PA1: Unsigned Multiplier and Unsigned Divider

a. Implement A 32-bit Unsigned Complete Multiplier

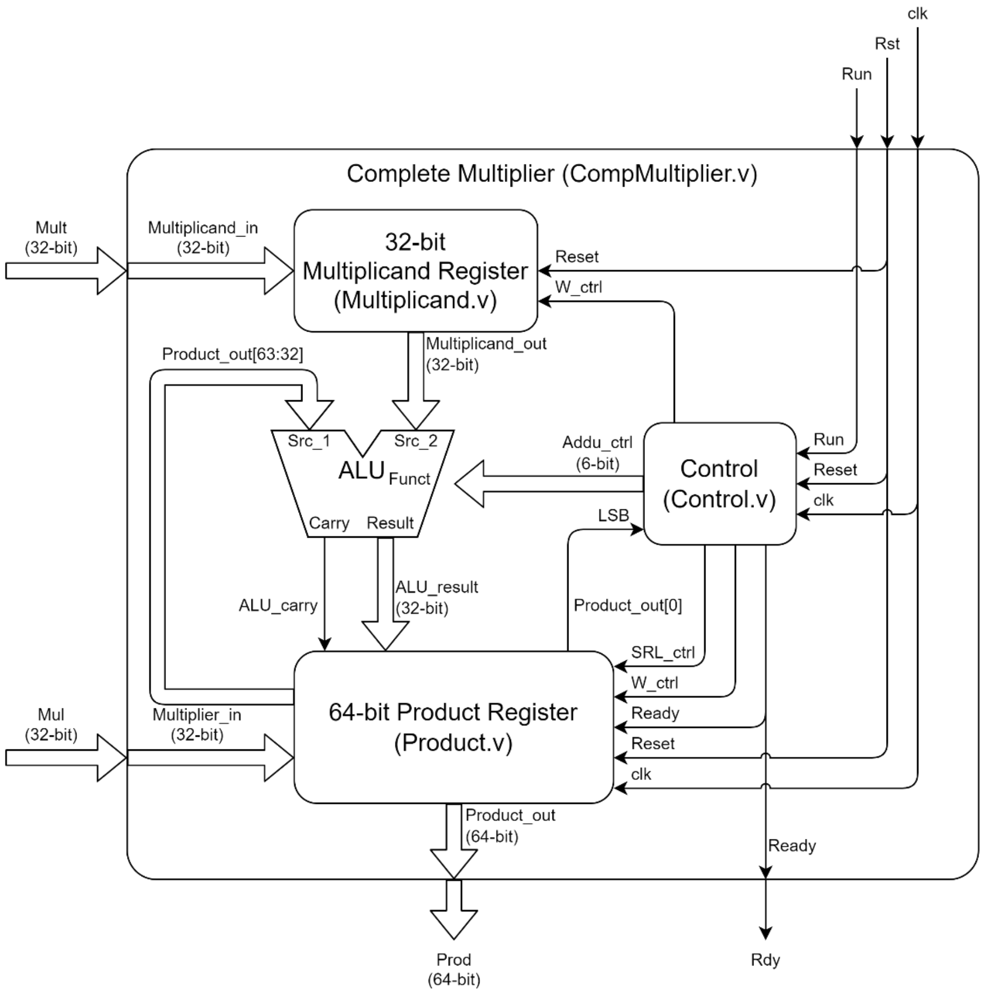


Fig. 1: 32-bit unsigned multiplier

Implement a 32-bit unsigned multiplier as shown in Fig. 1.

Note: Do not use multiplication instructions (e.g., A*B). Please use add and shift to implement multiplication.

1. Module Definition

The module name and I/O ports of the top-level module CompMultiplier must follow I/O interface definitions. The names of the modules inside the system can be defined by the designer, but the number of modules should be equal to Fig. 1. The data bus and control bus in the system can be added or deleted by the designer. Fig. 1 is one of the solutions for the designer's reference.

I/O Interface

```
module CompMultiplier (
        output [63:0] Prod,
        output Rdy,
        input [31:0] Mult,
        input [31:0] Mul,
        input Run,
        input Rst,
        input clk
);
```

2. Signal Description

Signal symbol	Signal name	Signal Description
Prod	64-bit calculation result	This signal is set as an output before the **Rdy** signal is activated to forbid the testbench to read the previous calculation result.
Rdy	completion signal	The “high” level represents the system has completed multiplication and maintained the result.
Mult	32-bit multiplicand	The signal is generated by the testbench and updated when the **Rst** signal is “high”.
Mul	32-bit multiplier	The signal is generated by the testbench and updated when the **Rst** signal is “high”.
Run	execution signal	The system performs multiplication as the Run signal is “high” level.
Rst	initialization signal	The “high” level indicates that the system is initialized before multiplication, and the output results are set to zero. This signal has the highest priority and is generated by the testbench.
clk	clock signal	Periodic square waves are generated by the testbench for synchronizing each signal with the drive system.

3. External Signal Action Flow

This testbench (tb_CompMultiplier.v) will initialize the **Rst** signal for a time unit (between two **positive edges of clk**) before each multiplication starts. When the **Rst** signal is high, testbench will read the value in tb_CompMultiplier.in and output the corresponding values to **Mult** and **Mul** respectively. After one clock (the positive edge of the clock), the testbench set the **Rst** signal as low. Then, after one clock (the positive edge of the clock), the testbench set the **Run** signal as high. The **Rdy** signal shall be high after the multiplication is completed. When the **Rdy** signal is high, the testbench store the result of **Prod** and the current tick in tb_CompMultiplier.out. If there are more operations to be executed in tb_CompMultiplier.in, the procedure will be restarted at the positive edge of the clock. Refer to Fig. 2 for the example waveform.

- Note: The testbench will continuously wait for the **Rdy** signal. If a system fails to set the **Rdy** signal, the simulation will not stop. The designer shall deal with this issue.
- Note: The assignment only provides the testbench of the top-level module, and the functionality of other internal modules needs to be tested by yourself. Designers shall write the modifications based on the simplified testbench provided in this assignment.
- Note: The provided test input file (tb_CompMultiplier.in) needs to be added into the folder (..\testbench\) which is the same as the folder of the project. The instructions are written in hexadecimal, and each line contains 2 values as respectively multiplicand and multiplier, separated by an "under line". The format of instruction is as follows: 32-bit multiplicand (**Mult**) "_" 32-bit multiplier (**Mul**). Designers can add more instructions for the testing.

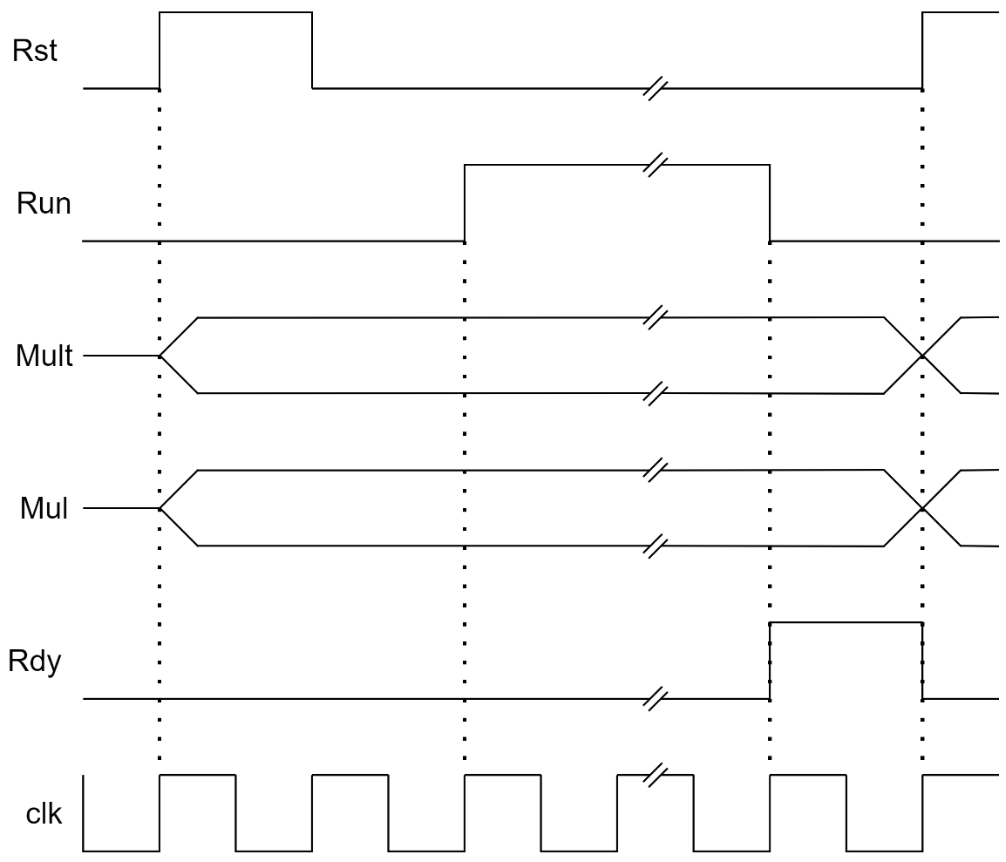


Fig. 2：Waveform Example

4. Operation Process

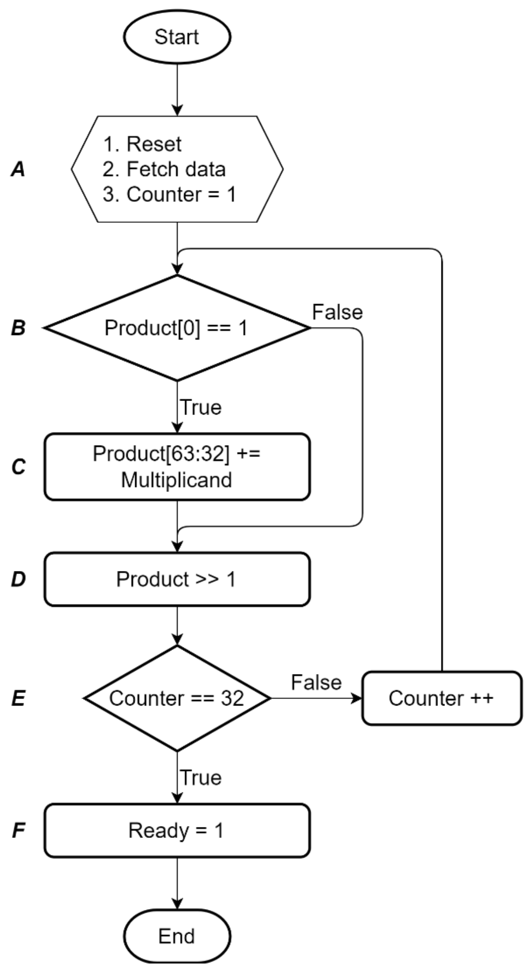


Fig. 3：Multiplication flow chart

Flowchart Description

Step A: The **Rst** signal initializes the system. The **Run** signal triggers the reading of the **Mult** and **Mul** and then starts the operation.
Step B: Check if the LSB of the **Mul** is 1.
Step C: Accumulate the left half of the result register with the **Mult (**with ALU carry flag).
Step D: Shift the result register to the right by 1-bit.
Step E: If the counter is less than 32, increase the value of the counter by 1.
Step F: Set the Ready signal as 1 to complete the multiplication

Example (4-bit multiplication operation)

Counter.Step	Prod	Mult
	0000_0000	0000
0.A	0000_0011	0010
1.C	0010_0011	0010
1.D	0001_0001	0010
2.C	0011_0001	0010
2.D	0001_1000	0010
3.B	0001_1000	0010
3.D	0000_1100	0010
4.B	0000_1100	0010
4.D	0000_0110	0010
4.F	0000_0110	0010

b. Implement A 32-bit Unsigned Complete Divider

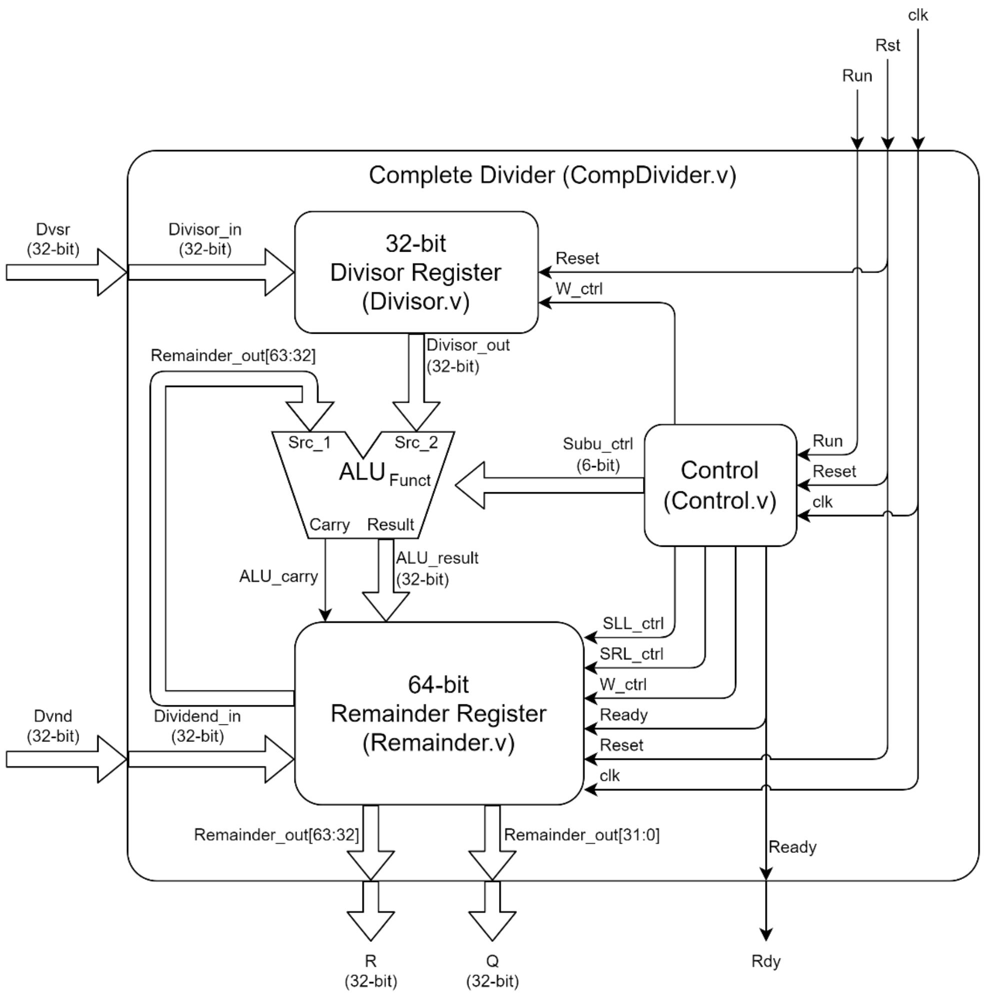


Fig. 4： 32-bit unsigned divider

Implement a 32-bit unsigned divider as shown in Fig. 4.

The design rules are the same as the multiplier, please refer to the content of the lecture notes for the operation process.

I/O Interface

```
module CompDivider (
        output [31:0] Q,
        output [31:0] R,
        output Rdy,
        input [31:0] Dvnd,
        input [31:0] Dvsr,
        input Run,
        input Rst,
        input clk
);
```

Signal Description

Signal symbol	Signal name
Q	32-bit Quotient
R	32-bit Reminder
Rdy	completion signal
Dvnd	32-bit Dividend
Dvsr	32-bit Divisor
Run	execution signal
Rst	initialization signal
clk	clock signal

- Note: Do not use the division command（e.g., A/B）. Please use subtraction and shift to perform division.
- Note: The mathematically invalid operation (e.g., dividing by 0) is not within the scope of this project.
- Note: The provided test input file (tb_CompDivider.in) needs to be added into the folder (..\testbench\) which is the same as the folder of the project. The instructions are written in hexadecimal, and each line contains 2 values as respectively dividend and divisor. The format of instruction is as follows: 32-bit dividend (**Dvnd**) "_" 32-bit divisor (**Dvsr**). Designers can add more instructions for the testing.

c. Submission

Report structure:

a. Cover.
b. Screenshots and descriptions of each module.
c. Screenshots and descriptions of each test bench for all modules.
d. Screenshots and descriptions of the test results (hexadecimal waveforms) for each module.
e. Conclusion and insights.

※ Convert the report to PDF and name it the student ID - "BYYYDDXXX.pdf".

Files required for PA1:

- BYYYDDXXX.zip
  - Part1
    - CompMultiplier.v
    - Multiplicand.v
    - ALU.v
    - Product.v
    - Control.v
  - Part2
    - CompDivider.v
    - Divisor.v
    - ALU.v
    - Remainder.v
    - Control.v
  - Report (BYYYDDXXX.pdf)

- Note: Please make sure all your program files and PDF files are directly in the zipped file, not all wrapped in one folder.

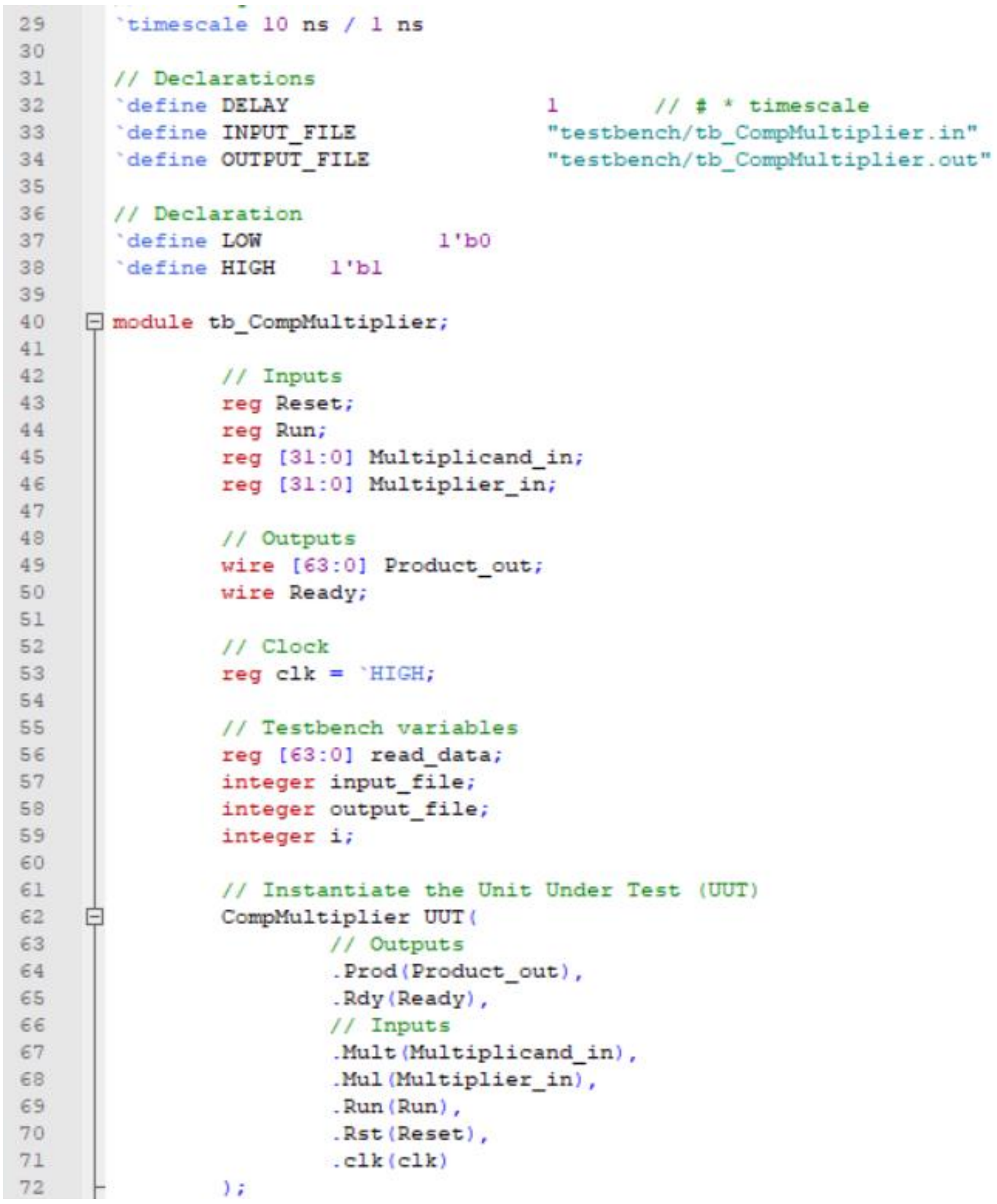
Grading:

a. (40 points) Multiplier.
b. (20 points) Divider.
c. (20 points) Screenshots of each program, and description of the process.
d. (10 points) Screenshots of each testbench, and description of the test functions.
e. (10 points) Simulation results with analysis.
f. Follow naming rules and file formats.
g. No plagiarism.

We will test your modules with another generated testbench.

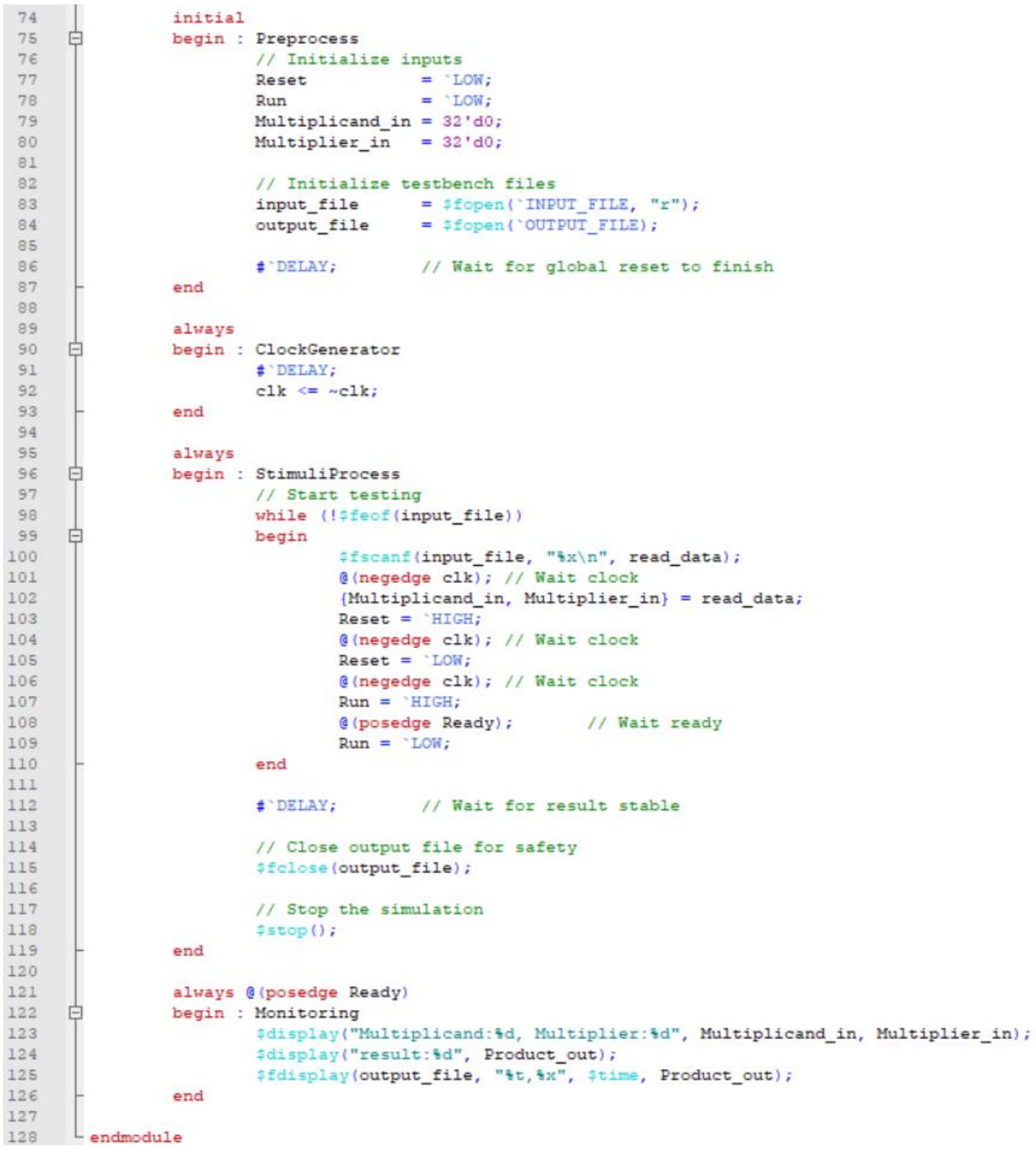
Deadline: 2024-04-14 13:00 on moodle

Appendix: Testbench



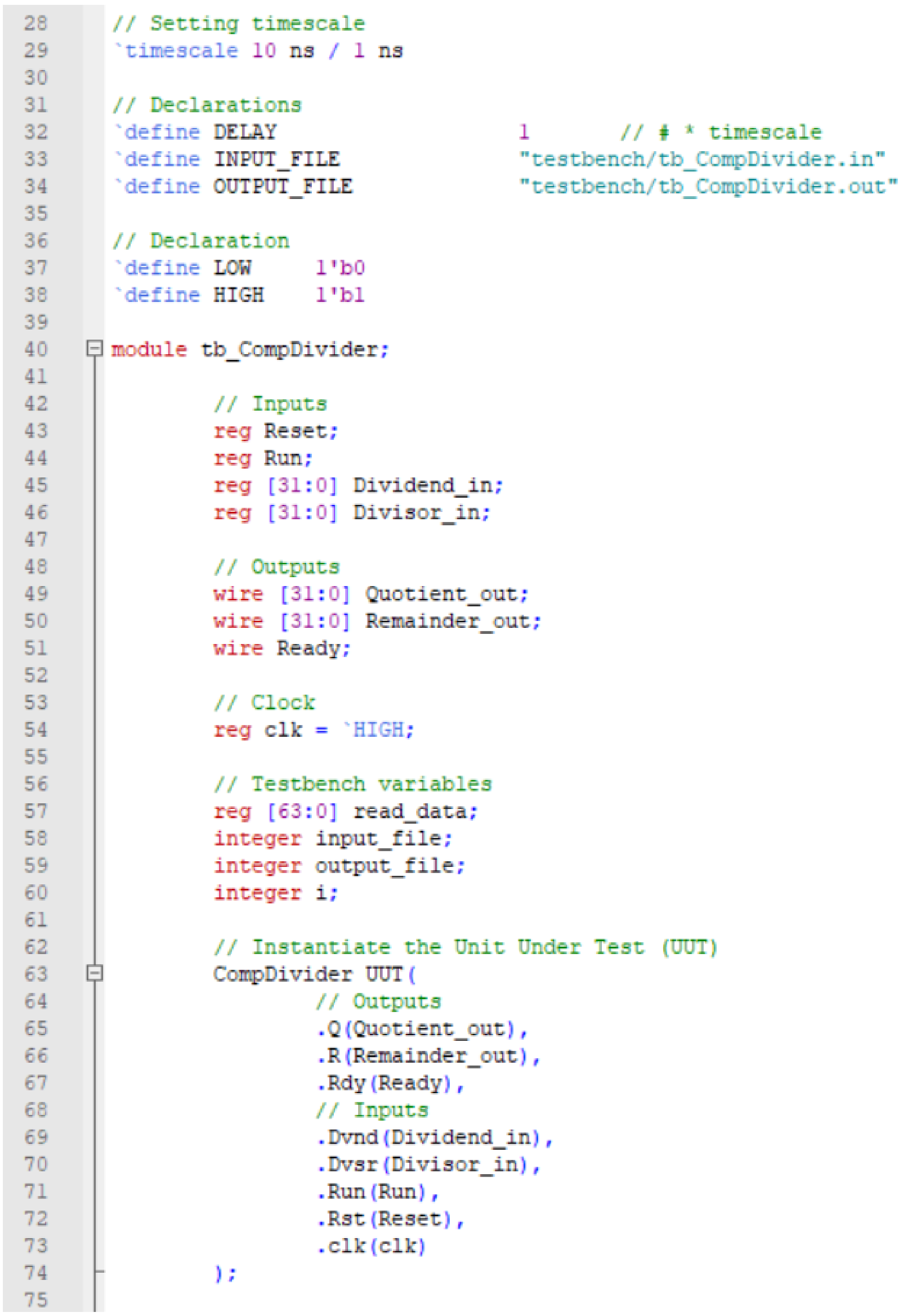
```
`timescale 10 ns / 1 ns

// Declarations
`define DELAY                     1        // # * timescale
`define INPUT_FILE                "testbench/tb_CompMultiplier.in"
`define OUTPUT_FILE               "testbench/tb_CompMultiplier.out"

// Declaration
`define LOW             1'b0
`define HIGH    1'b1

module tb_CompMultiplier;

        // Inputs
        reg Reset;
        reg Run;
        reg [31:0] Multiplicand_in;
        reg [31:0] Multiplier_in;

        // Outputs
        wire [63:0] Product_out;
        wire Ready;

        // Clock
        reg clk = `HIGH;

        // Testbench variables
        reg [63:0] read_data;
        integer input_file;
        integer output_file;
        integer i;

        // Instantiate the Unit Under Test (UUT)
        CompMultiplier UUT(
                // Outputs
                .Prod(Product_out),
                .Rdy(Ready),
                // Inputs
                .Mult(Multiplicand_in),
                .Mul(Multiplier_in),
                .Run(Run),
                .Rst(Reset),
                .clk(clk)
        );
```

Fig. 5 (a): tb_CompMultiplier.v



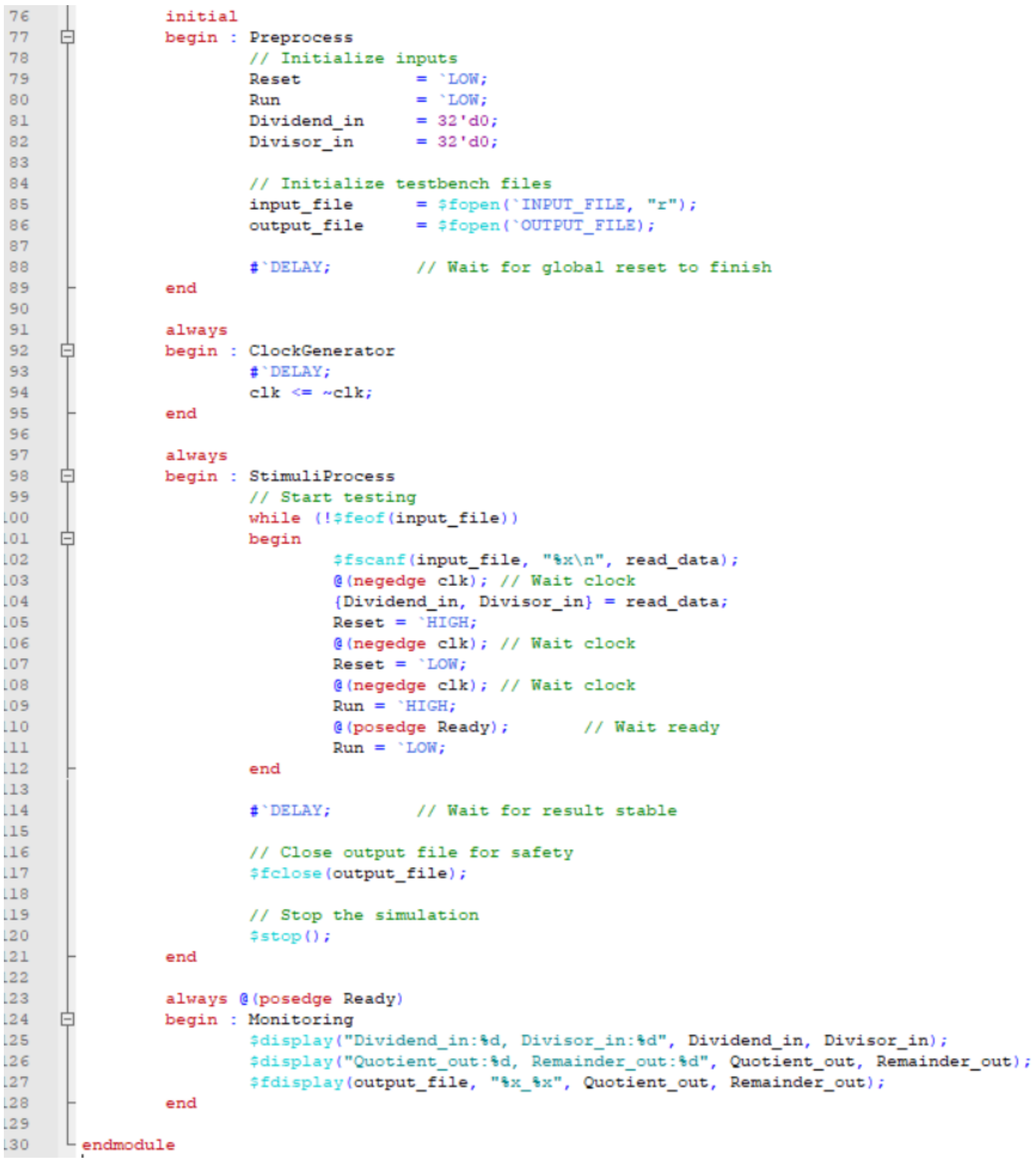
```
        initial
        begin : Preprocess
                // Initialize inputs
                Reset           = `LOW;
                Run             = `LOW;
                Multiplicand_in = 32'd0;
                Multiplier_in   = 32'd0;

                // Initialize testbench files
                input_file      = $fopen(`INPUT_FILE, "r");
                output_file     = $fopen(`OUTPUT_FILE);

                #`DELAY;        // Wait for global reset to finish
        end

        always
        begin : ClockGenerator
                #`DELAY;
                clk <= ~clk;
        end

        always
        begin : StimuliProcess
                // Start testing
                while (!$feof(input_file))
                begin
                        $fscanf(input_file, "%x\n", read_data);
                        @(negedge clk); // Wait clock
                        {Multiplicand_in, Multiplier_in} = read_data;
                        Reset = `HIGH;
                        @(negedge clk); // Wait clock
                        Reset = `LOW;
                        @(negedge clk); // Wait clock
                        Run = `HIGH;
                        @(posedge Ready);       // Wait ready
                        Run = `LOW;
                end

                #`DELAY;        // Wait for result stable

                // Close output file for safety
                $fclose(output_file);

                // Stop the simulation
                $stop();
        end

        always @(posedge Ready)
        begin : Monitoring
                $display("Multiplicand:%d, Multiplier:%d", Multiplicand_in, Multiplier_in);
                $display("result:%d", Product_out);
                $fdisplay(output_file, "%t,%x", $time, Product_out);
        end

endmodule
```

Fig. 6 (b): tb_CompMultiplier.v (cont.)



```verilog
// Setting timescale
`timescale 10 ns / 1 ns

// Declarations
`define DELAY                   1       // # * timescale
`define INPUT_FILE              "testbench/tb_CompDivider.in"
`define OUTPUT_FILE             "testbench/tb_CompDivider.out"

// Declaration
`define LOW     1'b0
`define HIGH    1'b1

module tb_CompDivider;

        // Inputs
        reg Reset;
        reg Run;
        reg [31:0] Dividend_in;
        reg [31:0] Divisor_in;

        // Outputs
        wire [31:0] Quotient_out;
        wire [31:0] Remainder_out;
        wire Ready;

        // Clock
        reg clk = `HIGH;

        // Testbench variables
        reg [63:0] read_data;
        integer input_file;
        integer output_file;
        integer i;

        // Instantiate the Unit Under Test (UUT)
        CompDivider UUT(
                // Outputs
                .Q(Quotient_out),
                .R(Remainder_out),
                .Rdy(Ready),
                // Inputs
                .Dvnd(Dividend_in),
                .Dvsr(Divisor_in),
                .Run(Run),
                .Rst(Reset),
                .clk(clk)
        );

```

Fig.6 (a): tb_CompDivider.v



```
        initial
        begin : Preprocess
                // Initialize inputs
                Reset           = `LOW;
                Run             = `LOW;
                Dividend_in     = 32'd0;
                Divisor_in      = 32'd0;

                // Initialize testbench files
                input_file      = $fopen(`INPUT_FILE, "r");
                output_file     = $fopen(`OUTPUT_FILE);

                #`DELAY;        // Wait for global reset to finish
        end

        always
        begin : ClockGenerator
                #`DELAY;
                clk <= ~clk;
        end

        always
        begin : StimuliProcess
                // Start testing
                while (!$feof(input_file))
                begin
                        $fscanf(input_file, "%x\n", read_data);
                        @(negedge clk); // Wait clock
                        {Dividend_in, Divisor_in} = read_data;
                        Reset = `HIGH;
                        @(negedge clk); // Wait clock
                        Reset = `LOW;
                        @(negedge clk); // Wait clock
                        Run = `HIGH;
                        @(posedge Ready);       // Wait ready
                        Run = `LOW;
                end

                #`DELAY;        // Wait for result stable

                // Close output file for safety
                $fclose(output_file);

                // Stop the simulation
                $stop();
        end

        always @(posedge Ready)
        begin : Monitoring
                $display("Dividend_in:%d, Divisor_in:%d", Dividend_in, Divisor_in);
                $display("Quotient_out:%d, Remainder_out:%d", Quotient_out, Remainder_out);
                $fdisplay(output_file, "%x_%x", Quotient_out, Remainder_out);
        end

endmodule
```

Fig.6 (b): tb_CompDivider.v (cont.)