Advanced Digital Design [LU]
Lab Exercise II

Andreas Steininger & Jürgen Maier & Florian Huemer

TU Wien
December 1, 2020

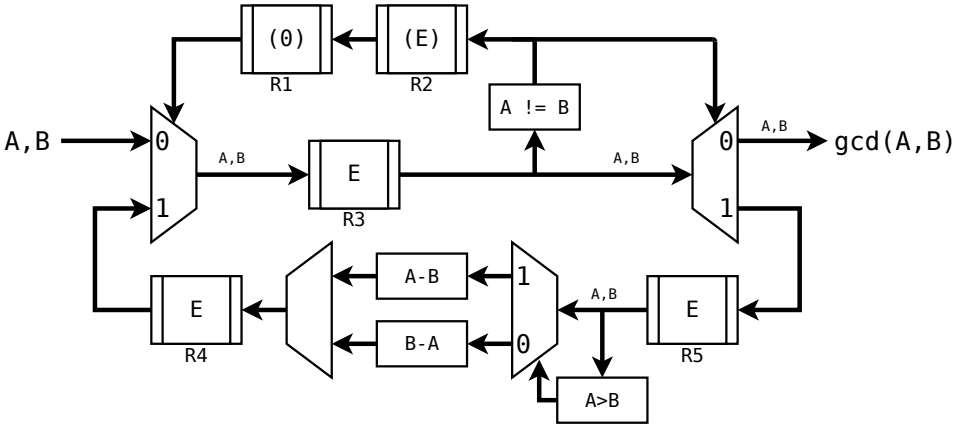

Figure 1: Dataflow representation of an algorithm that calculates the greatest common divisor. Slightly adapted from [2, p. 38].

In this exercise you will experience first hand the cumbersome, tedious and sometimes frustrating job of designing asynchronous circuits. A comparison to a synchronous design makes the much higher effort clearly visible. Although tools exist, that are able to generate code based on a higher level description (you will use one of these also in the exercise), it is crucial to implement an asynchronous design at least once manually to fully realize why they are not very popular and to clearly see the possible paths of improvement.

Part A

Data flow diagrams are an efficient way to model asynchronous circuits. Figure 1 shows a 4-phase circuit that calculates the greatest common divisor (GCD). Your first task is to retrace the data flow through the shown implementation and explain it in your presentation. Then alter the circuit such that the algorithm from Listing 1 is implemented. Show and explain your result in the presentation, especially the temporal behavior. Note that the sources of Figure 1 are available in the template (subfolder `figures`).

Listing 1: Least Common Multiple (LCM)

```
2   uint LCM (uint A, uint B){
3     uint sumA=A;
4     uint sumB=B;

6     do{

8       if (sumA < sumB)
9         sumA += A;
10      else
11        sumB += B;

13    } while (sumA != sumB)

15    return sumA;
16  }
```

Afterwards investigate the click-element template presented by Mardari et al. [1] (paper in assignment template) and concrete implementations provided in the corresponding repository[1]. Evaluate the following questions thoroughly and add the answers to your presentation:

1. What are the main differences between click-elements and a Muller pipeline?
2. Can the click-element template also be used for a four-phase communication? If not highlight the necessary changes.
3. In [1] a device is presented that can handle different phases on its in- and output. What is the difference to a regular click-element and where is such a device required?
4. In the repository two versions of some components, e.g. fork and reg_fork, are available. What is the difference? How does it change the temporal behavior?
5. Comparing the data flow representation in Figure 1 and the implementation from the click-element repository it can be seen that registers moved, got merged or were removed. Show and explain the changes!
6. Highlight the position of the data tokens for the click implementation of the GCD inside the circuit and retrace request and acknowledge signals, i.e. the phase of the data, for the first and second iteration. Which important role does the click multiplexer have?

In the next step use the code examples (single components and fully assembled designs) from the repository to implement the data flow model you developed for the algorithm in Listing 1, however, now using 2-phase communication. More specifically instantiate your design in the provided top entity (`src/lcm_top.vhd`) and make sure to consider the following hints:

[1] `https://github.com/zuzkajelcicova/Async-Click-Library`

- Debounce the inputs where necessary, which becomes important when you download your design to the FPGA.
- The interface width should be easily changable, as you need differing values for simulation and FPGA.
- Make sure that all internal signals are wide enough such that no overflows can occur.
- The delay element provided in the repository does not work with our libraries. Instead use the one from the template folder.

Since all memory elements used in click-elements are flip-flops, inferred latch warnings are prohibited. Solely the following ones are allowed:

- 12090 Entity "mux" obtained from . . .
- 19016 Clock multiplexers are found and protected
- 15714 Some pins have incomplete I/O assignments
- 332191 Clock target . . . of clock . . . is fed by another target of same clock
- 332148 Timing requirements not met

To convince Quartus that all signals that drive a flip-flop clock input are clocks use the command `derive_clocks -period <time>` in your `.sdc` file, which states that all signals that Quartus suspects to be clocks are set to the same period. If you want to have differing periods add each clock separately with the command *create_clock -name <string> -period <time> [command {string}]* whereat command can be for example *get_nets*, *get_ports* or *get_pins*[2]. Please note that in a real world application you should definitely choose the second method!

Run pre-layout simulations in QuestaSim of your solution for an input data width of eight Bits by calculating the LCM(A,B) for

$$(A, B) \in \{(64, 13), (28, 7), (33, 44), (8, 8)\}.$$

For this purpose extend the file `mac_tb.vhd` in folder `src`. In folder `modelsim` we furthermore provide a Makefile and some scripts to automize the simulation task. For example `make sim_cl` compiles everything and starts a simulation on the command line. This is useful for quick code compilations, which takes significantly longer in Quartus. To assure that newly added files are also compiled do not forget to add them to `sim.do`. Take a picture of the wave showing sufficiently many internal signals to retrace the dataflow inside the computational loop round per round.

Finally download your solution to an FPGA and verify that it is operating properly. Feel free to alter the chosen signal to board mapping, shown in Table 1, to your needs. Use SignalTap to record some iterations of the computational loop, comparable to what you did in simulations, showing clearly the internal dataflow.

[2]More information are available at the Quartus Timing Analyzer User Guide on the web which can also be reached from the help section in Quartus.

signal	board
A[3:0]	LEDR[15:12]
A[3:0]	SW[15:12]
B[3:0]	LEDR[10:7]
B[3:0]	SW[10:7]
result[7:0]	LEDG[7:0]
ack_AB	LEDR[17]
req_AB	SW[17]
ack_result	SW[0]
req_result	LEDR[0]
res_n	KEY0

Table 1: Mapping of in- and outputs to board I/O.

Part B

Develop a synchronous implementation for the same algorithm. Be aware that no timing violations are allowed to occur, independent of the applied input values! New data is indicated by a high value on input `ready` which shall be acknowledged by setting output `done` to `1` for one clock cycle when the input has been read. To indicate a valid LCM result at the output use signal `valid` (again set to `1` for one clock cycle). Verify your solution once again by simulations and by downloading it to the FPGA.

Show in your presentation simulation results for the same values as in Part A and screenshots of SignalTap. Again it should be possible to retrace the internal mode of operation. Compare the implementation effort for the synchronous and asynchronous design and add it to your presentation. Which was harder to achieve? Explain what the causes of the (maybe big) differences were.

Part C

Analyze your solution of Part A and answer the following questions: What was the most time consuming task? How could the effort be reduced? Is it necessary to map the data flow description one to one to hardware? Can one simplify the description? Justify your answers in the presentation and provide examples wherever possible.

The original intent of HDLs was to desribe the behavior of a hardware component. With this in mind search for the most simplistic implementation of the algorithm from Listing 1! Note that your design is not required to be synthesizable! The only constraint is, that the testcases from Part A are correctly solved in a simulation.

Deliberate about your solution, answer the following questions and include your responses to your presentation: Is there a benefit of describing designs in this fashion compared to Part A & B? Is your design implementation-style independent? What can you deduce when you compare the present approaches to design hardware to the original purpose of a hardware description language?

Part D

Asynchronous circuits are currently only marginally used, also due to the difficulties you experienced already. To ease the development of click-element circuits, a former student of this course developed a tool[3] that generates VHDL code based on an algorithmic level description. Take a look at the repository, use the tool to reimplement the algorithm from Listing 1 and verify the generated solution using your framework from Part A. *Hint:* The internal and output data type are derived from the input data types. Thus be sure to scale these appropriately.

Evaluate your design by (i) simulations and (ii) downloading it to the FPGA and making measurements with SignalTap. Include the results in your presentation and elaborate on (dis-)advantages compared to the manual design in Part A.

References

[1] A. Mardari, Z. Jelčicová, and J. Sparsø. Design and FPGA-implementation of Asynchronous Circuits Using Two-Phase Handshaking. In *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 9–18, May 2019.

[2] Jens Sparsø and Steve Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[3] `https://github.com/Wiede5335/go2async`