
02203 Design of Digital Systems (Fall 2025)

Assignment 1: Greatest Common Divisor

A digital circuit design assignment using VHDL or SystemVerilog,
XILIX Vivado, and a Artix-7 FPGA board.

2025 v.8.0.LP

Preparing for the assignment

Before showing up at the first lab session you are expected to:

- *have read the entire document (this is Task 0a)*
- *have downloaded and skimmed the VHDL or SystemVerilog files (this is Task 0a)*
- *have answered the questions in Tasks: 0b), 2a), and 2b)*

Once you get started, you will realize that there is quite some work to do, and you can not expect to complete it all only using the 2-3 lab sessions that we have scheduled.

Abstract

The purpose of this lab is to teach a systematic, top-down, simulation-based, design flow to be used when specifying, designing, and implementing digital systems. The design flow is based on the VHDL or SystemVerilog hardware description language, the XILINX Vivado synthesis tool, and a XILINX Artix-7 board (Nexys 4 DDR or Nexys A7 from Digilent Inc.) for implementation. The particular circuit that we consider implements Euclid's algorithm for computing the greatest common divisor of two positive integers. This algorithm should be well known, and hence we can focus on the design flow. We will provide you with a number of VHDL or SystemVerilog files (including a specification and a test-bench) and you will have to write some new VHDL or SystemVerilog files describing your design. **It is up to you to decide if you want to use VHDL or SystemVerilog (or both).** Appendix A contains a list of the files that we provide. The work you have to do is organized into a number of tasks – one for each section in the document. At the end of each section, you will find a list of questions related to that section.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 2 | Step 1: Executable specification and test environment. | 5 |
| 3 | Step 2: RT-level FSM-style implementation. | 5 |
| 4 | Step 3: Exploring design optimizations | 10 |
| 5 | (OPTIONAL) Step 4: Low-level component-based implementation. | 11 |

| | | |
|----------|-------------------------------|-----------|
| 6 | Your report | 12 |
| A | List of provided files | 12 |

1 Introduction

The particular circuit we will design should implement Euclid's algorithm for computing the greatest common divisor of two positive integers. Figure 1 shows the interface of the circuit, as well as the algorithm that it should implement.

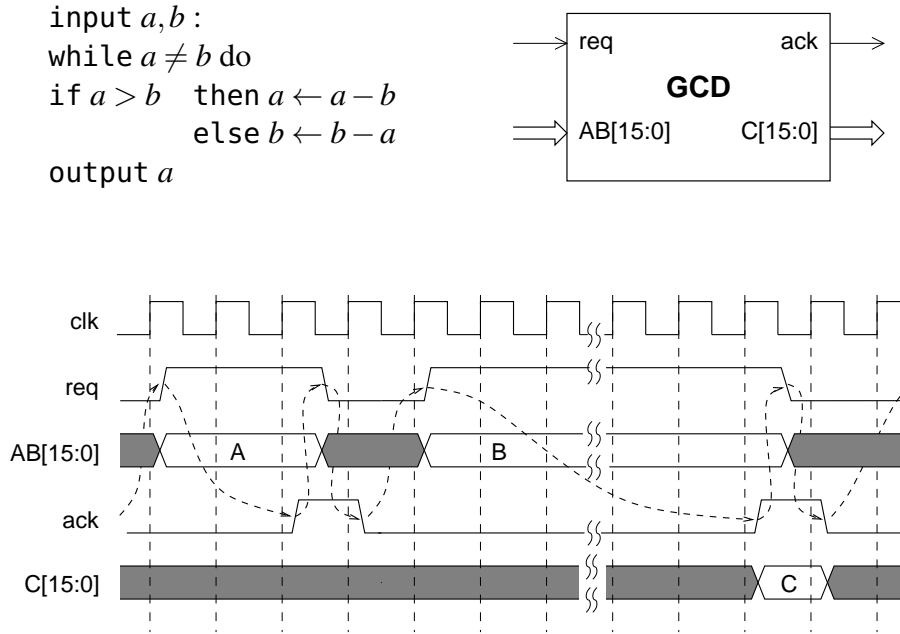


Figure 1: SLT module computing the greatest common divisor: $C = \text{gcd}(A, B)$.

On its interface, the circuit must behave as a synchronous locally timed module (SLT-module), with signals `req` and `ack` following a 4-phase fully interlocked handshake protocol as illustrated in Figure 1. In order to allow you to test the circuit using the Nexys board, operands `A` and `B` are input one at a time (rather than in parallel). For this reason, the “normal” SLT protocol is extended with an additional and initial `req`-`ack` handshake, where the first operand is input. The second `req`-`ack` handshake is the “normal” SLT handshake where `req` indicates that the operand is valid and that the computation can start, and where `ack` indicates that the computation has completed and that the result is valid on the output.

Looking at the algorithm, it is obvious that a possible implementation could involve a finite state machine controller and a datapath containing two registers (`A` and `B`), an ALU, and one or more busses/multiplexors to implement the required signal flow. One example of such a possible FSMD-style (Finite State Machine with Data-path) implementation is shown in Figure 2 and Table 1.

But, let us not make the common mistake of getting carried away in low-level implementation details early on. Let us first develop an executable specification and a test environment. Often, when doing this, you clear out misunderstandings and resolve ambiguities; issues which would have resulted in “bugs” in the final implementation, perhaps causing the need for a major redesign.

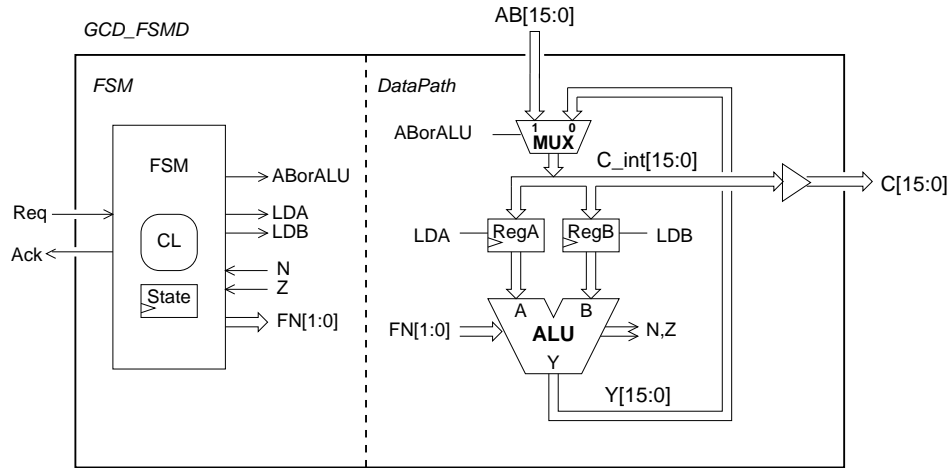


Figure 2: A *possible* FSMD-style implementation of the GCD-module, using two registers and one ALU.

| FN[1:0] | ALU operation | | Flags (all operations) |
|---------|---------------|----------------|--|
| 00 | subtract | $Y \leq A - B$ | $Z \leq '1'$ when $Y = 0$ (result is zero) |
| 01 | subtract | $Y \leq B - A$ | $N \leq '1'$ when $Y[15] = 1$ (result is negative) |
| 10 | pass | $Y \leq A$ | |
| 11 | pass | $Y \leq B$ | |

Table 1: A possible ALU that could be used in an implementation of the GCD module.

Task 0:

- Read the entire document in order to get an overview of what you are expected to do and download and skim the VHDL or SystemVerilog files.
- Assume that the data-path is implemented as shown in Figure 2, that A, B, and C are 16-bit unsigned integers, and that the technology primitives available on the FPGA are D-flip-flops and 6-input LUT's. Estimate how many flip-flops and LUTs are needed to implement the data-path? Briefly explain your estimate. Hint: this is not a difficult question – you do not need to perform a detailed implementation, a LUT can implement *any* 6-input Boolean function.

2 Step 1: Executable specification and test environment.

The first step in any design project of a certain size is to implement an executable specification and a test bench to exercise the specification. We have already done this for you. For the VHDL version of the assignment, the files are provided in a ZIP file on DTU-Learn. For the SystemVerilog version, you can clone or download the following GitHub repository: https://github.com/OttoWRas/02203_A1_SystemVerilog. It contains 3 directories, one for each of the following tasks (more information provided in Appendix A). In the `task1`-directory, you find a set of files that constitute a complete executable specification of GCD and a test environment. The test environment provides inputs to GCD and absorbs (and checks) the outputs. The GCD module and its test environment implement the signals and the handshake protocol specified in Figure 1.

Task 1:

- a) Compile the files and simulate the complete system (`gcd_tb`). You do this by opening Xilinx Vivado, creating a new project, importing the files from `task1`, and clicking the Run Simulation button in the left pane.

3 Step 2: RT-level FSM-D-style implementation.

Designing a circuit like GCD involves figuring out what registers are needed, what operations are needed, and the sequence of operations – at the clock-cycle level – that the circuit should perform. Each clock-cycle step can be described as computing a set of functions, whose arguments are a (sub)set of the current register values, and whose result values are assigned to registers at the clock tick that ends the clock cycle. This level of design is known as the RT-level (register transfer level), and a general register transfer operation may be written as:

$$R_{dest} \leftarrow f(R_{src1}, R_{src2}, \dots, R_{srcn}) \quad (1)$$

The sequence of register transfer operations, and the conditions/predicates that control this sequence, can be captured in the form of a finite state machine. In a simple Moore-type finite state machine the values of the input signals are annotated to the state transitions and the values of the output signals are annotated to the states. We will specify a similar but more abstract state graph where predicates (like “ $A > B$ ” or “ $Req=1$ ”) are annotated to the state transitions and where RT-operations are annotated to the states. In this way we get what is known as a “Finite state machine with datapath” (FSMD) description. Such an FSMD description specifies “what” but not details on “how”, and it can be synthesized and implemented.

The latter is important: you have full control over the implementation produced by the synthesis tool, but you save the effort of writing all the corresponding low-level structural VHDL or SystemVerilog code yourself. This is a key message to get from this lab assignment! There is also a danger of working at this higher level of abstraction, and that is that you may start to think as if you are “programming in VHDL or SystemVerilog” – a mindset that will lead you to fail miserably as a digital systems designer. You should always think in terms of what hardware you want to create, and then write your HDL code accordingly in a carefully structured way as indicated in Figures 4, and 5, and 6, i.e., cleanly separated into combinational logic (which is synthesized) and D-flip-flops (which are inferred).

Let us now continue the design of the GCD module, and let us aim for a simple implementation using a finite state machine controller and a datapath with the smallest possible amount of hardware resources: one ALU, two registers (for A and B), and some selection circuitry to provide data-pathways between these resources. Such a datapath was introduced previously and shown in Figure 2.

Your task, which will be explained in more detail in the following, is to design an FSM-style implementation (architecture body) of the GCD module. This involves: (1) drawing an FSM-style state graph as explained above, (2) writing the corresponding VHDL or SystemVerilog code for the new architecture body, (3) simulating your design by re-using the testbench from before (4) synthesizing your design, and (5) uploading it to the FPGA-board and testing it. In the following, we will explain in more detail what to do.

The necessary files needed to describe this new architecture of the entity/module `gcd` are in the `task2` directory in provided files. As you will see, the `gcd` entity now has a new architecture body (called `fsmd`). Your task is to complete the `fsmd` architecture of the new entity `gcd`.

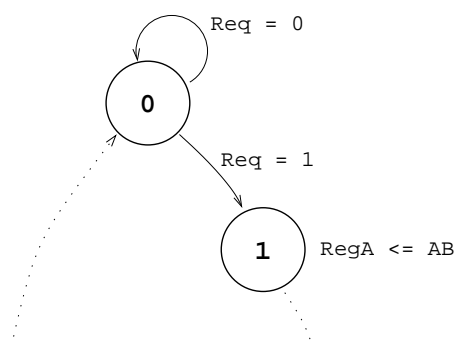


Figure 3: State diagram for the control FSM (Moore type).

| GCD signal | Connected to |
|-----------------|--|
| clk | A 100 MHz clock generated on the Nexys4DDR board |
| reset | Pushbutton BTNU. |
| req | Pushbutton BTNR. |
| AB(15 downto 0) | Switches SW[15], SW[14], ... , SW[0]. |
| ack | LED16_B. |
| C(15 downto 0) | LED[15], LED[14], ... , LED[0]. |

Table 2: Connecting GCD to the outside world.

Task 2:

- a) Study the VHDL or SystemVerilog files in the `task2` directory. Draw a block diagram of the complete system (entities connected by signals) and indicate the names of the entities and the architecture bodies used for the different entities. Hint: the top entity is `gcd_top`.
- b) Design a state diagram for the controller by completing the state diagram fragment shown in Figure 3. Do not specify individual control signals; we will add such details later. What matters are the states, the state transitions, the conditions/predicates controlling the state transitions, and the RT operations performed in the different states. The latter is exemplified by `RegA <= AB` in state 1 in Figure 3. *Keep in mind that the aim is an implementation using only one ALU. Hence, you can not do more than a single two-operand operation in a clock cycle.*
- c) Write a new RT-level FSM-style architecture body for the GCD entity (complete the code in the file `task2/gcd.vhd`). The code should be structured following the two-process template explained in the lectures and illustrated in Figure 4, and Figure 5, and 6. Remember to implement the required fully interlocked handshaking, and remember that after reset your circuit should be able to perform many computations in sequence without being reset again.
- d) Create a new project for task 2 in Vivado. Simulate your gcd design using the `gcd_tb` testbench from task 1. Verify that your design is correct. Make sure that you perform an exhaustive test, by providing (a sequence of) operand pairs, which will cause all state transitions in the state diagram to occur during the simulation. You may need to increase the simulation time in Vivado (in the top toolbar) to finish all of the tests.
- e) Synthesize `gcd_top` along with your FSM-style design of gcd and the debouncer circuit using XILINX Vivado. An XDC file specifying the pinout is provided. Table 2 shows the connections it establishes between the GCD entity and resources on the Nexys4DDR board.
- e) Upload the synthesized design to the FPGA board and test it. Demonstrate your working solution to the teaching assistant.
- f) Check the synthesis report and state the amount of resources required to implement your design. Is it as you expected, i.e., can you explain the number of adders/subtractors/comparators, the number of flip-flops, and the number of 6-input LUTs? Do the numbers match with your answer to task 0b).

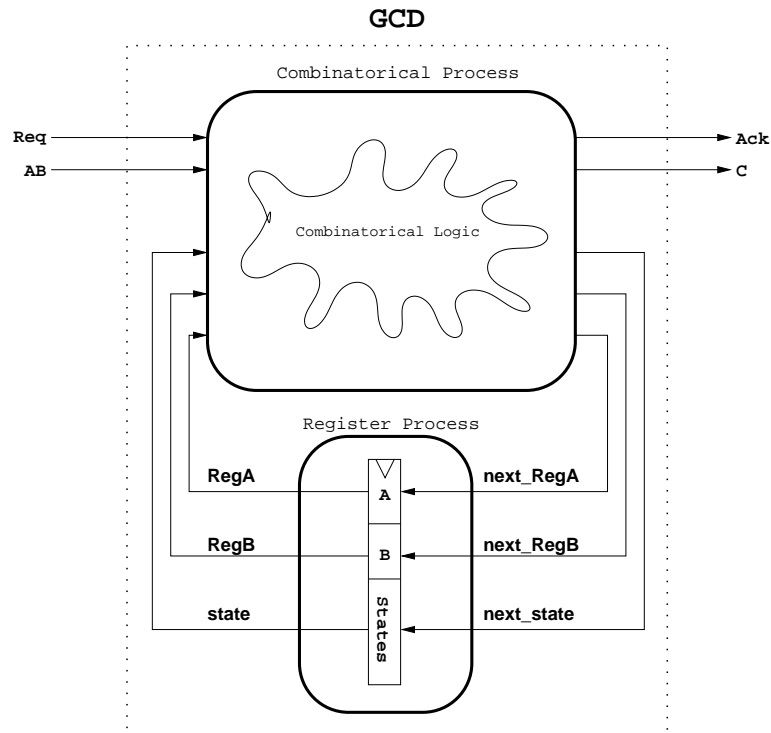


Figure 4: An RTL structure describing the processes needed.

```

1
2 ARCHITECTURE behavioural OF gcd IS
3     TYPE StateType IS (
4         ...
5     );
6
7     SIGNAL state, next_state : StateType;
8
9     SIGNAL RegA, next_RegA : unsigned(15 downto 0);
10    SIGNAL RegB, next_RegB : unsigned(15 downto 0);
11
12 BEGIN
13 comb: PROCESS (state, RegA, RegB, req, AB) is
14     BEGIN
15         CASE state IS
16
17             ...< Combinational logic body > ...
18
19         END CASE;
20     END PROCESS;
21
22 reg: PROCESS (clk, reset)
23     BEGIN
24
25         ...< Register body >...
26
27     END PROCESS;
28 END behavioural;

```

Figure 5: Template for a FSM in VHDL.


```

1 module gcd (
2     ...
3 );
4
5     // State type
6     typedef enum logic [1:0] {
7         ...
8     } state_t;
9
10    state_t state, next_state;
11
12    logic [15:0] RegA, next_RegA;
13    logic [15:0] RegB, next_RegB;
14
15    always_comb begin
16        case (state)
17
18            ... <Combinational logic code> ...
19
20        endcase
21    end
22
23    always_ff @(posedge clk or posedge reset) begin
24
25        ... <Register code> ...
26
27    end
28
29 endmodule

```

Figure 6: Template for a FSM in SystemVerilog.

4 Step 3: Exploring design optimizations

Most likely you just figured out that your design uses more adders/subtractors/comparators than you had in mind. The reason is that every time you write an operator symbol in VHDL or SystemVerilog, you get a corresponding hardware unit. Using techniques such as *operand sharing* (implementing the arithmetic operations performed in different states using a single time-shared operator circuit) or *functionality sharing* (implementing “related” arithmetic operations like “>” and “-” performed in different states using a single time-shared operator circuit) you may be able to reduce the size of the circuit towards what you estimated in Task 0b).

Another issue is the speed of the design, i.e., the number of clock cycles required to compute a result. Perhaps it can be reduced by using more than one operator circuit. One idea is to calculate both A-B and B-A and to commit the relevant result to the relevant register. Not sharing operators may also reduce the area of the circuit.

Finally, we mention that it is possible to use more efficient algorithms than the repeated subtraction we have been studying so far.

All of these optimizations can be performed by performing small modifications to your VHDL or SystemVerilog code from Task 2.

Task 3:

- a) Select and describe at least one optimization that you want to perform and explain briefly what improvements you anticipate.
- b) Write a new VHDL or SystemVerilog file describing your optimized GCD circuit. Use the same entity declaration as before, but give your architecture a new name.
- c) Report and discuss the improvement you achieved.

5 (OPTIONAL) Step 4: Low-level component-based implementation.

We will now explore the VHDL's or SystemVerilog's capabilities to express structure, i.e., schematics. Your task will be to develop a structural implementation of GCD using a finite state machine and a structural implementation of the data path as shown in Figure 2. You will have to write some new VHDL or SystemVerilog files from scratch, in which you define the new entities and architecture bodies. The structural architecture of the entity `gcd` must consist of two entities `FSM` and `datapath`, and the structural architecture body of `datapath` must use instances of the following components:

- `mux` : 16-bit 2-to-1 multiplexor
- `buf` : 16-bit buffer/driver
- `reg` : 16-bit register with enable
- `alu` : 16-bit ALU with the functions described in Figure 2.

The file `comp.vhd` contains entity declarations and behavioral architecture bodies for these components. Similar components can be expected to be available in a range of implementation libraries from FPGA and ASIC vendors.

Task 4:

- a) Develop a structural VHDL model of the system shown in Figure 2. This involves:
(1) A new structural architecture body for `gcd_module` consisting of two entities/-components `FSM` and `datapath`. (2) A definition of the `datapath` entity. (3) A structural architecture body for the `datapath`.
- b) Simulate the design (as in Task 2c).
- c) Synthesize the new implementation
- d) Compare the size of the circuit with the FSMD-based circuit from Tasks 2 and 3. Did it pay off to perform the structural design of the datapath?

6 Your report

Write a short report describing your solutions to tasks 0, 1, 2, 3, and 4.

Each team must hand-in an electronic version of the report in PDF format and the source files as ZIP archive. In the ZIP file, only include the VHDL or SystemVerilog files and any relevant file you have produced. *Do not include the large Vivado files.* The hand-in is done by one student. The remaining students in the group should make sure they join the group so that we have everybody registered. The deadline is stated in the assignment facility of DTU-Learn.

The report must have a cover page containing the following elements: (i) the title of the report, (ii) your team number, and (iii) the name and student ID of each team member. In addition, you must include the following statement at the bottom of the cover page:

*We confirm that this report contains our own independent and original work.
Unless otherwise explained in the preface of the report, all group members have
contributed equally to all parts of the work.*

The report (and this Assignment) will contribute approximately 15% towards your final grade in this course.

A List of provided files

As previously mentioned, you can choose to do the assignment in both VHDL or SystemVerilog. For the VHDL version of the assignment, the files are provided in a ZIP file on DTU-Learn. For the SystemVerilog version, you can clone or download the following GitHub repository: https://github.com/OttoWRas/02203_A1_SystemVerilog.

For both version, you will find 3 directories task1, task2, and task4, with files needed for the lab exercise. In the following, you will find a description based on the VHDL version. For the SystemVerilog version, the same applies, but the files will have the .sv extension. In addition, for the SystemVerilog version, you can also refer to the README file in the repository.

Task 1

The following files constitute a complete top-level specification and testbench for the gcd_top entity.

gcd_top.vhd The entity declaration and an abstract specification of the architecture (suitable for simulation only). As we have not yet decided how to implement the module, the delays of 15 ns and 5 ns are arbitrary. The purpose of these delays is to produce “nice” waveforms during simulation.

gcd_tb.vhd A testbench for the gcd_top entity used for simulation and waveform generation.

Task 2

Task 2 uses the same environment and GCD entity as in Task 1. The new files specify a new architecture body for GCD:

gcd_tb.vhd The same testbench as used in task 1. Used to simulate gcd_top.

gcd_top.vhd The top-level circuit to be implemented on the FPGA. It instantiates and connects two components gcd and debounce.

gcd.vhd The entity gcd and a FSMD-style architecture body for it. *Your task is to complete the architecture description.*

debounce.vhd The entity and architecture of the debounce component used to debounce and synchronize the request signal.

Nexys4DDR_gcd.xdc XDC-file defining the pinout, i.e., how signals from the top level entity gcd_top are connected to switches, buttons, and LEDs on the Nexys4DDR-board.

Task 4

Definitions of the components used in the structural implementation of the datapath:

comp.vhd The components:

- buf : 16-bit buffer
- mux : 16-bit 2-to-1 multiplexor
- reg : 16-bit register with enable
- alu : 16-bit ALU with the functions described above