EEC0055 - Digital Systems Design

2020/2021

Laboratory project 1 - V1.0
19 October 2020

1. Introduction

In this project you will create a synthesisable Verilog module that implements a sequential divider for 32-bit unsigned integer numbers, implementing the logic diagram shown in figure 1.

The header of the module to build is:

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

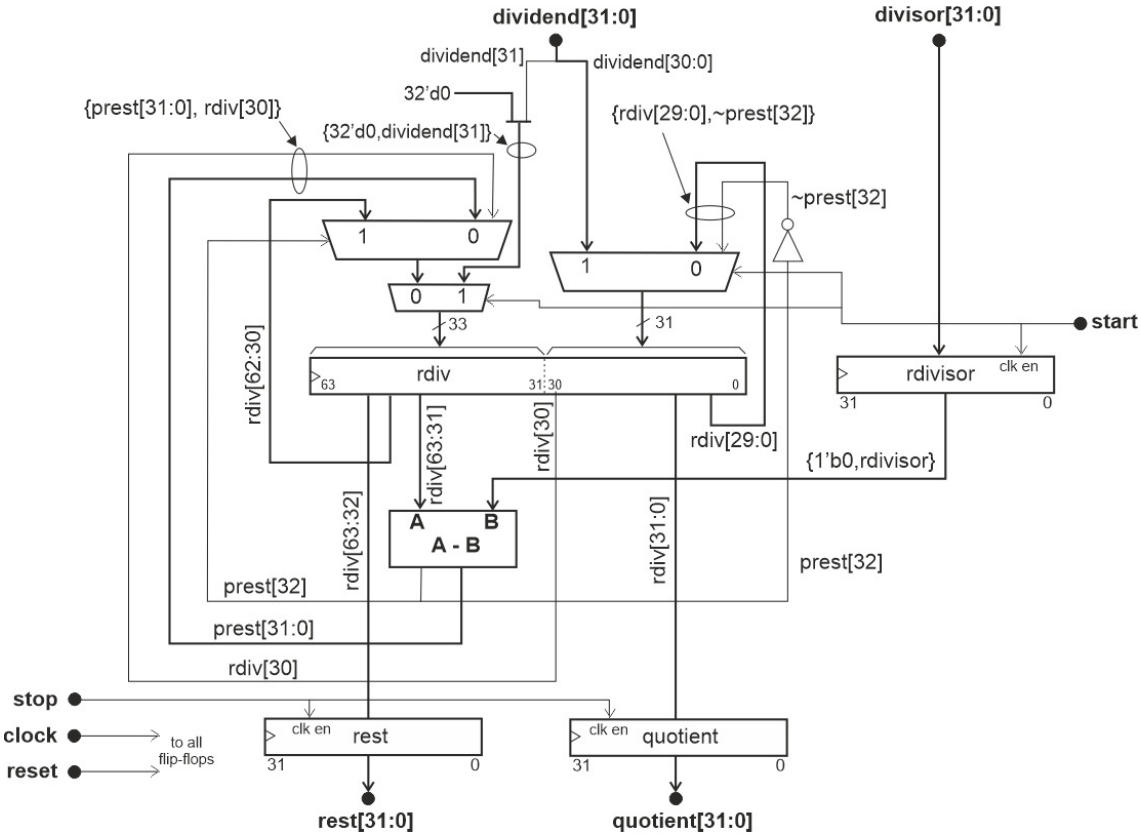

Figure 1 – RTL diagram of the sequential divider.

2. Functional description

Figure 1 shows the RTL (Register-Transfer Level) schematic of the circuit to build. Although you do not need to understand the way the division algorithm is implemented by the circuit your will implement, this section presents a brief explanation of the function of each block and the way the divider works.

The sequential divider uses a 64-bit register (**rdiv**) that is initially loaded with the dividend (when a new division starts) and another 32-bit register (**rdivisor**) that is loaded with the divisor. The register **rdiv** is shown divided by a dashed line into a 31-bit section (right part) and a 33-bit section (left part). The rightmost 31 bits of the dividend are initially loaded into the right part of **rdiv** and the most significant bit, left-padded with zeroes, is loaded into the left part of **rdiv**.

The partial rest **prest** is calculated by subtracting the divisor from the left part of register **rdiv**. If the partial rest is positive (indicated by the condition **prest**[32]==0), the register **rdiv** is loaded with the partial rest (**prest**) in the high significant bits and shifted left to load into its LSB a bit equal to one (the complement of the sign bit of the partial rest, **prest**[32]), which represents a new quotient. If the partial rest is negative (**prest**[32]==1), the register **rdiv** is only shifted left, loading into its LSB a new bit zero for the quotient.

The proposed circuit performs the load and shift operations in a single clock cycle, thus completing the integer iterative division in 32 clock cycles. Two additional clock cycles are necessary to load the operands before staring the division algorithm and to load the two results (quotient and rest) to the output registers. To start a new division the input **start** must be set during one single 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 32 clock cycles and at the end one additional clock is necessary with the input **stop** set, to load the output registers with the quotient (**rdiv[31:0]**) and the rest (**rdiv[63:32]**)

3. Implementation

The system must be implemented as a behavioral Verilog synthesizable module using a single clock signal and a global synchronous reset active high. This means all registers will have the same clock signal and the same reset signals.

You should write your module using a set of **assign** and **always** processes to model the combinational and sequential blocks shown in figure 1. There are a few different approaches to build the module: you may use individual **assign** statements for modeling each combinational block shown in the circuit diagram (the multiplexers, inverter and subtractor), or you can combine all those blocks in only one **always @*** statement. In a similar way, you can use one sequential process implemented by a **always @(posedge)** statement for modelling each register or you can model all the registers using a single sequential **always**; you can model the combinational logic by using one or more combinational processes or you can include the combinational logic functions within the sequential processes.

Note that 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. Register **rdiv** do not have a clock enable control.

4. Verification

To verify your Verilog model you must adapt the testbench given and choose a convenient and limited set of operands, as it is impossible to simulate all the combinations of the input operands. Thus, you must select a relatively small set of stimuli to give enough confidence that the circuit will work correctly for any operand, verify the correct operation of the **reset**, **start** and **stop** control signals. Your testbench must guarantee a minimum total coverage of only 95% (for the divider module only), considering at least the coverage metrics for statements, branches and toggles, and run a successful simulation in less than 15 minutes (running QuestaSim in the PCs of the lab). Note that you may need to simulate up to some millions of divisions, depending on the way the operands are generated. The testbench must also include some mechanism for automating the verification of the results produced by your divider module.

The testbench given in the project archive includes a very simple task to perform a simulation of just one division and an example of how to call that task. You can use the a **for( ), while** or **repeat** Verilog looping statements to run a loop for simulating a set of operands. You can also use the Verilog task **$random** to obtain a pseudo-random 32-bit number or you can select a specific set of operands to run simulation.