

M.EEC041 - Digital Systems Design

2025/2026

Laboratory exercise 2

October 2025

1. Introduction

This project consists in implementing a sequential divider for 32-bit unsigned integer numbers. The circuit must be built as a synthesizable Verilog module representing exactly the logic diagram shown in figure 1. The interface of the module is:

```
module psddivide(
    input          clock,      // master clock, posedge
    input          reset,      // synch reset, active high
    input          start,      // start a new division
    input          stop,       // load output registers
    input [31:0]   dividend,   // dividend
    input [31:0]   divisor,   // divisor
    output reg [31:0] quotient, // quotient (register)
    output reg [31:0] rest     // rest (register)
);
```

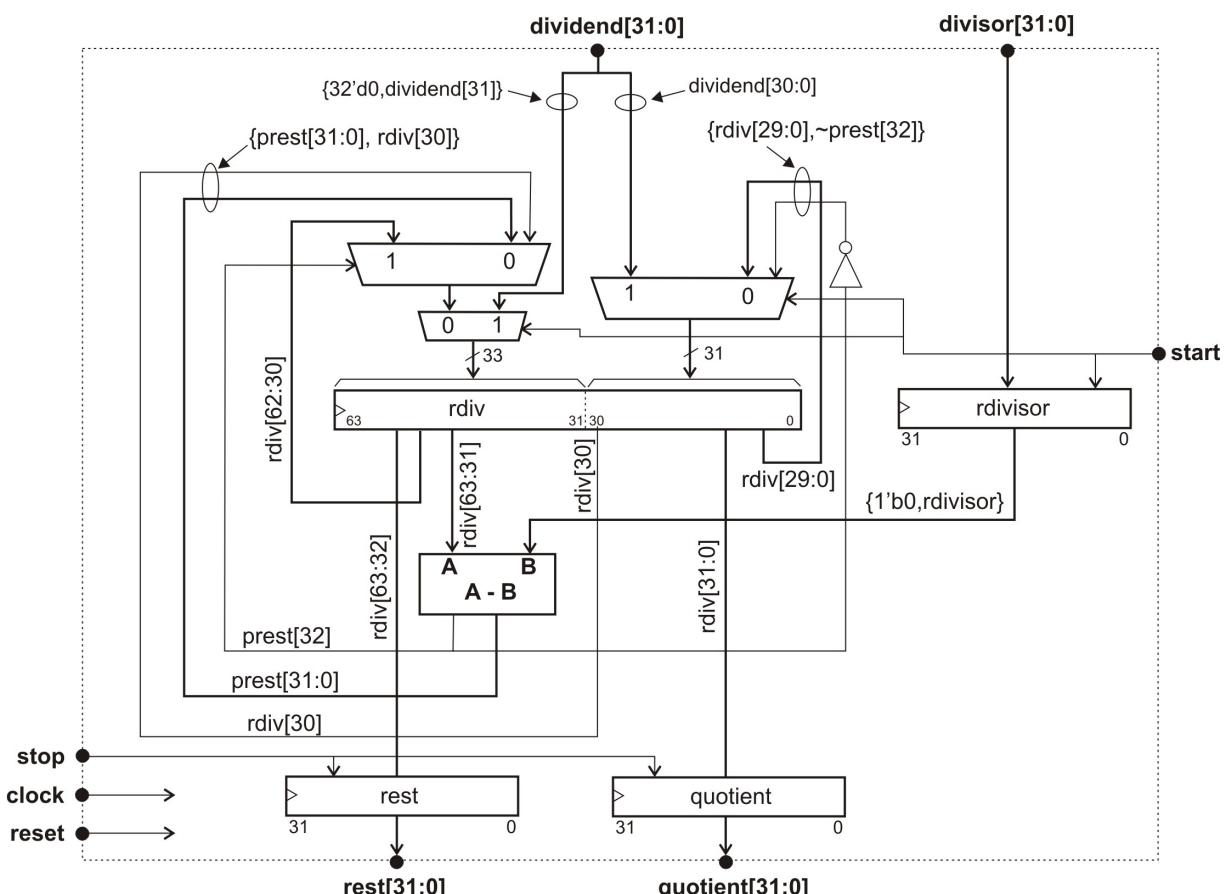


Figure 1 - RTL diagram of the sequential divider.

2. Functional specification

This circuit implements the sequential non-restoring division algorithm, similar to the manual paper-and-pencil process. The circuit is synchronous with the global clock signal and computes the quotient and rest in a number of iterations equal to the number of bits of the dividend, performing one iteration per clock cycle. The operands and results are unsigned integer 32-bit numbers. If the divisor is zero, the quotient and rest are not meaningful and there is no special treatment for that case.

The function implemented by this circuit is functionally equivalent to the C function below:

```
HWdivider( unsigned dividend, unsigned divisor,
            unsigned quotient, unsigned rest )
{
    quotient = dividend / divisor;
    rest = dividend % divisor;
}
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with a 64-bit word formed by the dividend in the 32 least significant bits and zeros in the 32 most significand bits. The partial rest (**prest**) is updated in each iteration by subtracting the divisor (register **rdivisor**) from the high part of the register **rdiv**. If the partial rest is positive (**prest[32]==0**), the register **rdiv** is loaded with the partial rest (**prest**) in its high significant bits and shifted left, loading into the LSB the negation of the sign bit of the partial rest (**prest[32]**). This bit represents a new quotient bit that is 1 if the partial rest is positive (meaning that the divisor “fits” in the partial dividend), or 0 otherwise.

To execute a division, the input **start** must be set during one clock cycle to load the divisor into register **rdivisor** and the dividend into the low part of register **rdiv**. Then the iterative process proceeds for exactly 32 clock cycles. After that, one additional clock cycle is necessary with the input **stop** set, to load the two output registers. The timing diagram representing the two control signals is represented in figure 2.

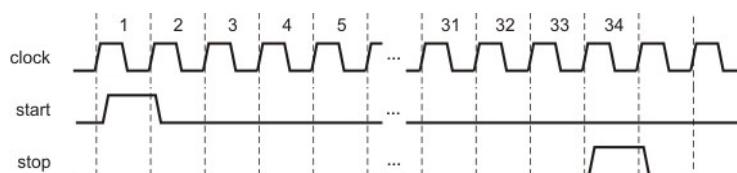


Figure 2 - Timing diagram of the control signals **start** and **stop**.

3. Implementation

The system must be implemented as a single behavioral Verilog synthesizable module, using a single clock signal and a global synchronous reset, active high (all registers have the same clock signal). The activation of the reset signal must set all

the registers to zero. The sequential and combinational blocks shown in figure 1 must be coded with appropriate Verilog statements **assign** and **always**. Include comments in your code to explain the relationship between each Verilog process and the corresponding block in the block diagram. The module should be built in a single hierarchical level and should not instantiate any other module.

The register **rdivisor** uses the input **start** as a load enable (or clock enable) signal and the two output registers use the input **stop** for the same purpose. The register **rdiv** does not have a clock enable signal.

4. Verification

To verify this model you must use and the testbench included in the project archive (`./src/verilog/testbench/psddivide_tb.v`). The task **execdivide** already included in the testbench module implements the correct sequence of signals **start** and **stop** to perform a division. You are free to improve the testbench in order to improve the verification procedure. The simulations must be run in folder `./sim/iverilog` (this folder also contains the file `psddivide.dat` with the names of the files to compile with iverilog; run the simulation with `iverilog -f psddivide.dat -o mysimout`, then `vvp mysimout`)

5. Additional developments

5.1 Use one parameter to configure the number of bits for the two operands and results). The external controller that sequences the activation of the **start** and **stop** signals must use the same parameter to generate the appropriate timing for those signals, as the number of clock cycles is equal to the number of bits of the dividend. See the Verilog documentation to learn how to use parameters in the module definitions and how to assign values to parameters when a module is instantiated.

5.2 Build a synthesizable module implementing a sequential controller (finite-state machine) to generate the **start** and **stop** signals required by the divisor, and activate a **busy** output signal while the division process is running. The controller receives a one-clock pulse in input **run** (clock cycle #1 in the timing diagram below), asserts the **start** signal, counts the number of clocks necessary to complete the division (this is equal to the number of bits of the dividend) and at the end asserts the **stop** signal to load the output registers (clock cycle #34). Figure 3 shows the timing diagram of these signals, considering a 32-bit dividend.

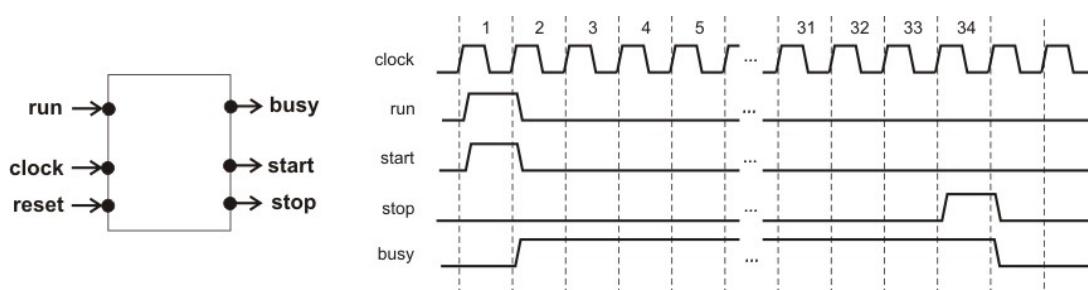


Figure 3 - Timing diagram to be implemented by the sequential controller.