M.EEC041 - Digital Systems Design

2023/2024

Assessment project 2 - V1.2

13 November 2023

TO BE CONCLUDED UNTIL THE LAST PL CLASS (week of Dec 11th)

Revision history

Date	Notes	Author
Nov 13, 2023	First release V1.0	jca@fe.up.pt
Nov 30, 2023	New section 4 ("Building a testbench…", new text in blue), previous section 4 renumbered to 5	jca@fe.up.pt
Dec 4, 2023	Figure 3 updated and text adjusted accordingly: the sequential multiplier and divisor require the run input to be set for exactly one clock cycle and the duration of these operations is reduced by one clock period (17 clocks for the multiplication, 33 clocks for the division)	jca@fe.up.pt

1. Introduction
This project consists in implementing a digital system for calculating the division of two complex numbers. That module must be integrated with a project for the ATLYS FPGA development board, using the same low speed serial interface utilized in previous projects.

2. Functional description
The circuit should calculate the arithmetic division of two complex numbers given in cartesian format (real and imaginary part). The real and imaginary parts of the two operands are represented as 16 bit fixed-point signed numbers (two's complement), with 8 bits for the integer part and 8 bits for the fractional part. The real and imaginary part of the result are represented as 32-bit fixed-point signed numbers, with 16 bits representing the fractional part and 16 bits representing the integer part.
The calculation of the complex division can be represented by the following elementary operations:

Operands are two complex number given in Cartesian format (real and imaginary parts):

$$A = Re_A + jIm_A$$
$$B = Re_B + jIm_B$$

The result to compute is the division A/B:

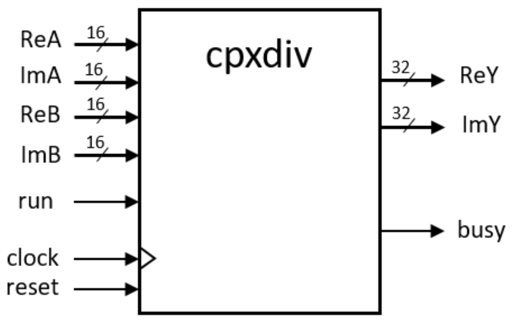
$$\frac{A}{B} = Re_{AdivB} + jIm_{AdivB}$$

Where the real and imaginary parts (Re_{AdivB} and Im_{AdivB}, respectively) are calculated as:

$$Re_{AdivB} = (Re_A \times Re_B + Im_A \times Im_B)/({Re_B}^2 + {Im_B}^2)$$
$$Im_{AdivB} = (Re_B \times Im_A - Re_A \times Im_B)/({Re_B}^2 + {Im_B}^2)$$

All multiplications receive two 16-bit operands representing 8 integer bits and 8 fractional bits, and generate a 32-bit result with 16 integer bits and 16 fractional bits. The divisions are performed between a 32-bit dividend and only the most significant 16 bits of the divisor (the integer part of $({Re_B}^2 + {Im_B}^2)$), generating a 32-bit quotient where the high 16 bits represent the integer part and the low 16 bits are the fractional part. The rest of the integer division is not used in this implementation.

3. Implementation
The circuit must be designed as a clocked synchronous sequential system using a single clock signal, active on the rising edge. All registers must have a synchronous reset signal that loads the registers with zero. The circuit must have only two control inputs, having a function similar to the control signals used by the sequential divider: setting the input **start** to 1 for one clock cycle starts the division process and the output **busy** is high while the circuit is performing the division (setting **start** while **busy** is high should be ignored). Figure 1 shows the toplevel interface and the timing diagram with the winput and output signals.



```
module cpxdiv(
        input        clock,
        input        reset,
        input        run,
        input [15:0] ReA,
        input [15:0] ImA,
        input [15:0] ReB,
        input [15:0] ImB,
        output[31:0] ReY,
        output[31:0] ImY,
        output       busy
           );
```

Figure 1 – Toplevel interface of the complex number divider.

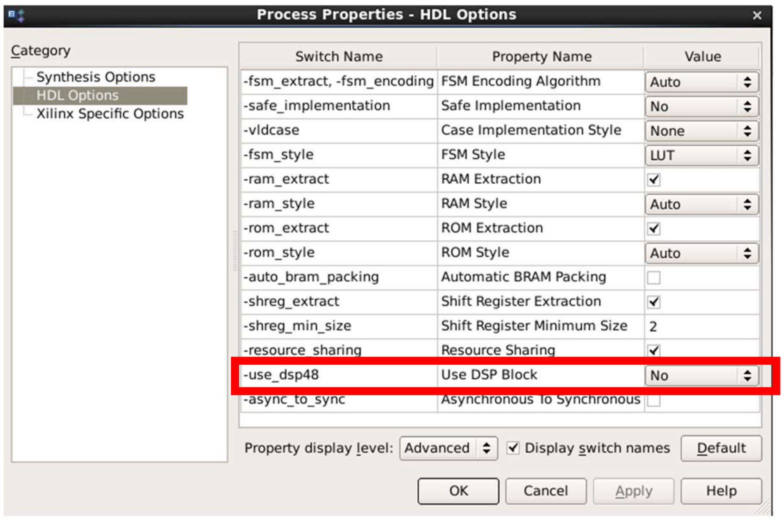
This circuit may be implemented by synthesizing the expressions above. However, such implementation results into an extremely large circuit that is not an acceptable solution. For example, synthesizing the following Verilog code results into a circuit that uses 6 combinational multipliers and two combinational dividers. Implementing this for the Spartan6 FPGA results in 5780 LUTs and 315 flip-flops, and a maximum clock of 7.7 MHz (130 ns period).

```
always @(posedge clock)
begin
  ar <= a; br <= b; cr <= c; dr <= d;
  yrealr <= ( ar*cr + br*dr ) / ( cr*cr + dr*dr );
  yimagr <= ( cr*br - ar*dr ) / ( cr*cr + dr*dr );
end
```

The solution to build should perform the arithmetic operations by re-using along time the most complex operators (the multipliers and dividers). For example, using only one multiplier and one divider the calculation could be done in 8 "stages": six "stages" for calculating the 6 cross multiplications and then two more for the divisions. Is fully combinational operators are used, each "stage" may be a single clock cycle which duration is constrained by the slowest operator (the combinational divider). If one of the operators is built as a sequential implementation, than the corresponding "stage" will last for the number of clock cycles needed to complete that operation. Between the trivial, but unacceptable, solution given using 6 combinational multipliers and two combinational dividers, and the smallest, but slow, solution using only one sequential multiplier and one sequential divider, there are many other alternatives that should be considered.

As discussed above, the arithmetic operators can be implemented by synthesizing the corresponding Verilog operators, resulting in fully combinational circuits. Although this is the normal way to implement the additions and subtractions, the multiplication and the division are much more complex operations that may be built with sequential implementations, trading area for time.

To force the no utilization of the DSP48 blocks that exist in the Spartan6 FPGA, the flag "**-use_dsp48**" in the Synthesis options must be set to "**No**" (see figure 2).



Switch Name	Property Name	Value
-fsm_extract, -fsm_encoding	FSM Encoding Algorithm	Auto
-safe_implementation	Safe Implementation	No
-vldcase	Case Implementation Style	None
-fsm_style	FSM Style	LUT
-ram_extract	RAM Extraction	✓
-ram_style	RAM Style	Auto
-rom_extract	ROM Extraction	✓
-rom_style	ROM Style	Auto
-auto_bram_packing	Automatic BRAM Packing	☐
-shreg_extract	Shift Register Extraction	✓
-shreg_min_size	Shift Register Minimum Size	2
-resource_sharing	Resource Sharing	✓
-use_dsp48	Use DSP Block	No
-async_to_sync	Asynchronous To Synchronous	☐

Property display level: Advanced ✓ Display switch names

Figure 2 – Before running the RTL synthesis, set the flag "**-use_dsp48**" to "**No**".

To help building the system, two implementation of a sequential divider and a sequential multiplier are provided and can be used freely in your design. These implementations are based on the exercises done in the lab classes and were adapted for handling signed operands with the required number of bits. The sequential divider receives a 32-bit signed dividend, a 16 bit unsigned divisor and generates a 32-bit signed quotient. The sequential multiplier receives two 16-bit signed operands and generates a 32-bit signed result.

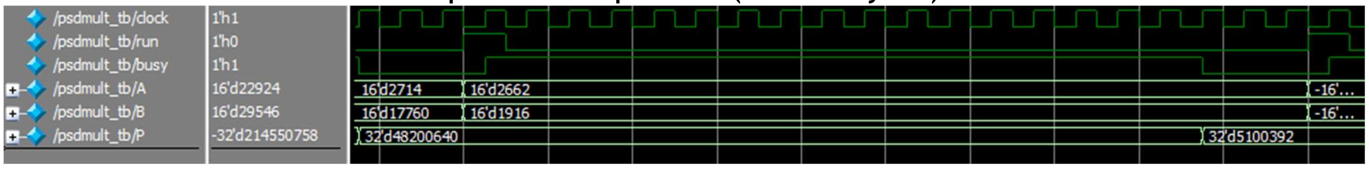
The following table presents the main speed-area characteristics obtained after RTL synthesis in XILINX ISE, using the optimization goal "speed high". Note that the maximum clock frequency reported for these implementations was obtained after synthesis and should be used only as a rough estimate for the maximum clock frequency, and for the relative comparison between these four different implementations.

Operator	**Number of LUTs**	**Number of FFs**	**Max clock frequency**
Sequential divider	207	120	257 MHz
Combinational divider	1865	N.A.	7.9 MHz
Sequential multiplier	180	88	284 MHz
Combinational multiplier	397	N.A.	131 MHz

Table 1 – Synthesis results for the sequential and combinational multipliers and dividers.

Both the sequential multiplier and the divider have a similar set of two control signals: setting to high the input **run** starts the operation in the first clock transition with run set to high. When the operation is running the output busy is set to high and returns to low when the operation concludes. The results are loaded to the outputs in the clock transition when busy goes low. Thus, the external circuit using these operators should set the input run to high for exactly one clock cycle and read the results when busy returns to low (see the timing diagrams in figure 3). The sequential multiplication completes in 17 clocks cycles and the division in 33 clock cycles.

Sequential multiplication (17 clock cycles)



Sequential division (33 clock cycles)

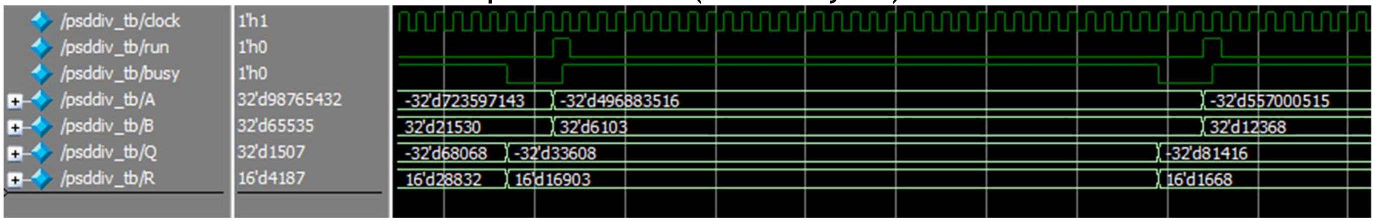


Figure 3 – Timing diagrams for the sequential implementations of the multiplier and divider.

The source code of these operators is available in the project design kit in folder ./src/IP_verilog/rtl:

psddivide_top.v	top-level module of the sequential divider
psddivide.v	datapath
psddivide_ctrl.v	controller
psdmult_top.v	top-level module of the sequential multiplier
psdmult.v	datapath
psdmult_ctrl.v	controller

Two basic testbenches are provided for these modules in folder ./src/IP_verilog/testbench:

psddiv_tb.v	testbench of the sequential divider
psdmult_tb.v	testbench of the sequential multiplier

4. Building a testbench for cpxdiv

The updated version of the project design kit contains a basic set of simulation models for performing the functional verification of cpxdiv:

./src/verilog/testbench/cpxdiv_tb.v is the top level testbench that instantiates a dummy version of the complex divider **cpxdiv_dummy.v**

./src/verilog/testbench/cpxdiv_dummy.v is a simulation model of the complex divider that generates the expected correct results and uses the same run and busy control signals to start the operation and to report the status of the module.

The testbench **cpxdiv_tb.v** includes a set of Verilog tasks for computing the expected results of the complex divider. These tasks receive the four operands (real and imaginary parts of the dividend and divisor) and output the real and imaginary parts of the complex quotient:
golden_cpxdiv_float (lines 284-298): receives 4 input float operands and generates 2 float results; this calculates the complex division with full floating point precision, implementing the expressions referred in section 2.

golden_cpxdiv_fxpoint (lines 301-357): receives four 16-bit inputs representing the input operands in signed fixed-point format (8 integer bits and 8 fractional bits) and outputs two 32-bit results also in signed fixed-point format (16 integer bits ad 16 fractional bits), as specified in section 2.

This task implements the two expressions given in section 2 using fixed-point arithmetic and calling two additional tasks (**fxpdiv()** and **fxpmult()**) that execute the fixed-point division and multiplication as the two sequential modules given in **./src/IP_verilog/rtl** (modules **psddivide_top.v** and **psdmult_top.v**). If the global variables **FULL_DEBUG** and **PRINT_RESULTS** are set to 1, this task also prints the intermediate results after each elementary arithmetic operation.

golden_cpxdiv (lines 208-280): receives 4 input float operands and generates 2 float results and calls the previous tasks to calculate the expected results (i) using the float input variables and floating point arithmetic (calling the task **golden_cpxdiv_float**), (ii) using fixed-point input data (converted from the floating point inputs) and floating point arithmetic and (iii) using fixed-point input data and fixed-point arithmetic (calling task **golden_cpxdiv_fxpoint**). Note that this task is not necessary for building your testbench and has been implemented just to compare the fixed-point results with the floating-point "exact" results.

Setup a simulation project with just these two files, run a simulation and analyze the output text report. Open the testbench **cpxdiv_tb.v** and go to line 86 (start of the main verification program).

First section (lines 86 to 140) illustrates the process of setting the operands and calling task **golden_cpxdiv**. As said, the code within this section is not intended to verify your module but rather to check the correctness of the verification tasks and analyze the errors resulting from the limited precision of the fixed-point representation.

The second section exemplifies the sequence of statements for doing a single verification of the module **cpxdiv_dummy.v**: select the 4 float operands, convert to fixed-point, call task **golden_cpxdiv_fxpoint** to obtain the expect fixed-point results, start the **cpxdiv_dummy** module by pulsing the input **run** for at least one clock cycle, wait for **busy** to be zero, calculate the difference between the results generated by the module and the golden results and print an error message if any error was found (note the utilization of the inequality operator "**!==**" in the if statement, that also compares the "**x**" (unknown) value).

The simulation module **cpxdiv_dummy.v** implements the complex division by calling the same task used in the verification program (**golden_cpxdiv_fxpoint**). The parameter **NOCLOCKS** define the number of clock cycles that the module needs to conclude the operation. The code in lines 59-60 show how to inject an error by just toggling one output bit. This may important to check the behavior of your testbench when an error is found. Note that this is a simulation model and does not reflect in any way the intended synthesizable implementation in Verilog.

5. Design goals and evaluation

The system will be integrated in a FPGA project similar to the design used in the last training laboratory (implementation of the sequential divider in the ATLYS FPGA board). The system clock frequency can be chosen among 50 MHz, 100 MHz or 200 MHz. Note that to calculate the time required to compute one complex division you must consider the highest clock frequency that can be used in your design.

The two metrics that will evaluate the quality of your design are the time needed to perform one complex division (not the clock frequency!) and the FPGA occupancy, measured as the number of lookup tables ("Slice LUT")[1]. The design strategy should try to minimize the computation time while minimizing the FPGA resource utilization and the evaluation of the project will be penalized according to the speed-area results obtained.

The speed-area solution space is divided into 7 regions, representing different levels of quality of the design (figure 4). Each region translates to a penalization that will be applied to the project grade. Outside the regions shown in figure 4 the penalization will be 100%.

The area to consider (number of LUTs) is obtained by the synthesis of your module alone and the whole design must be able to run with the clock frequency used to calculate the computation time for the division operation.

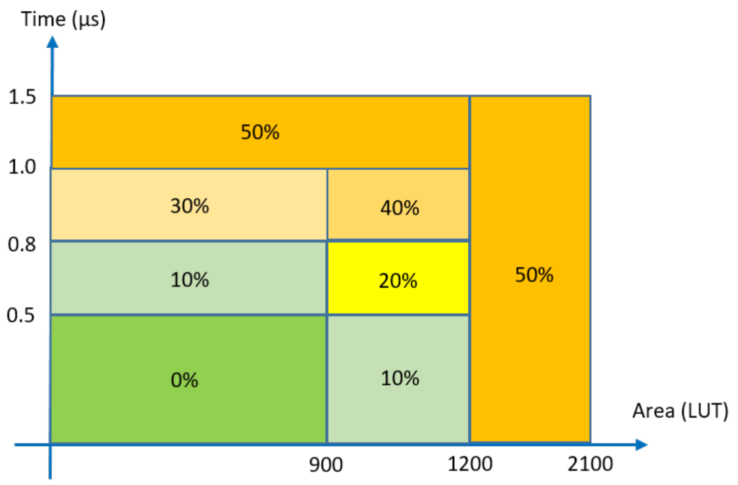


Figure 4 – Penalization levels based on the speed-area result.

The final evaluation will be based on the achievement of the 4 principal design stages:

1) Functional verification of the synthesizable RTL code: 50%
2) RTL Synthesis, optimization and post-synthesis verification: 20%
3) Integration in the FPGA reference project and verification: 20%
4) Final implementation (P&R) and demonstration in the FPGA board: 10%

Example: a team completed the final integration in the FPGA project but missed the final implementation and FPGA demonstration: will be graded 90%. If the design metrics fits in the 0% penalization region the final grade will be 90% (18/20), but if it falls into the 40% penalization the final classification will be 90% * 0.6 = 54% (10.8/20)

[1] The number of flip-flops is also important but can be estimated as a fraction of the number of LUTs