by 1 bit.
Test 6	Shift 0 bits on data_reg_a with 0xAAAAAAAA, data_reg_a should output Zero.
Test 7	Initiate a shift on data_reg_a with 0x12345678, then attempt to write csr_in again while FSM is busy. Validates that new commands are ignored until operation completes.
Test 8	Begin shift on data_reg_a with 0xCAFEBABE, then assert asynchronous reset during the operation. Validates reset behavior during active FSM state.

Table 13: Description of functional tests performed in the testbench.

These tests are directed by a series of tasks and functions implemented inside an interface created for this testbench.

6.8.1 Test 1

```
1     // Test 1: Shift data_reg_a (0xA5A5A5A5) right 32 bits
2     intf.write_data_reg_a(32'hA5A5A5A5);
3     intf.write_csr_in(16'b1000_0001_1111_0101);
4     intf.wait_done();
```

Listing 9: Test 1 implementation

Only first test will be used to demonstrate de csr signals, since the behaviour is constant throughout all tests, however, waveform will be used to show the process of the rtl.

1. Data for data_reg_a and write_csr_in are written at the same time unit, by looking at the task definition shown before.
2. The csr_in_wdata value and csr_in_we will be set (the we signal being only high for one clock cycle).
3. The csr_in_wdata value will be set to the actual csr_in input (csr_in[15:11] will be set for only one clock cycle since they are meant to be single pulse signals), propagating the csr_in_wdata value to csr_in.
4. If csr_in[15] is set, then the START signal is triggered, and in the next clock cycle the FSM transitions from IDLE to LOAD, in the same pulse csr_in[15:11] are cleared, and csr_in_re will be set.
5. In the next clock cycle after csr_in_re will be set, csr_in[0:3] will be cleared, the busy signal on csr_out (15th bit) will be set, and the FSM state will change to SHIFT to compute the shifting.
6. In the next clock cycle the csr_out_we signal will be asserted from the first shift until the DONE state is reached.
7. This behaviour is shown in Figure 2.

Figure 9: test 1 csr_in waveform.

1. The next line in `test1` calls the `wait_done()` task. This task waits for the computation to be finished by monitoring `csr_out[0]`, remembering that when that bit is highj it indicates DONE.
2. Once the operation is complete and `csr_out_re` is set to zero state is now IDLE (385 ns.), the done flag keeps set to the `csr_out_r` register.
3. Then, `csr_out_re` is ativated for one clock cycle, causing `csr_out_r[3:0]` to clear, since they are clear on read.
4. As a result, `csr_out[3:0]` is cleared, completing the process.

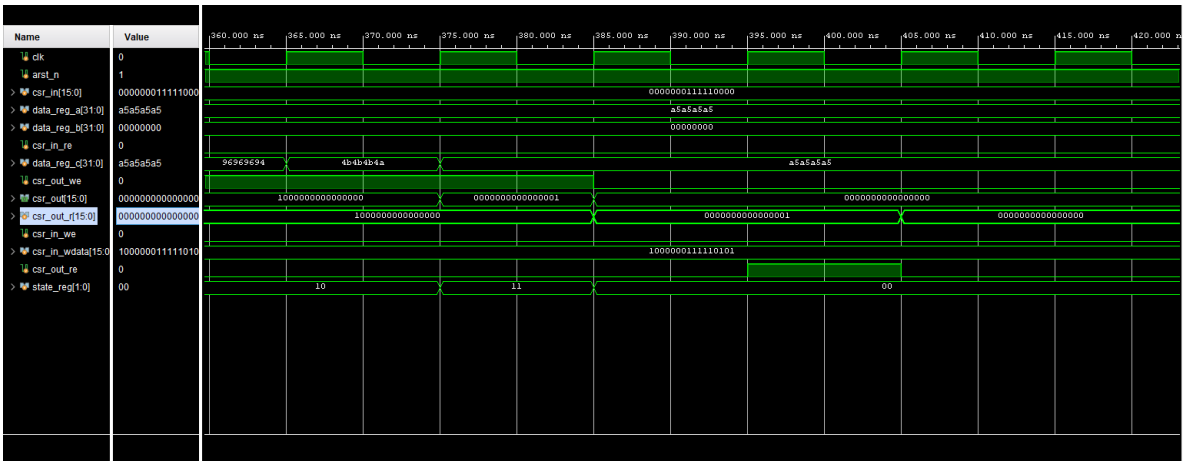

Figure 10: test 1 csr_out waveform.

6.8.2 Test 2

```
1     // Test 2: Shift data_reg_b (0xF0000000) left 4 bits
2     intf.write_data_reg_b(32'hF000_0000);
3     intf.write_csr_in(16'b1000_0000_0011_1010);
4     intf.wait_done();
```

Listing 10: Test 2 implementation

Control signals will work the same way, the computing of the calculations should be faster, since it is shifting less bits.

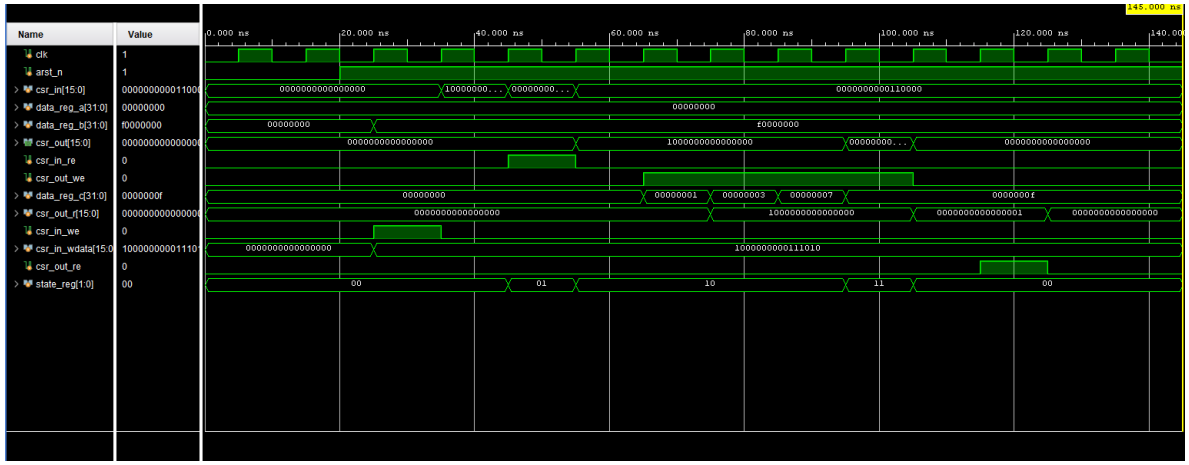

Figure 11: test 2 waveform.

6.8.3 Test 3

```
    // Test 1 & 2

    // Test 3: Shift data_reg_a (0xF0F0F0F0) left 16 bits
    intf.write_data_reg_a(32'hF0F0F0F0);
    intf.write_csr_in(16'b1000_0000_1111_1001);
    intf.wait_done();
```

Listing 11: Test 3 implementation

Since this test is similar to previous this section will be used to demonstrate how the module and testbench act correctly if a test is driven one after another.

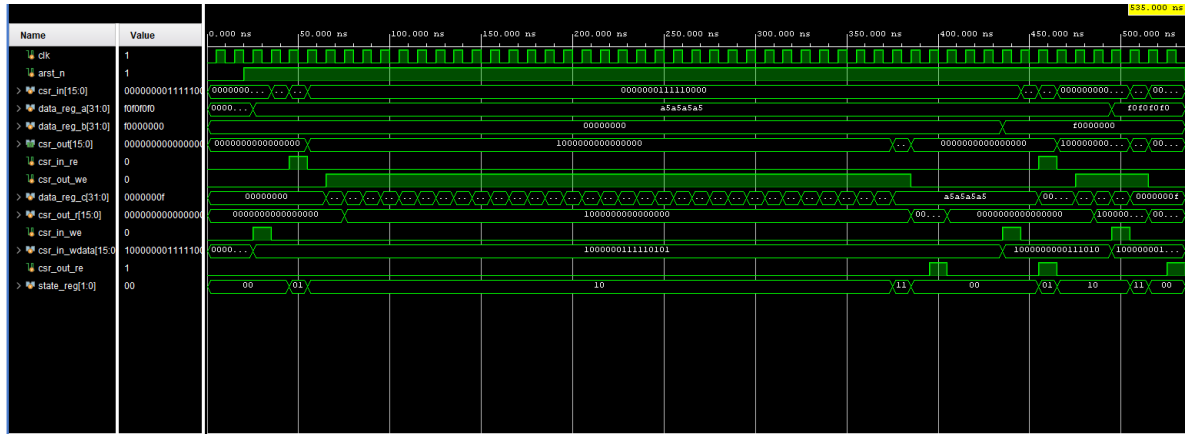

Figure 12: test 3 waveform.

6.8.4 Test 4

```
    // Test 4: Shift data_reg_b (0xFFFFFFFF) right 32 bits
    intf.write_data_reg_b(32'hFFFF_FFFF);
    intf.write_csr_in(16'b1000_0001_1111_0110);
    intf.wait_done();
```

Listing 12: Test 4 implementation

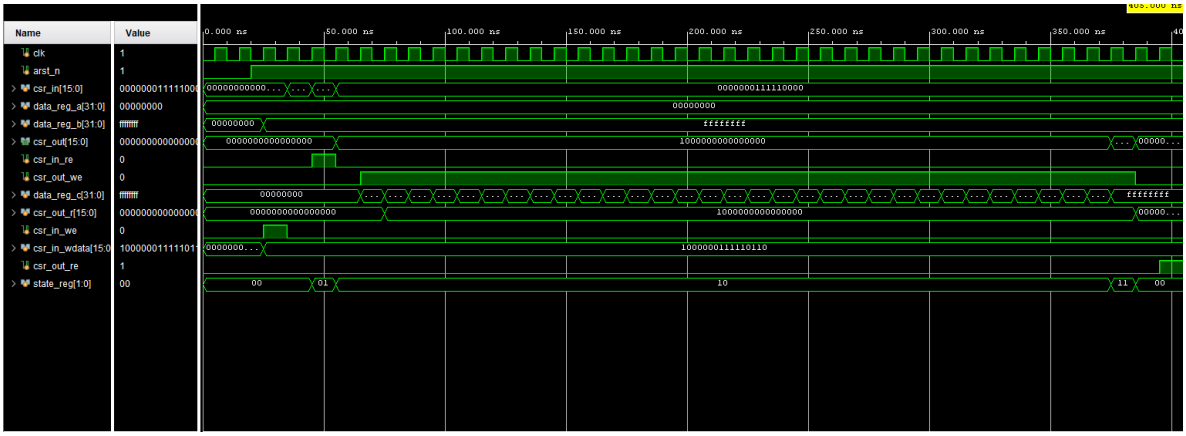

Figure 13: test 4 waveform.

6.8.5 Test 5 & 6

```
    // Test 5: Shift data_reg_b (0xBBBB_BBBB) right 1 bit
    intf.write_data_reg_b(32'hBBBB_BBBB);
    intf.write_csr_in(16'b1000_0000_0000_0110);
    intf.wait_done();

    // Test 6: Shift data_reg_a (0xAAAA_AAAA) 0 bits
    intf.write_data_reg_a(32'hAAAA_AAAA);
    intf.write_data_reg_b(32'h0000_0000);
    intf.write_csr_in(16'b1000_0000_0000_0001);
    intf.wait_done();
```

Listing 13: Test 5 & 6 implementation

Test 5 & 6 are specific and short cases so the are evaluated at the same time, when shifting number bits are 00000 then a 1 bit shift should occur, and when neither of the sifting direction bits are set then no shifting should occur.

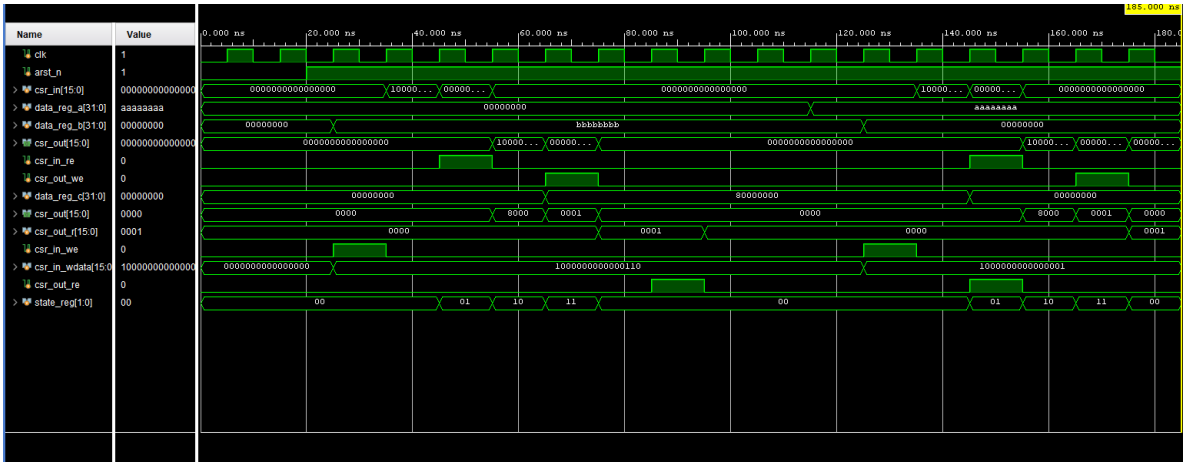

Figure 14: test 5 & 6 waveform.

6.8.6 Test 7

```
    // Test 7: Try to write csr_in while FSM is busy shifting
```

```
    intf.write_data_reg_a(32'h12345678);
    intf.write_csr_in(16'b1000_0001_1111_1001);
    repeat (8) @(posedge clk);
    intf.write_csr_in(16'b1000_0000_0010_0101);
    intf.wait_done();
```

Listing 14: Test 7 implementation

Test 7 is meant to give an incorrect use of semicolab, trying to set a input value when the module has not completed the proccess.

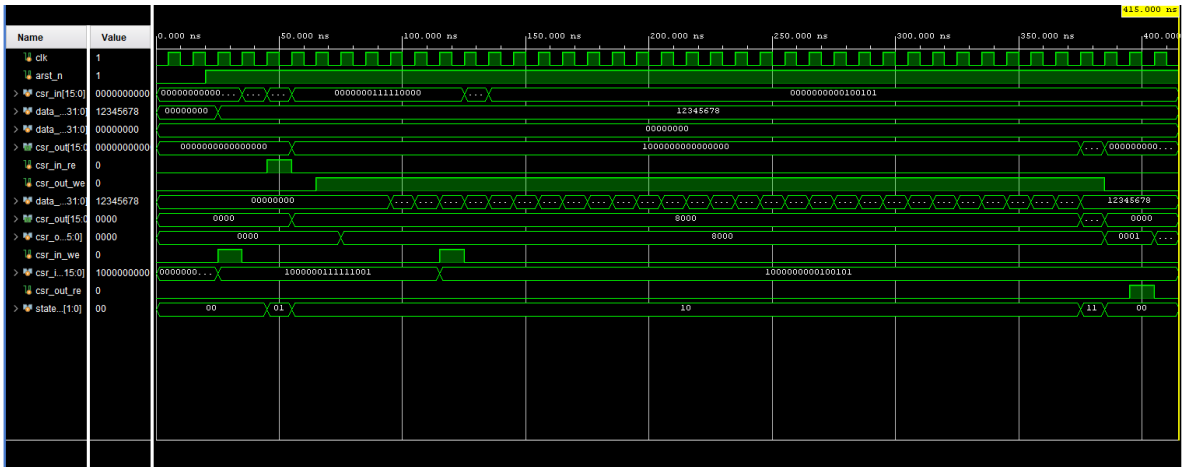

Figure 15: test 7 waveform.

As seen in the waveform when trying to write a new value before computing is completed results in the module ignoring the new writing and continuing with the current computing.

6.8.7 Test 8

```
    intf.write_data_reg_a(32'hCAFEBABE);
    intf.write_csr_in(16'b1000_0001_1111_1001);
    repeat (4) @(posedge clk);
    #2 arst_n <= 1'b0;
    @(posedge clk);
    arst_n <= 1'b1; // Release reset
    repeat (10) @(posedge clk);
```

Listing 15: Test 7 implementation

Test 8 evaluates the correct behavior with the reset, to ensure the reset occurs when set.

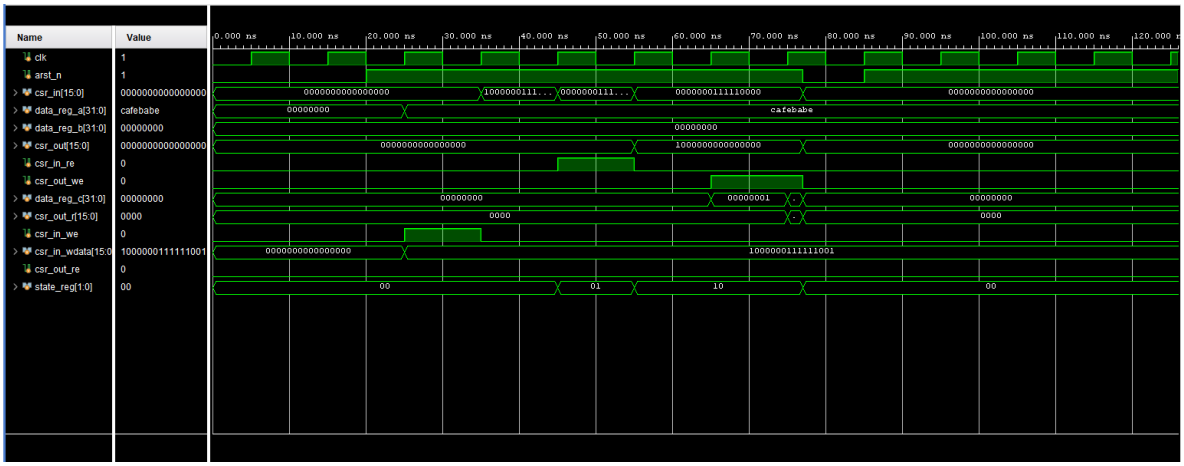

Figure 16: test 8 waveform.

7 How to Integrate Your Own RTL Module into the SemiColab User Tile

To design and integrate your own RTL module using the SemiColab tile standard explained in this guide and ensure a proper utilization of all available I/O ports, follow these steps:

- **Define your logic:** Use the RTL tile template, based on the ports given by the SemiColab ip tile wrapper, define your logic, you can implement a variety of RTL modules as long as the port conventions are being followed.
- **Respect the port naming:** Use the same port names as the template (e.g., `csr_in`, `data_reg_a`, `csr_out`).
- **Respect the port width:** Use the same width for the ports as the template (e.g., `csr_in 16 bits`, `data_reg_a 32 bits`, `csr_out 16 bits`).
- **Map the csr_in bits:**
  - Use bits [15:12] for pulse input signals
  - Use bits [11:4] for stable input configuration
  - Use bits [3:0] for clear-on-read inputs
- **Map the csr_out bits:**
  - Use bits [15:4] for stable output bits
  - Use bits [3:0] for clear-on-read outputs
- **Use the tile outputs:** Use `data_reg_c` and `csr_out` as your output signals.
- **Drive `csr_in_re` and `csr_out_we`** appropriately to synchronize with the wrapper.
- **Test your design:** Use the provided testbench interface and tasks to validate the expected behavior of your RTL module.

Appendix A: RTL Tile Template

```
module ip_tile_template #(
parameter REG_WIDTH = 32,
parameter CSR_IN_WIDTH = 16,
parameter CSR_OUT_WIDTH = 16)

(
input wire clk,
input wire arst_n,
input wire [CSR_IN_WIDTH-1:0] csr_in,
input wire  [REG_WIDTH - 1:0] data_reg_a,
input wire  [REG_WIDTH - 1:0] data_reg_b,
output wire [REG_WIDTH - 1:0] data_reg_c,
output wire [CSR_OUT_WIDTH-1:0] csr_out,
output wire csr_in_re,
output wire csr_out_we
    );
// User's Logic
endmodule
```

Listing 16: RTL Template for Tile Integration

Appendix B: Testbench Tile Template

```
module tb_ip_tile;

        parameter CSR_IN_WIDTH = 16;
        parameter CSR_OUT_WIDTH = 16;
        parameter REG_WIDTH = 32;

        bit clk;
        bit arst_n;
        interface_ip_tile intf(clk, arst_n);

        always #5ns clk = !clk;
        assign #20ns arst_n = 1'b1;

        ip_tile_user_name DUT(
          .clk(clk),
          .arst_n(arst_n),
          .csr_in(intf.csr_in),
          .csr_in_re(intf.csr_in_re),
          .data_reg_a(intf.data_reg_a),
          .data_reg_b(intf.data_reg_b),
          .csr_out(intf.csr_out),
          .csr_out_we(intf.csr_out_we),
          .data_reg_c(intf.data_reg_c)
        );

endmodule
```

Listing 17: Testbench Template for Tile Verification

Appendix C: Interface Tile Template

```
interface interface_ip_tile(input logic clk, input logic arst_n);

        // CSR parameters
        parameter CSR_IN_WIDTH = 16;
        parameter CSR_OUT_WIDTH = 16;
        parameter REG_WIDTH = 32;

        // Declaration of signals used by user tile
  bit [CSR_IN_WIDTH - 1 : 0] csr_in;
  bit [REG_WIDTH - 1 : 0] data_reg_a;
  bit [REG_WIDTH - 1 : 0] data_reg_b;
  logic [CSR_OUT_WIDTH - 1 : 0] csr_out;
  logic csr_in_re;
  logic csr_out_we;
  logic [REG_WIDTH - 1 : 0] data_reg_c;

        // Declaration of signals used by testbench only (can only be accessed
             by interface tasks/functions)
  logic [CSR_OUT_WIDTH - 1 : 0] csr_out_r;
  bit csr_in_we;
        bit [CSR_IN_WIDTH - 1 : 0] csr_in_wdata;
  bit csr_out_re;

        // This modport should be used by user to connect with his/her tile
            logic
        modport user_tile_modport(
                input csr_in, data_reg_a, data_reg_b,
                output csr_in_re, csr_out, csr_out_we, data_reg_c
        );

        // This task can be used to assign a value to data_reg_a user tile
            input
        task write_data_reg_a(logic [REG_WIDTH - 1 : 0] data);
                @(posedge clk);
                data_reg_a <= data;
        endtask

        // This task can be used to assign a value to data_reg_b user tile
            input
        task write_data_reg_b(logic [REG_WIDTH - 1 : 0] data);
                @(posedge clk);
                data_reg_b <= data;
        endtask

        // This task can be used to read the current value of user tile output
             data_reg_c
        function logic [REG_WIDTH - 1 : 0] read_data_reg_c();
                return data_reg_c;
        endfunction

        // This task can be used to read user tile output csr_out, this will
            clear all clear on read csr_out bits
        task read_csr_out(output logic [CSR_OUT_WIDTH - 1 : 0] csr_out_data);
                csr_out_data <= csr_out_r;
                csr_out_re <= 1'b1;
                @(posedge clk);
                csr_out_re <= 1'b0;
        endtask

```

```
54          // This task can be used to write user tile input csr_in
55          task write_csr_in(logic [CSR_IN_WIDTH - 1 : 0] csr_in_data);
56                  csr_in_wdata <= csr_in_data;
57                  csr_in_we <= 1'b1;
58                  @(posedge clk);
59                  csr_in_we <= 1'b0;
60          endtask
61
62          // This procedural block replicates the csr_out register
63          always_ff@(posedge clk, negedge arst_n) begin
64                  if(!arst_n) begin
65                          csr_out_r <= {CSR_OUT_WIDTH{1'b0}};
66                  end else begin
67        if(csr_out_we) // Writing is taking priority over clear on read
68          csr_out_r <= csr_out;
69        else if(csr_out_re)
70          csr_out_r[3 : 0] <= 4'd0; // 4 Least Significant Bits are clear on
                read, so we have to clear when reading
71                  end
72          end
73
74          // This procedural block implements the csr_in register
75    always@(posedge clk, negedge arst_n) begin
76      if(~arst_n) begin
77        csr_in <= {CSR_IN_WIDTH{1'b0}};
78      end else begin
79        csr_in[CSR_IN_WIDTH - 1 : CSR_IN_WIDTH - 4] <= 4'd0; // 4 Most
            Significant Bits are single pulse, so we have to clear every clock
            cycle except when we write
80        if(csr_in_we) // Writing is taking priority over clear on read
81          csr_in <= csr_in_wdata;
82        else if(csr_in_re)
83          csr_in[3 : 0] <= 4'd0; // 4 Least Significant Bits are clear on read,
              so we have to clear when reading
84      end
85    end
86
87  endinterface
```

Listing 18: Interface Template for Tile Verification