EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

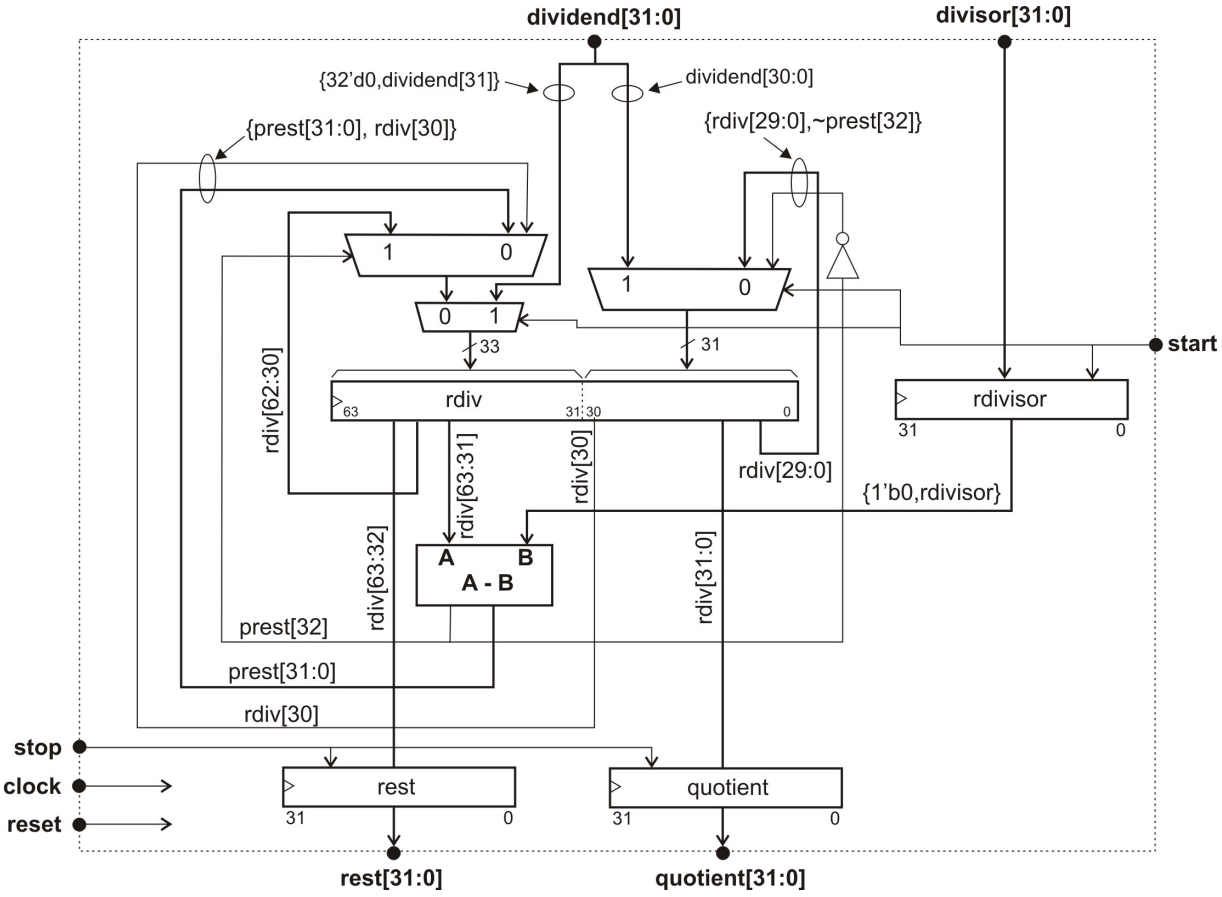

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

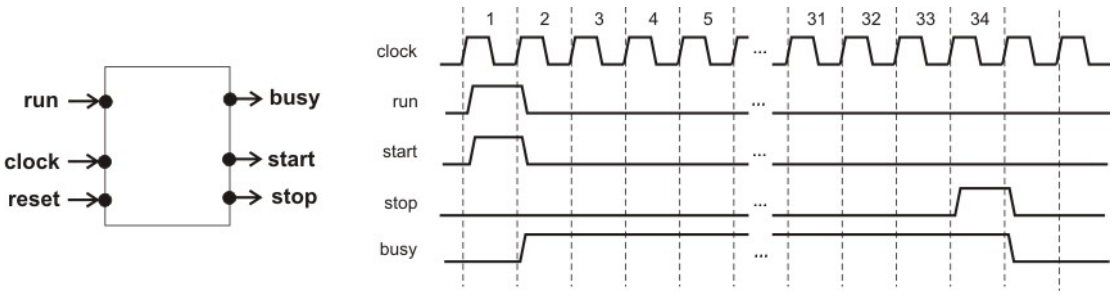

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

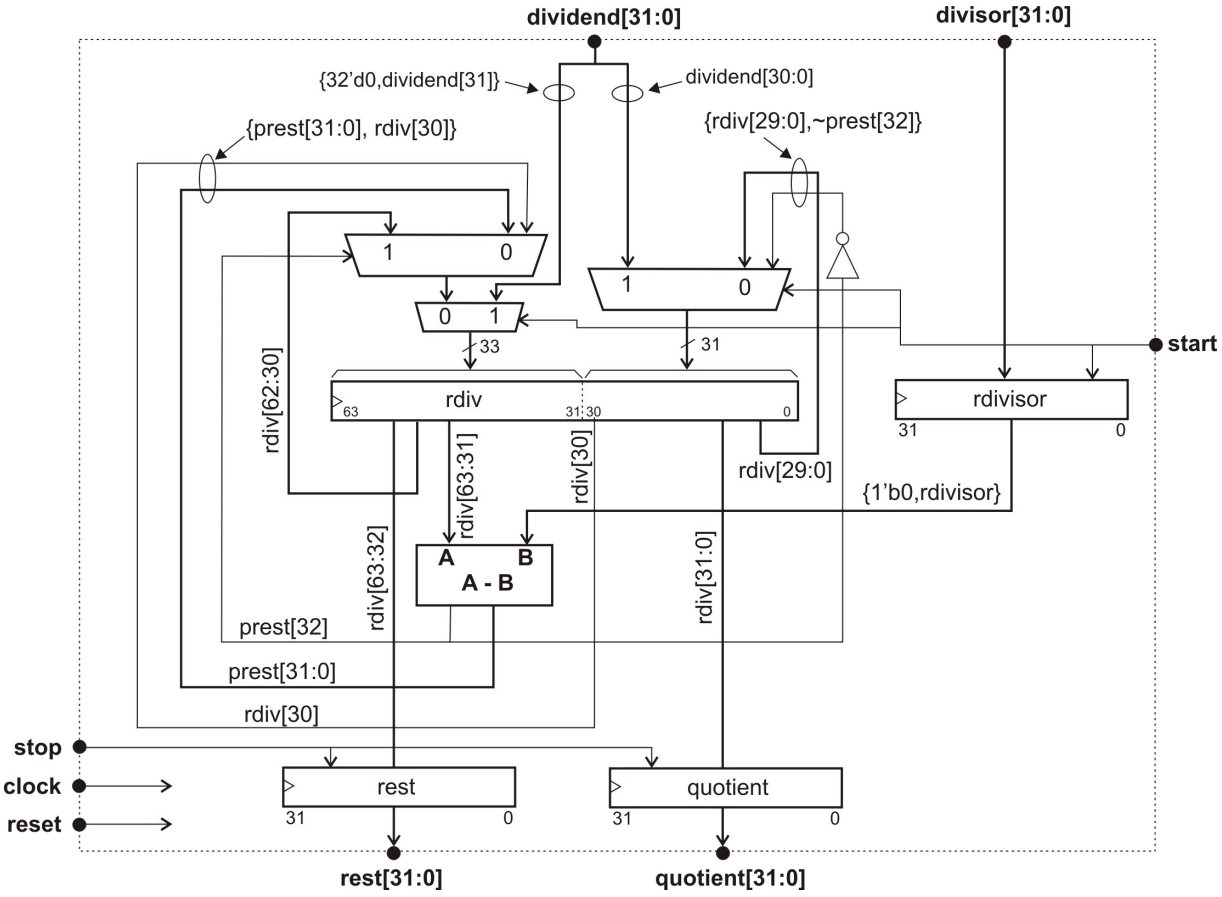

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

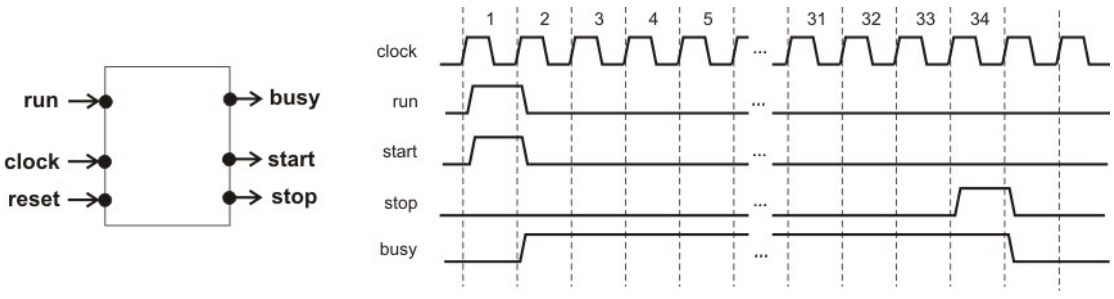

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

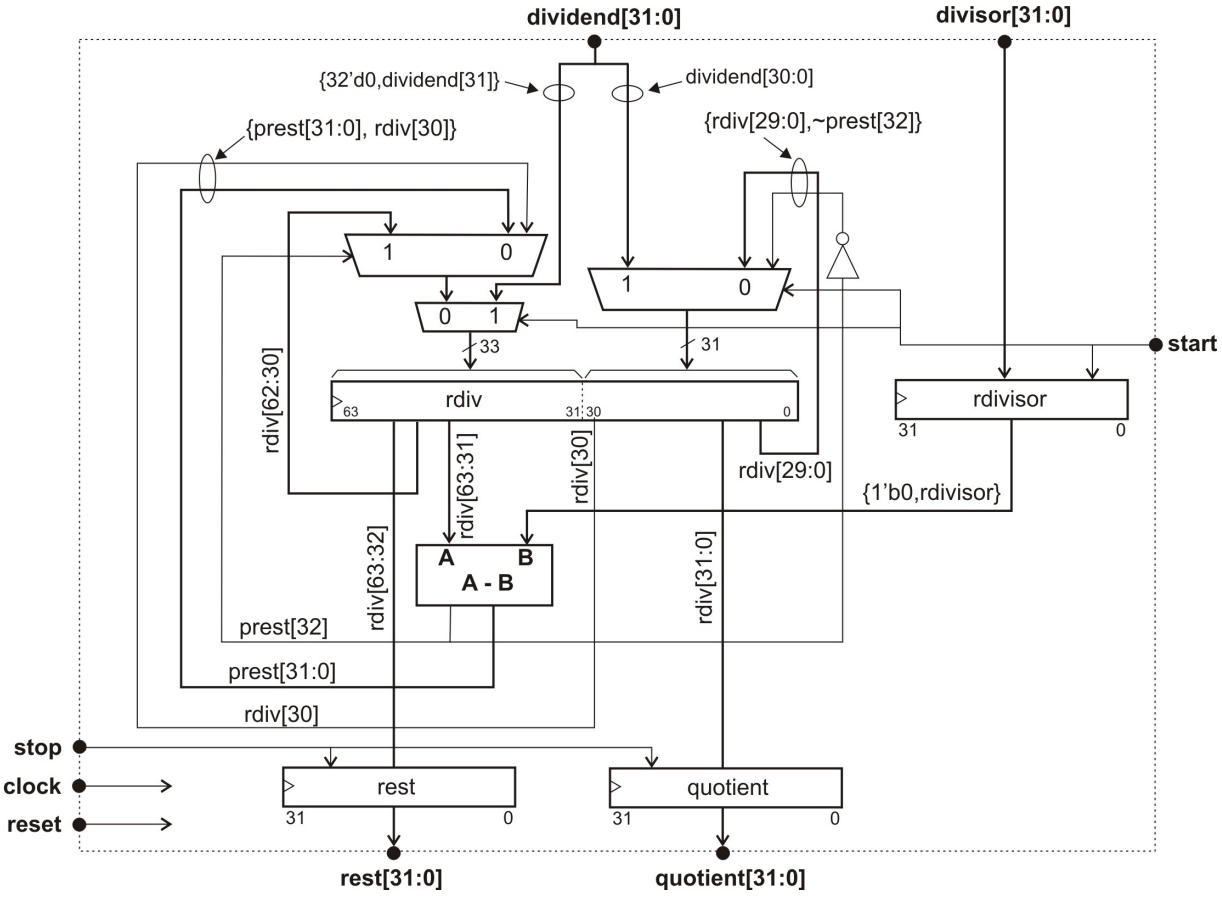

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

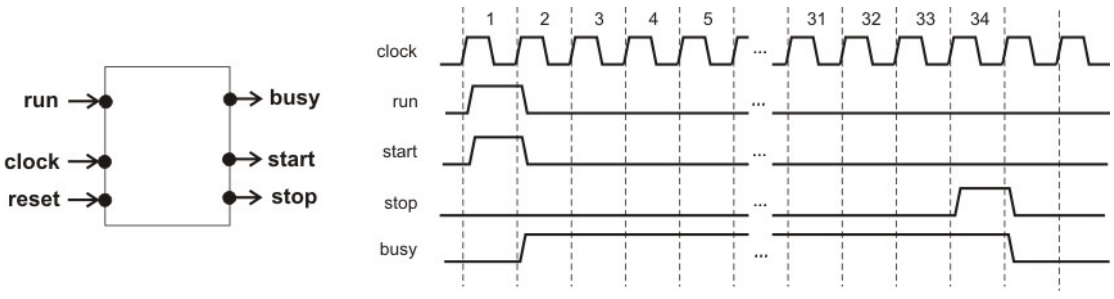

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

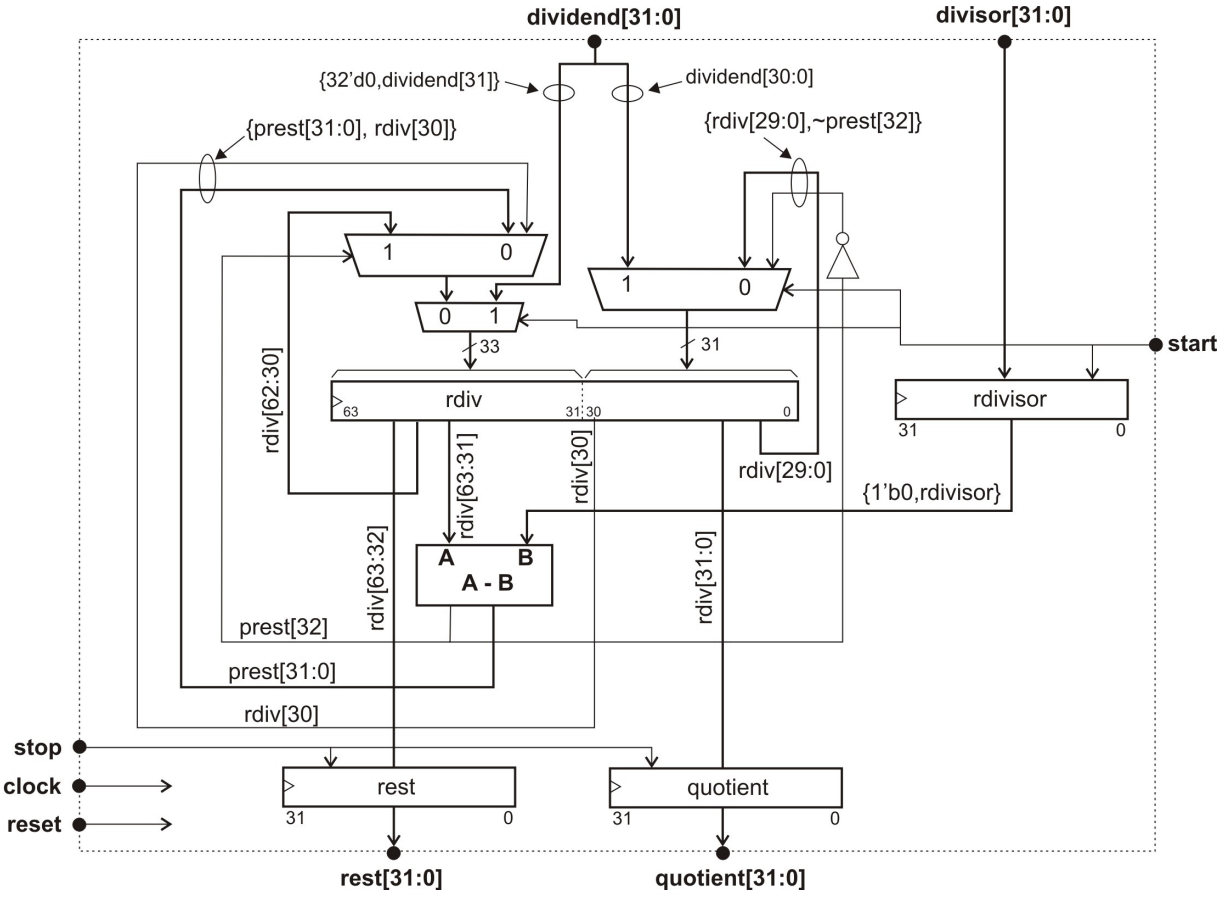

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

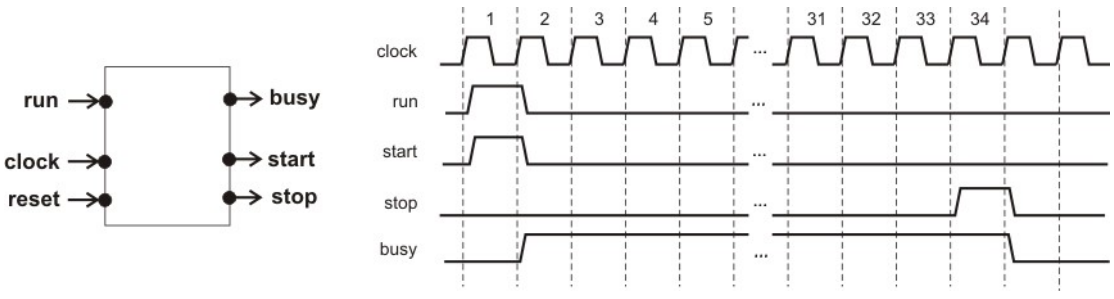

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

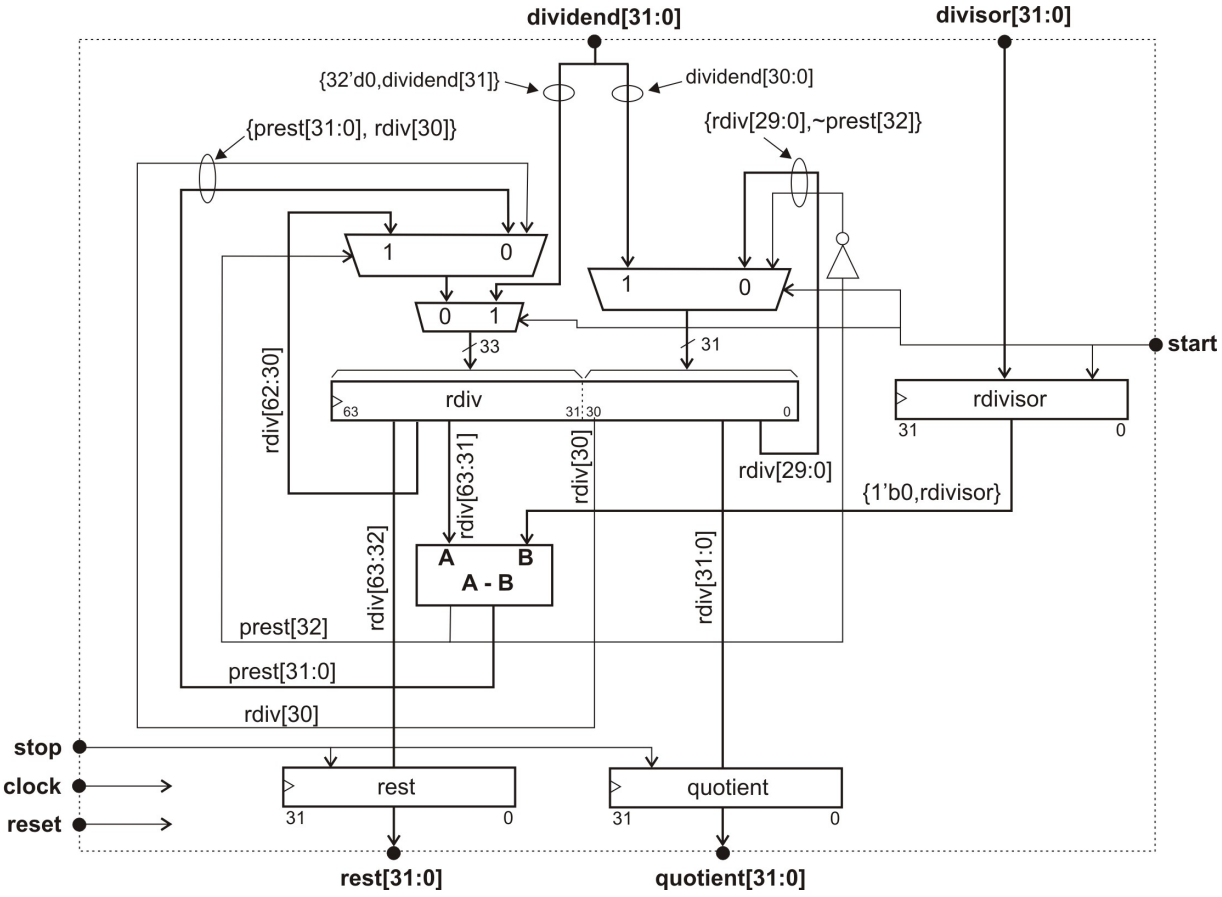

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

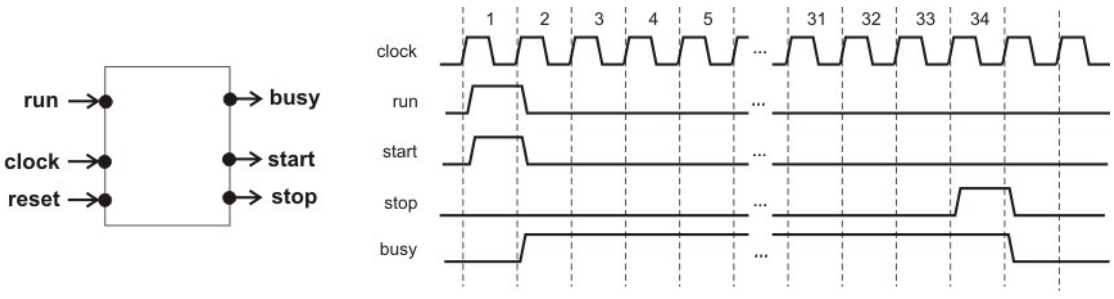

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

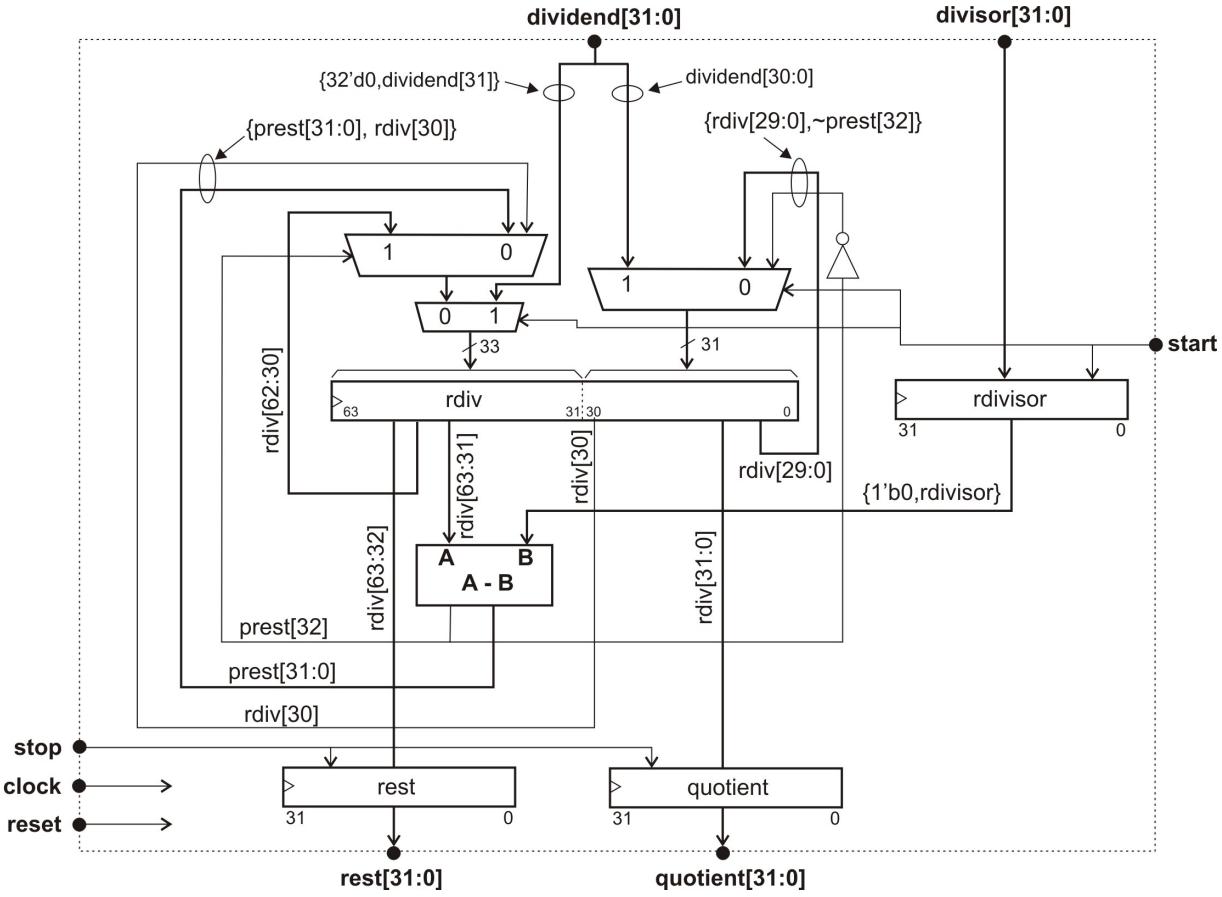

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

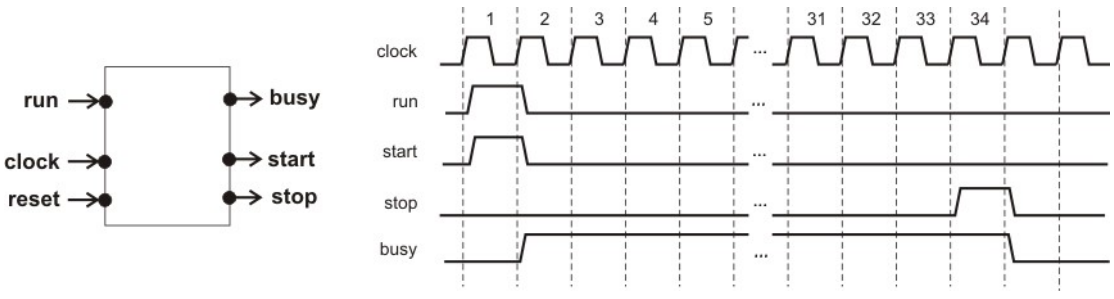

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

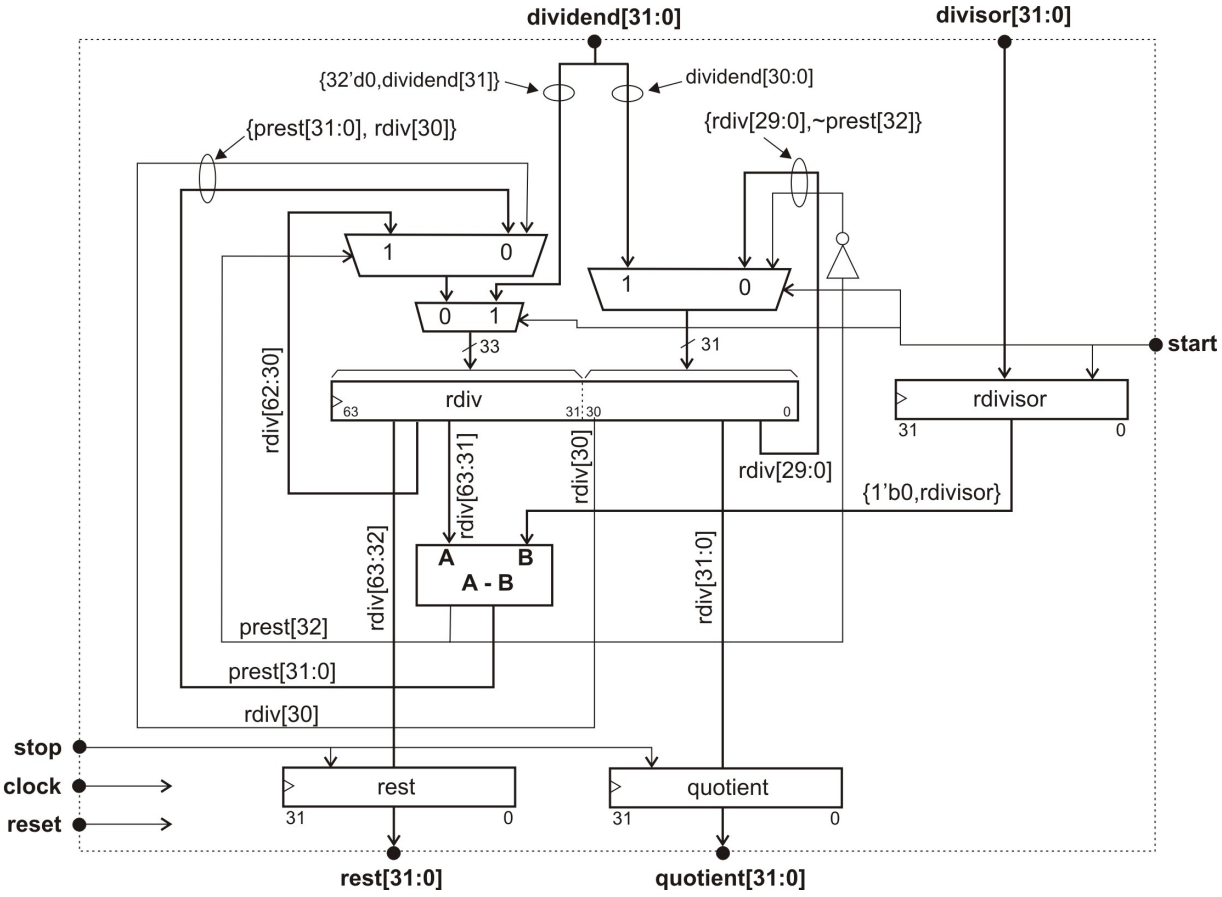

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

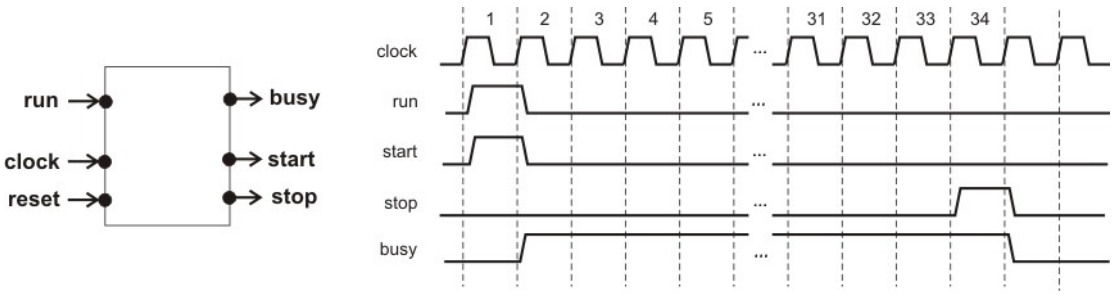

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

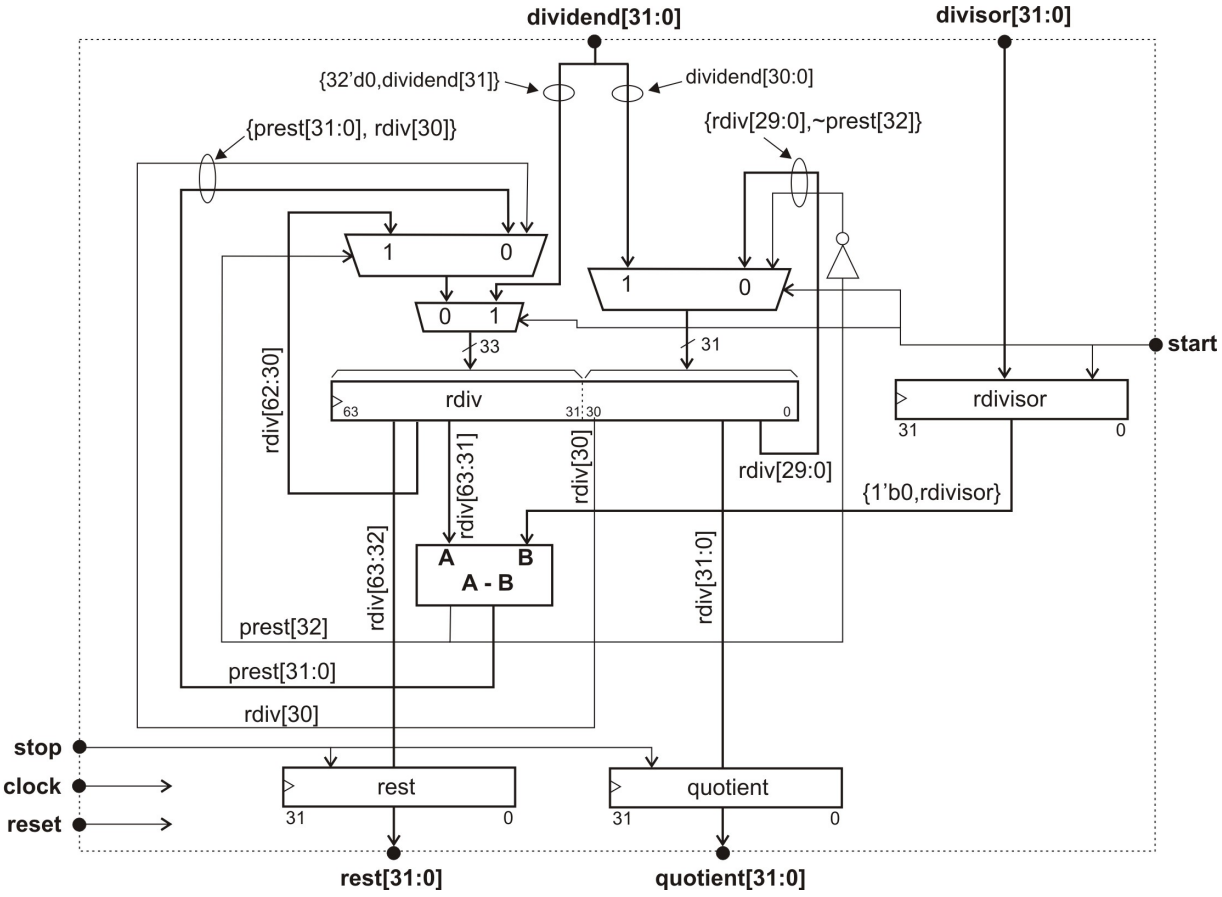

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

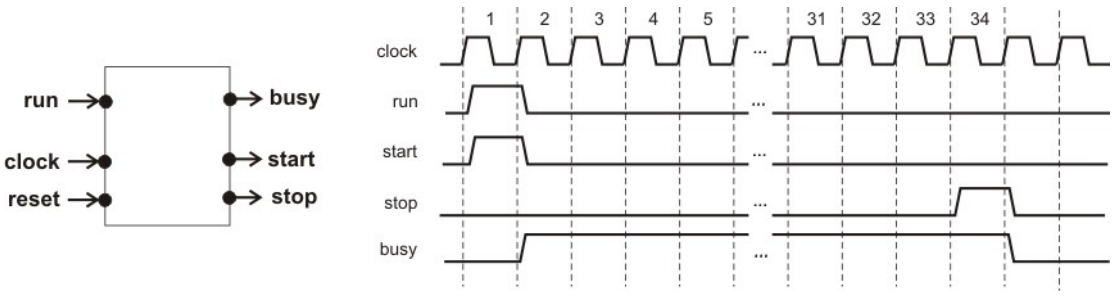

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

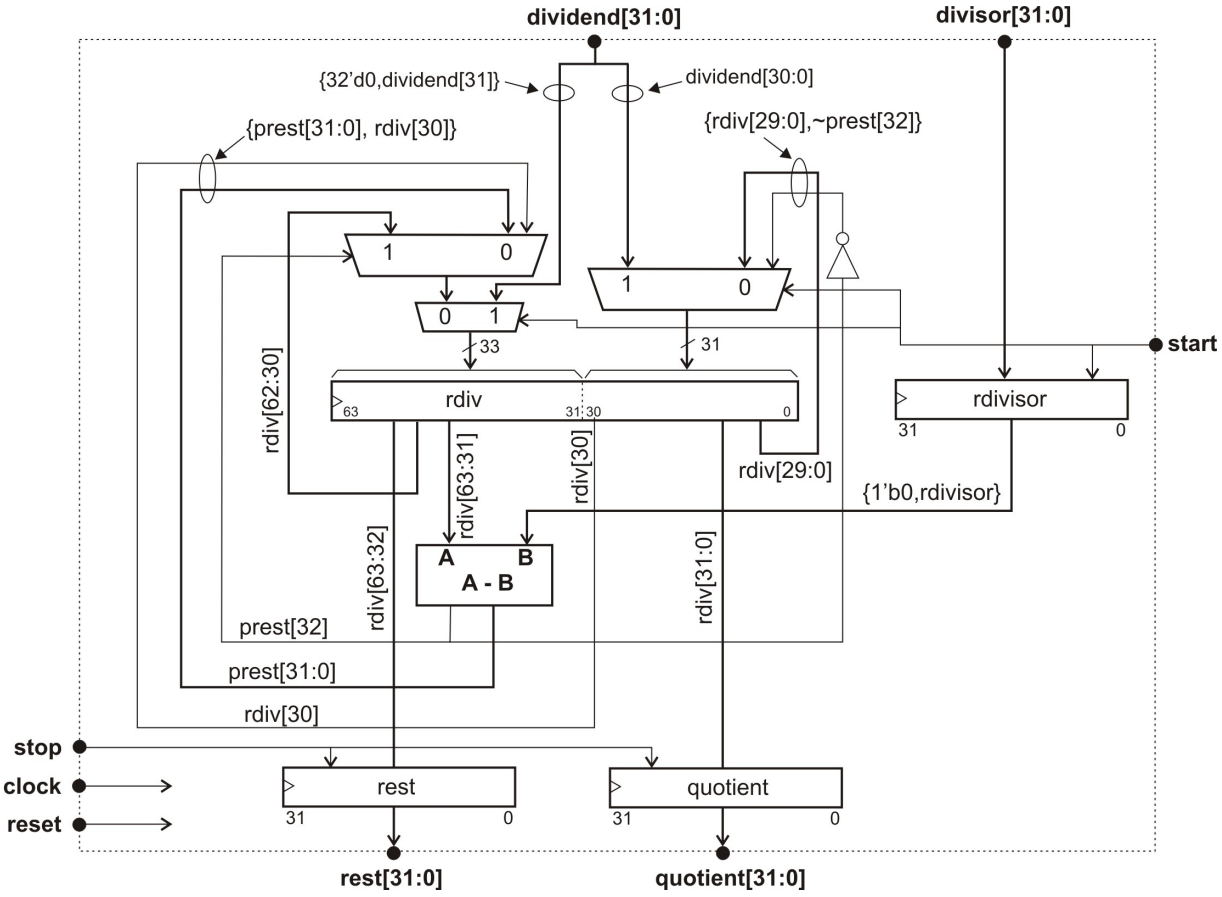

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

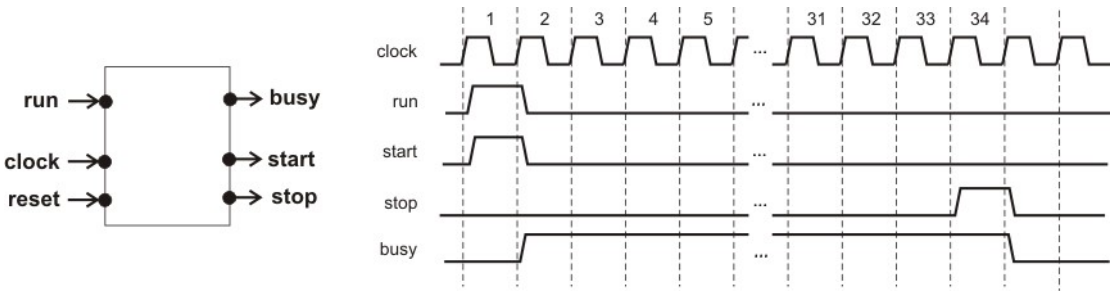

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

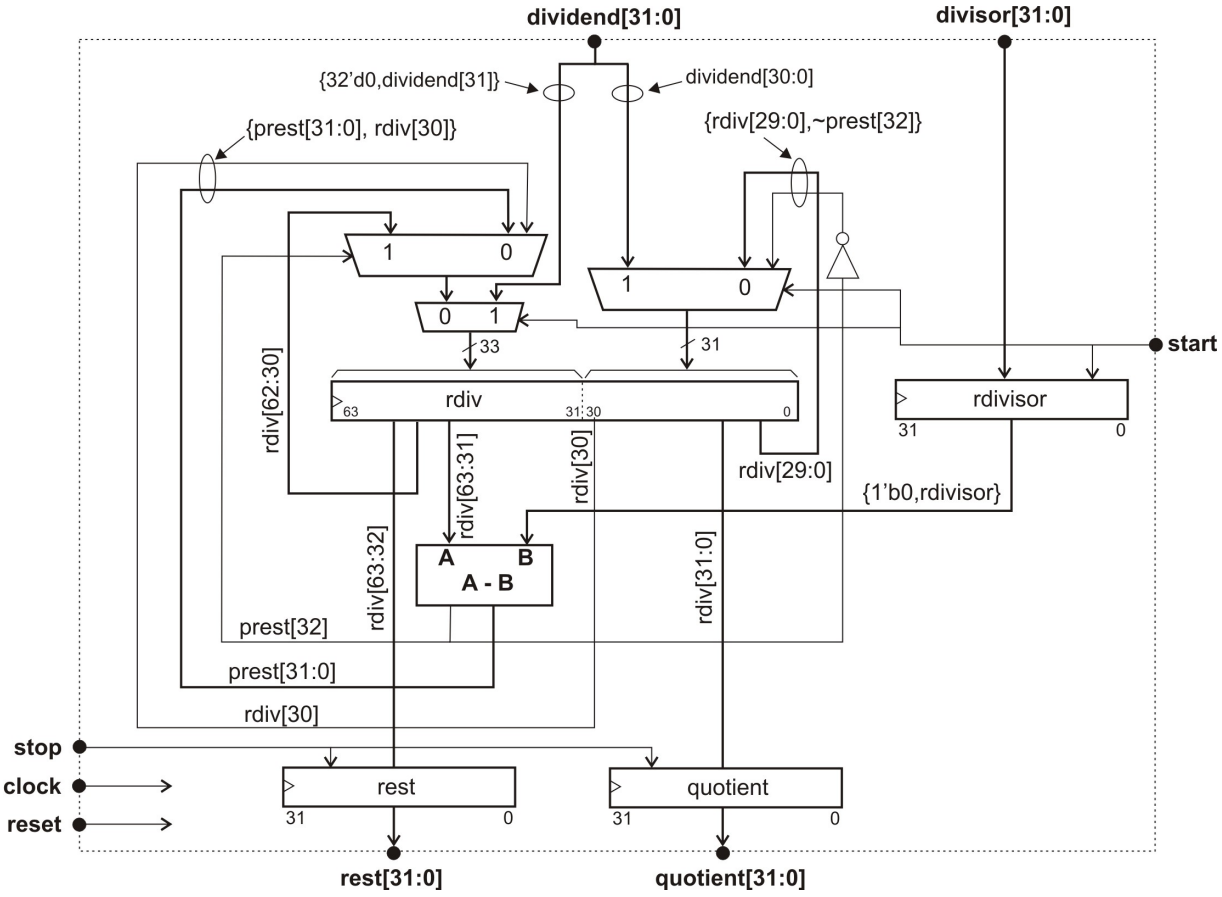

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

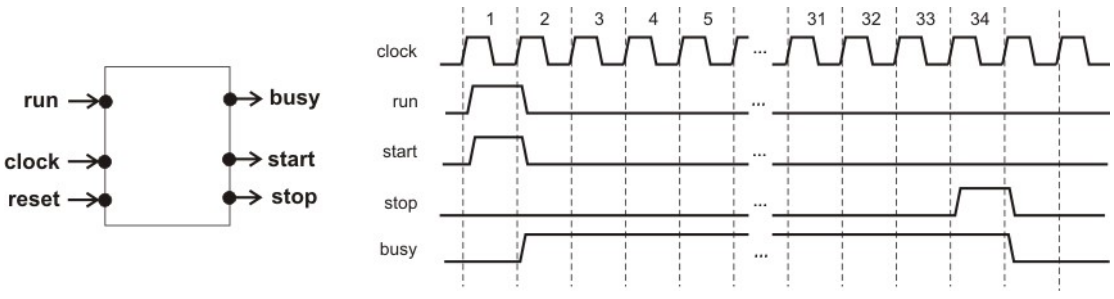

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

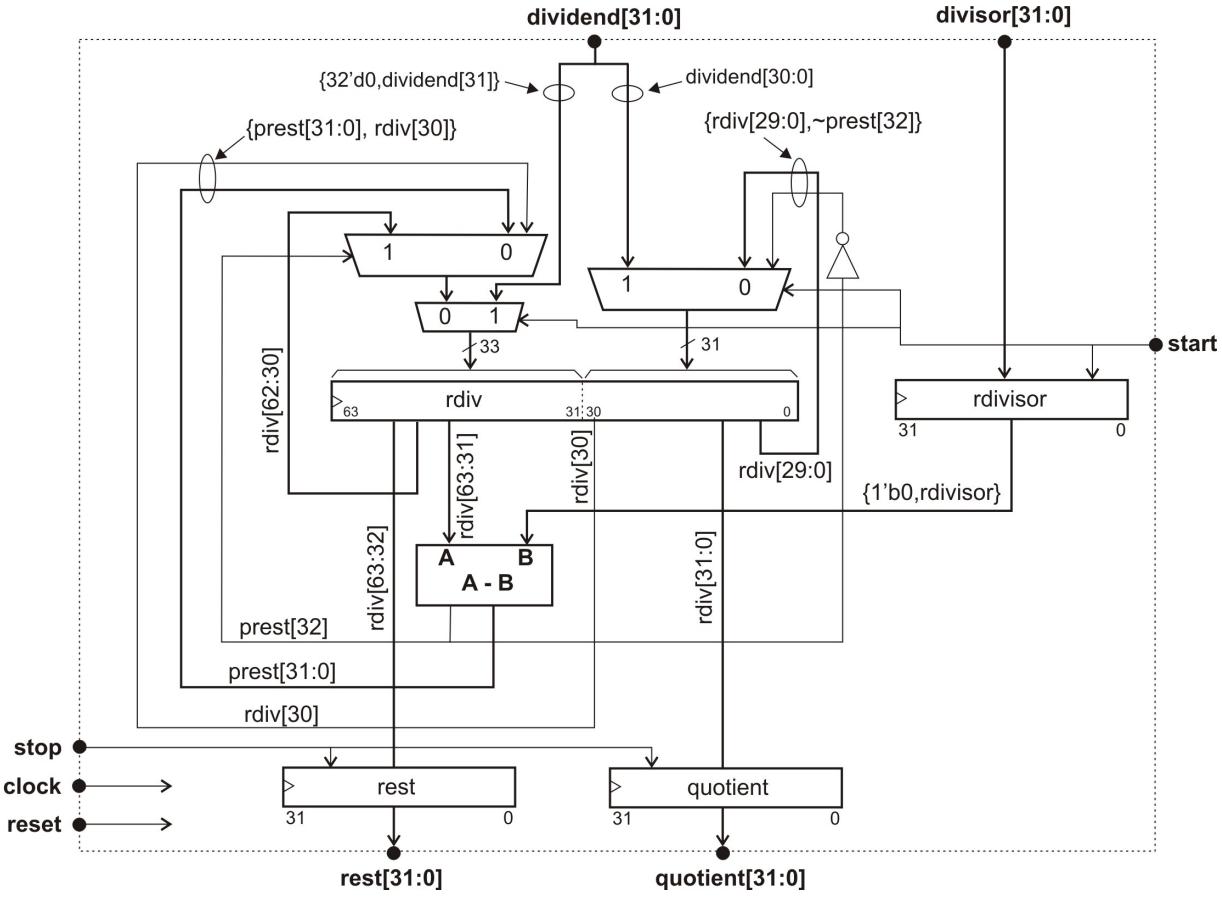

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

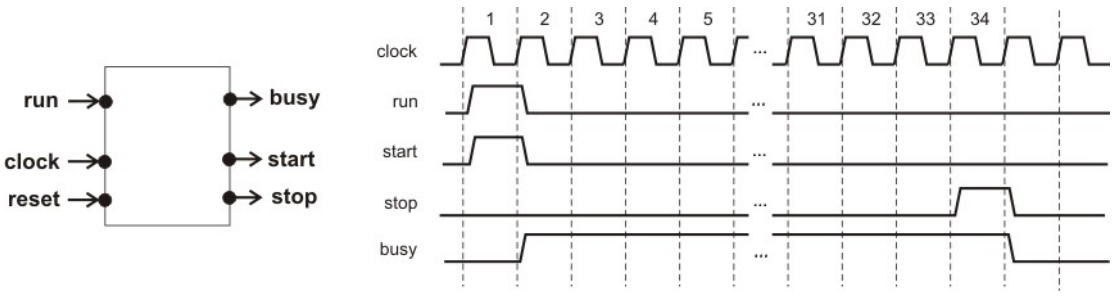

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

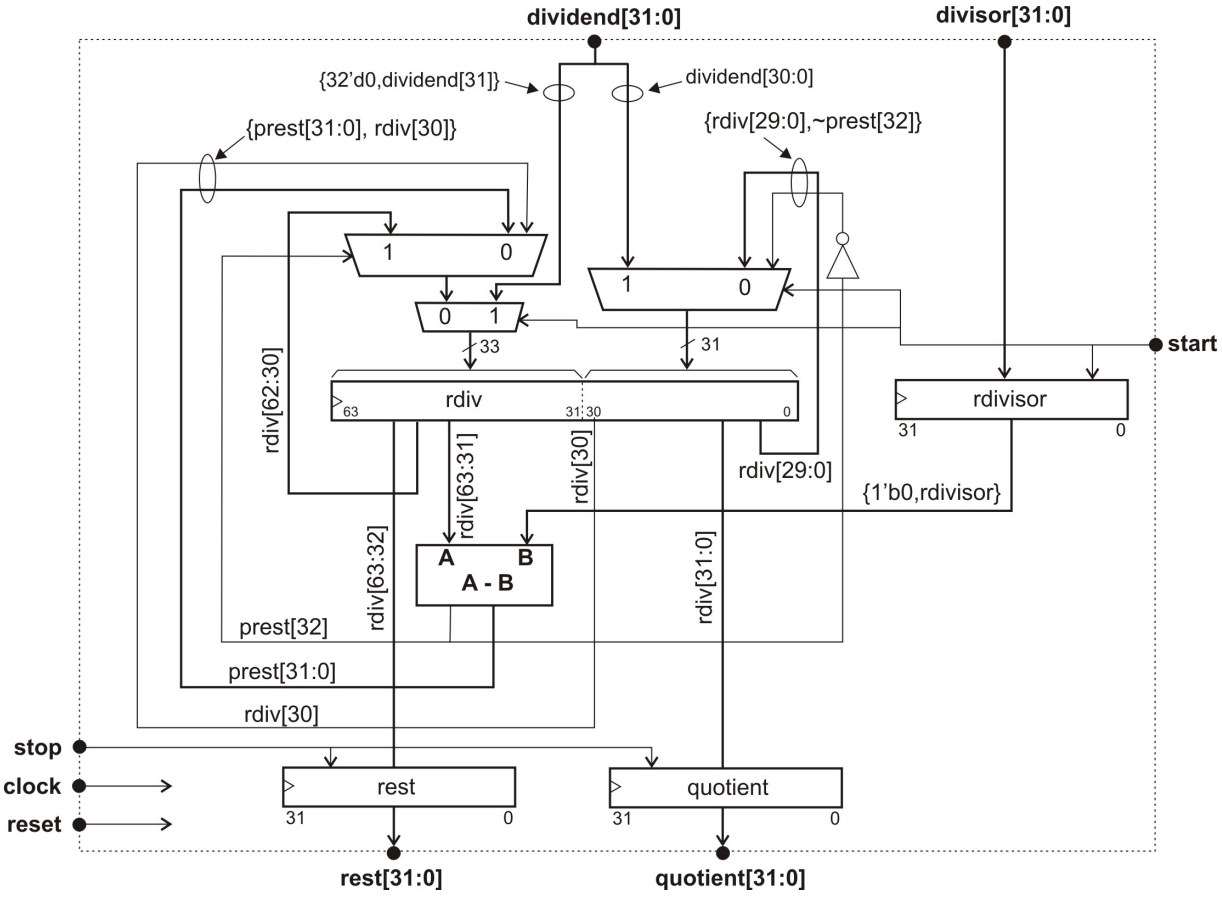

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

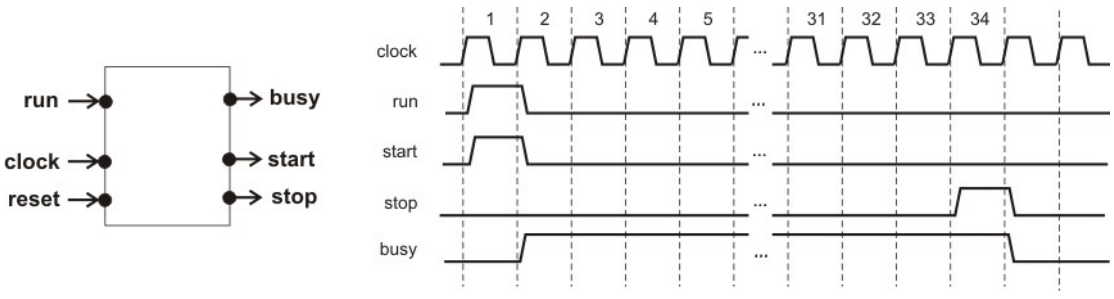

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

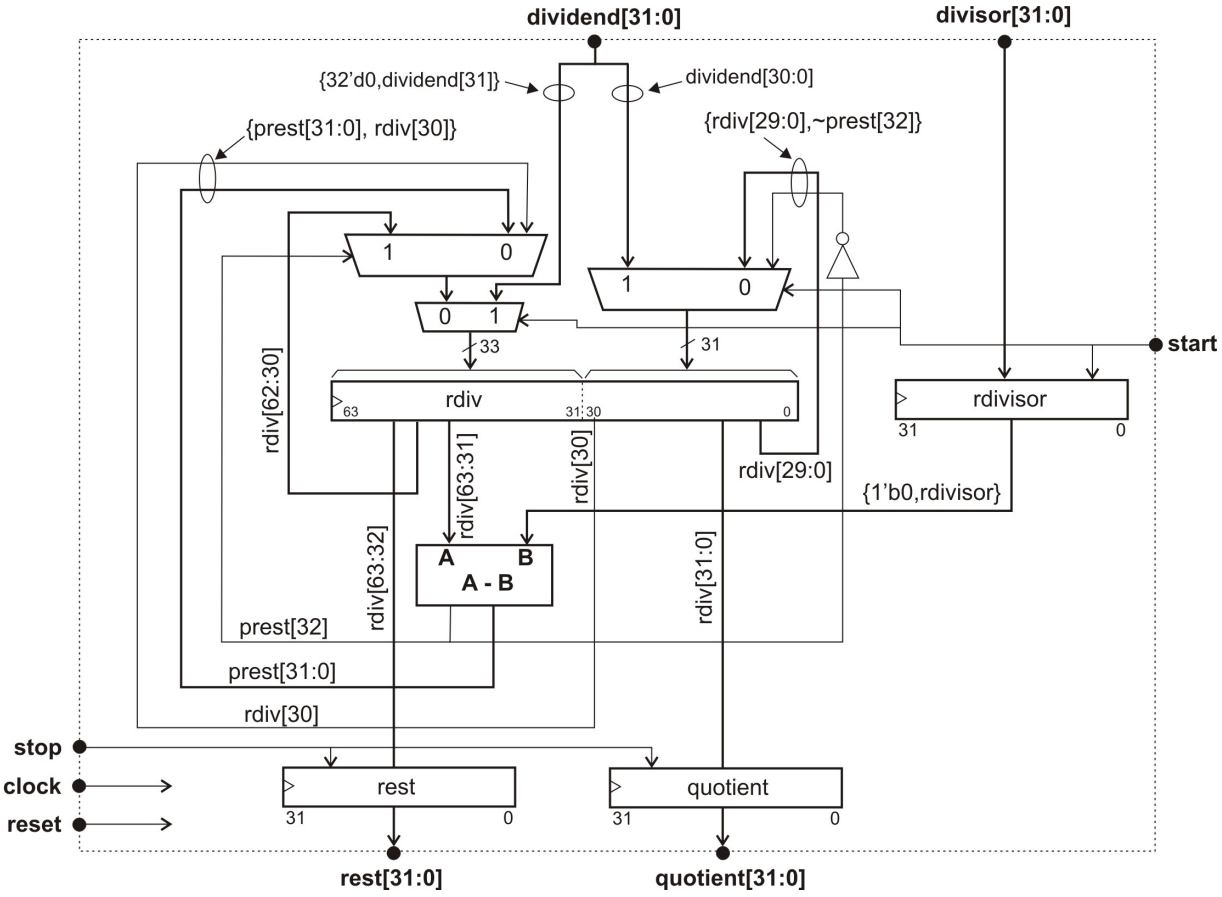

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

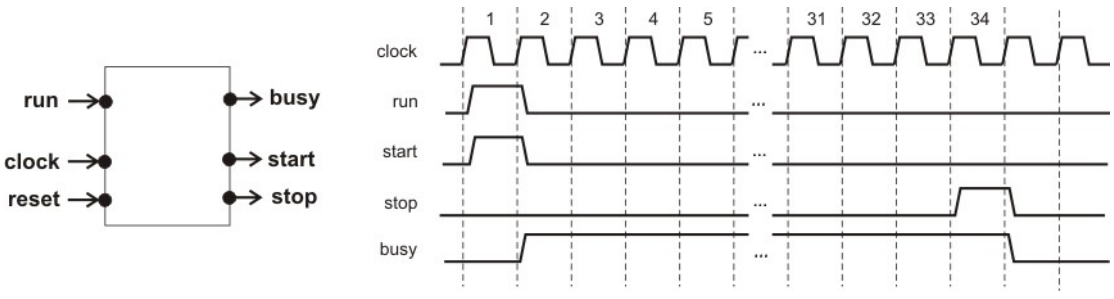

Figure 2 – Timing diagram to be implemented by the sequential controller.

EEC0055 - Digital Systems Design

2022/2023

Laboratory project 1 - V1.0

11 October 2022

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 [31:0]   quotient,   // quotient
            output [31:0]   rest        // rest
             );
```

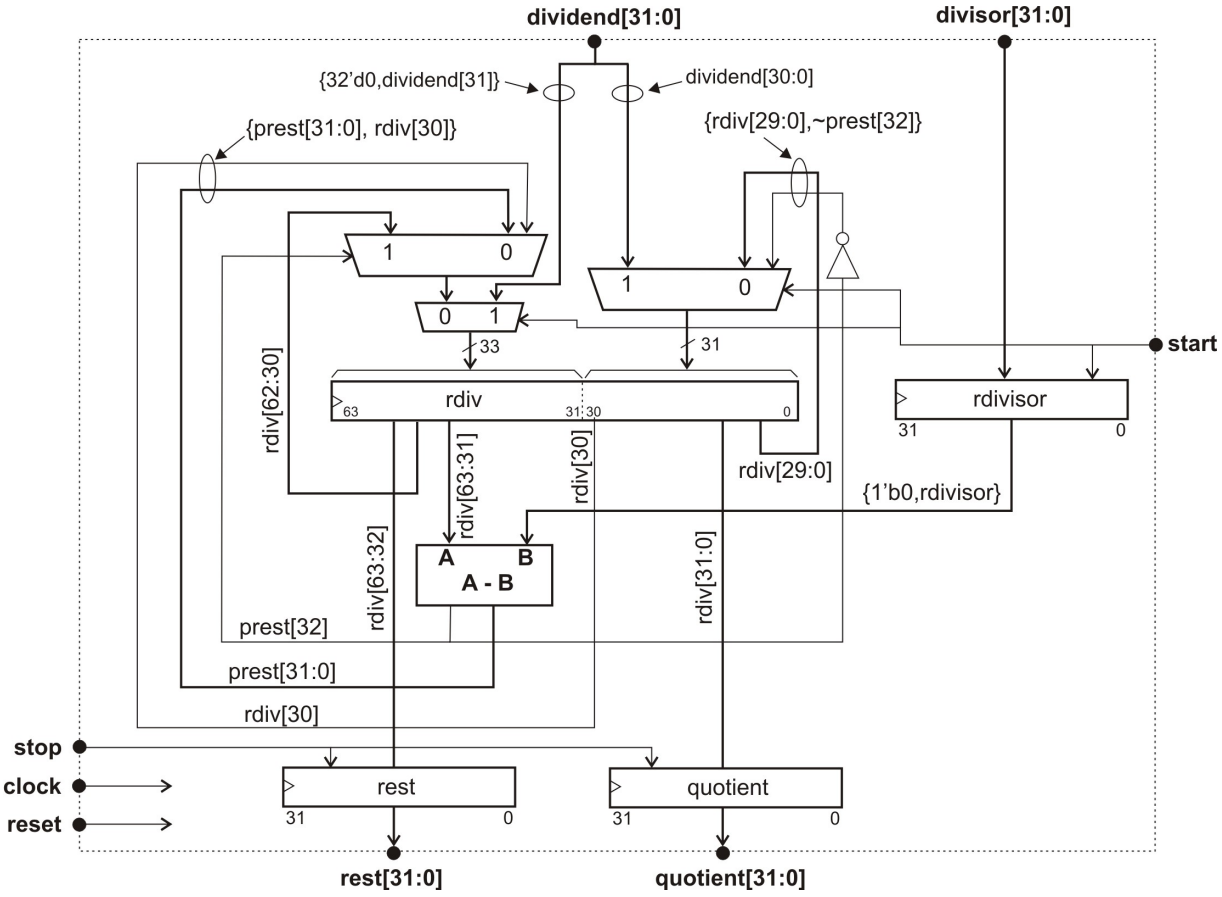

Figure 1 – RTL diagram of the sequential divider.

jca@fe.up.pt

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 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 code below, where all variables are 32-bit unsigned integers:

```
quotient = dividend / divisor;
rest = dividend % divisor;
```

The divider uses a 64-bit register (**rdiv**) that is initially loaded with 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** is 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.

3. Implementation [40%]

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**. 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 [40%]

To verify this model you must adapt the testbench included in the project archive. The task **execdivide** already included in the testbench implements the correct sequence of signals **start** and **stop** to perform a division. Improve the testbench in order to automate the verification procedure.

5. Additional developments [20%]

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 will 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.

The example below shows how to define a module with parameters and its instantiation with and without overriding the parameter default values:

```
module adder #( parameter NBITS = 32 )
              ( input [NBITS-1:0] a,
                input [NBITS-1:0] b,
                output[NBITS-1:0] c
               );

assign c = a + b;

endmodule
```

Instantiation of 3 adders with different number of bits:

```
adder adder_1 (.a(x1), .b(y1), .c(z1) ); // 32-bit adder (default NBITS)
adder #( .NBITS(4) ) adder_2 (.a(x2), .b(y2), .c(z2) ); // 4-bit adder
adder #( .NBITS(8) ) adder_3 (.a(x3), .b(y3), .c(z3) ); // 8-bit adder
```

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 2 shows the timing diagram of these signals, considering a 32-bit dividend.

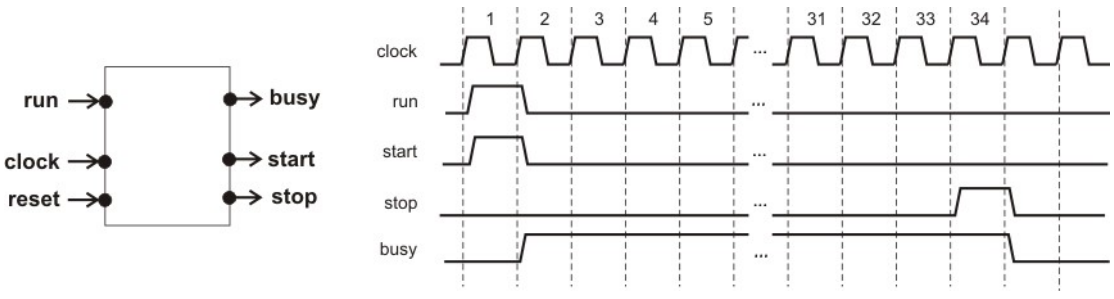

Figure 2 – Timing diagram to be implemented by the sequential controller.