02203 Design of Digital Systems (fall 2012)

Greatest Common Divisor

A digital circuit design exercise using VHDL, ModelSim,
XILIX ISE and a Spartan6 FPGA board.

2012 v.2

Preparing for the lab exercise

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-files (this is Task 0a)*
- *have answered the questions in Tasks: 0b), 1a), 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 during the 2-3 lab sessions that we have scheduled.

Abstract

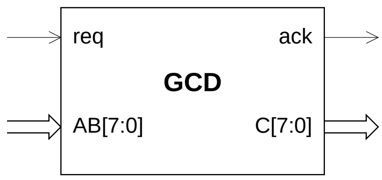
The purpose of this lab is to teach a systematic, top-down, simulation-based, design flow to be used when specifying, designing and implementing digital systems. The design flow is based on the VHDL hardware description language, the ModelSim simulator, the XILINX ISE synthesis tool, and a XILINX Spartan6 board (Nexys 3 from Digilent Inc.) for implementation. The particular circuit that we consider implements Euclid's algorithm for finding 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 files (including a specification and a test-bench) and you will have to write some new VHDL files describing your design. 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 find a list of questions related to that section.

Contents

1 Introduction

The particular circuit that we will design, should implement Euclid's algorithm for finding 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.

input a, b :
while $a \neq b$ do
if $a > b$ then $a \leftarrow a - b$
else $b \leftarrow b - a$
output a



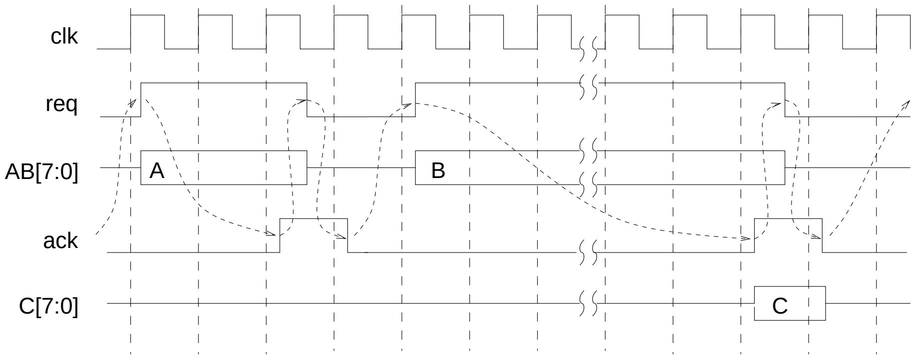


Figure 1: SLT module computing the greatest common divisor: C = gcd(A,B).

On its interface the circuit behaves like a synchronous locally timed module (SLT-module), with signals `req` and `ack` following a 4-phase fully interlocked handshake protocol. In order to allow you to test the circuit using the Spartan6 board, operands `A` and `B` are input one at a time (rather than i 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: to get carried away in low level implementation details early on. Let us first develop an executable specification and a test environment. Often when doing this for a complex system, 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.

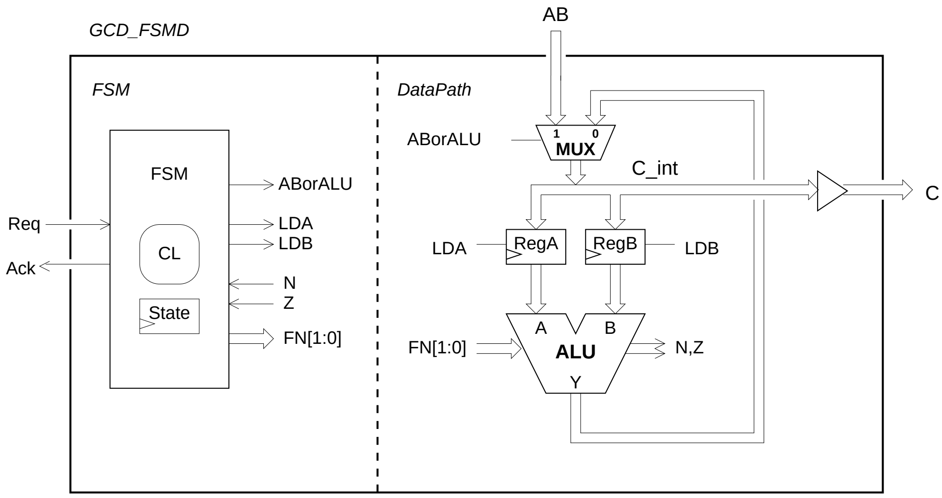


Figure 2: A possible FSMD-style implementation of the GCD-module, using two registers and one ALU.

FN[1:0]	ALU operation		Flags (all operations)
00	subtract	$Y <= A - B$	$Z <=$'1' when $Y = 0$ (result is zero)
01	subtract	$Y <= B - A$	$N <=$'1' when $Y[7] = 1$ (result is negative)
10	pass	$Y <= A$	
11	pass	$Y <= B$	

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

Task 0:

a) Read the entire document in order to get an overview of what you are expected to do and download, unzip and skim the VHDL-files.

b) Assume that the data-path is implemented as shown in in figure 2, that A, B and C are 8-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 LUT's 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 some size, is to implement an executable specification and a test bench to exercise the specification. We have already done this for you. In campusnet filesharing you find a file `gcd2012.zip`. It contains 3 directories, one for each of the following tasks. In the `task1`-directory you find a set of files which constitute a complete executable specification of GCD and a test environment. The test environment provides inputs to GCD and absorbs (and checks) the outputs. As VHDL does not allow abstraction of interfaces, the GCD module and its test environment implement the signals and the handshake protocol specified in figure 1.

Task 1:

a) Study the specification given in the `task1` files in `gcd2012.zip`. 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.

b) Compile all the files and simulate the complete system. You do this by opening ModelSim, changing the directory in the transcript window to the `task1` directory and running the task1.do with the command “`do task1.do`”.

3 Step 2: RT-level FSMD-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:

$$R_{dest} <= f(R_{src1}, R_{src2}, \ldots, R_{srcn}) \tag{1}$$

The sequence of register transfer operations, and the conditions/predicates that controls 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 an 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 that is produced by the synthesis-tool, but you save the effort of writing all the corresponding low level structural VHDL 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" – 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 VHDL code accordingly in a carefully structured way as indicated in*

Figures 4 and 5, i.e., cleanly separated into combinational logic (which is synthesized) and D-flip-flops (wich 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 FSMD-style implementation (architecture body) of the GCD module. This involves: (1) drawing an FSMD-style state graph as explained above, (2) writing the corresponding VHDL-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 `gcd` are in the `task2` directory in `gcd2012.zip`. 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`.

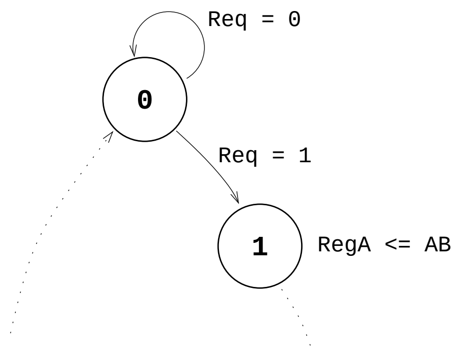


Figure 3: State diagram for the control FSM (Moore type).

GCD signal	Connected to
clk	A 100 MHz clock generated on the Spartan6 board
reset	Pushbutton BTN1.
req	Pushbutton BTN0.
AB(7 downto 0)	Switches SW7, SW6, ... , SW0.
ack	LED_0 on the Spartan6.
C(7 downto 0)	LED7, LED6, ... , LED0.

Table 2: Connecting GCD to the outside world.

Task 2:

a) Study the VHDL-files in the `task2` directory in `gcd2012.zip`. 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.

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.

c) Write in VHDL a new RTL-level FSMD-style architecture body for the GCD entity. The code should be structured following the two-process template described in most textbooks on VHDL and illustrated in Figure 4 and Figure 5.

d) Simulate the entire system again, this time using the new architecture body `fsmd_io` of the entity `gcd_sys`. 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. Run the `"do task2.do"` in ModelSim to configure and start the simulation as you did in task 1.

e) Synthesize the GCD entity (consisting of `gcd_sys` and your `gcd`) using XILINX ISE. A UCF-file specifying the pinout is provided. Table 2 shows the connections it establishes between the GCD entity and resources on the Spartan6 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 recourses 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).

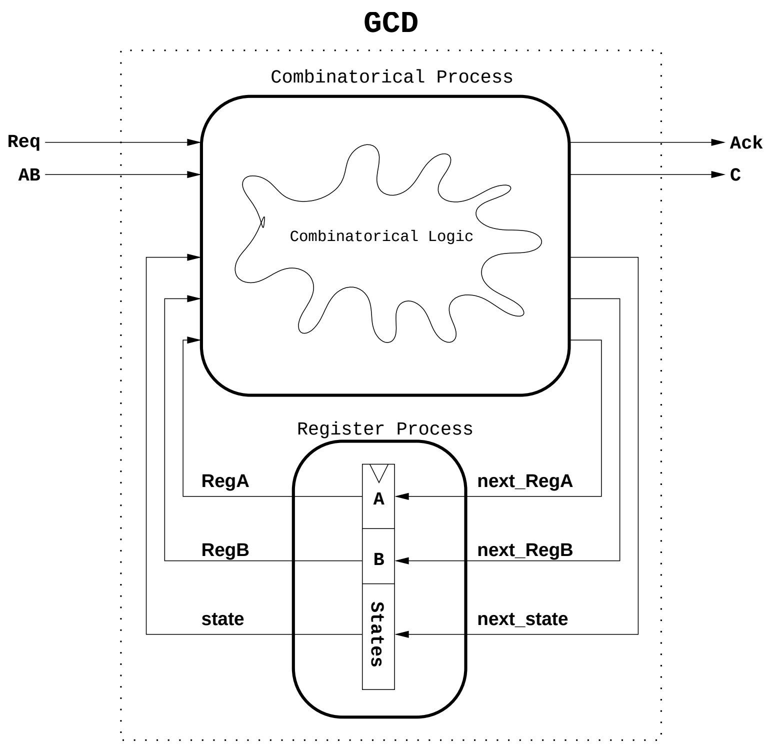


Figure 4: An RTL structure describing the processes needed.

```
ARCHITECTURE behavioural OF gcd IS
    TYPE StateType IS ( ...
        );
    SIGNAL state, next_state : StateType;
    SIGNAL RegA, next_RegA : unsigned(7 downto 0);
    SIGNAL RegB, next_RegB : unsigned(7 downto 0);

BEGIN
comb:  PROCESS (state,RegA,RegB,req,AB) is
    BEGIN
        CASE state IS

            ...< Combinational logic body > ...

            END CASE;
    END PROCESS;

reg: PROCESS (clk,reset)
    BEGIN

        ...< Register body >...

    END PROCESS;
END behavioural;
```

Figure 5: Template for a FSM.

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, 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 perform one of the following three assignments: (i) A<=A, B<=B (ii) A<=A-B (iii) B<=B-A. This may even reduce the size 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-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) Implement your optimization by rewriting your VHDL code from Task 2. Keep your existing work and write a new source-file describing your new and improved architecture body of the entity `gcd`. Write a new configuration (another new source file) in which you use your new architecture of `gcd`

c) Report and discuss the improvement you achieved.

5 Step 4: Low-level component-based implementation.

We will now explore VHDL'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 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	: 8-bit 2-to-1 multiplexor
buf	: 8-bit buffer/driver
reg	: 8-bit register with enable
alu	: 8-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). Write and use a new configuration that instantiates the desired implementation.

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 of tasks 0, 1, 2, 3 and 4 and upload your VHDL-files to filesharing in your teams subgroup in campusnet. Turn in a printed report using the mailbox located in the ground floor of building 322 labelled with the course number (02203) and name. The report must contain listings of your VHDL-code. The report will contribute approximately 20% towards your final grade in this course. The deadline is stated in the course plan in campusnet.

The report must have a cover page with: a title of the report, your team number and the name and student id. number for each team member. Each team member must sign this cover page, and you must include the following statement in the bottom of the the cover page:

> "By signing this report we confirm that it contains our own independent and original work and that, unless otherwise explained in a preface in the report, all group members have contributed equally to all parts of the work."

Hence, if you do find it necessary or relevant to include fragments of text or VHDL-code not developed by yourself, you should reference this properly. Failure to do so is considered plagiarism, and will be reported to DTU's administration.

A List of files provided.

The file `gcd2012.zip`, which is available in campusnet filesharing, contains 3 directories `task1`, `task2`, and `task4`, with files needed for the lab exercise.

Task 1

The following VHDL files constitute a complete top level specification and testbench for the `gcd_sys` entity.

`clock.vhd` A clock generator. The period is specified in a generic and defaults to 10 ns.

`env.vhd` The environment of GCD. It provides operands `A` and `B` and the associated `req` signal, and when `ack` indicates that the result is ready, it loads `C` into an internal (dummy) register.

`gcd_entity.vhd` The entity declaration of `gcd_sys`

`gcd_spec.vhd` An asynchronous and behavioral architecture body of the `gcd_sys` entity, i.e., an abstract and executable specification. The delays of 15 ns and 5 ns are arbitrary, as we assume that we have not yet decided how to implement the module. The purpose of these delays is to produce "nice" waveforms in ModelSim.

`test.vhd` The structure of the testbench to be simulated.

`sim.do` A script to setup the waveforms in ModelSim and run the simulation.

`task1.do` A script for compiling and starting the simulation in ModelSim.

Task 2

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

`gcd_sys.vhd` A new structural architecture body for the `gcd_sys` entity. In 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_entity.vhd` The entity of the debounce component to debounce and synchronize the request signal.

`debounce_sim.vhd` An architecture body of the debounce entity for simulation.

`debounce_synth.vhd` An architecture body of the debounce entity for synthesis.

`Nexys3_Master.ucf` A UCF-file defining the pinout, i.e., how signals in your design are connected to switches, pushbuttons and LED's.

`sim.do` A script to setup the waveforms in ModelSim and run the simulation.

`task2.do` A script for compiling and starting the simulation on ModelSim.

Task 4

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

`comp.vhd` The components:

tri	: 8-bit tristate buffer
mux	: 8-bit 2-to-1 multiplexor
reg	: 8-bit register with enable
alu	: 8-bit ALU with the functions described above