GCD: Hw—A Hardware Solution

(Laboratory Session 4, EDAN15)

Flavius.Gruian@cs.lth.se

February 28, 2017

1 Introduction

In this laboratory session you will have to choose a part (e.g. modulo operation, *gcd* for two numbers, or *gcd* for N numbers) of the *gcd* algorithm for N number and implement it in hardware. You will write and simulate your *gcd* module in VHDL using Vivado for a few of the data sets used in the previous sessions. The hardware should input/output data via the interface specified herein. Be aware that in the next laboratory session you will have to connect your module to a MicroBlaze processor, and implement/evaluate a hardware-accelerated *gcd* solution.

2 Hardware Acceleration

The performance of a pure software application can be improved in most cases by implementing certain tasks in hardware. If not for the increased speed of a hardware implementation, at least the possibility to execute operations in parallel is what makes hardware acceleration a performance booster. Furthermore, the chip area is often only marginally increased by adding a specialized core to a design, as opposed to use an additional general purpose processor. The drawbacks of building a custom core versus using an general purpose processor are the increased design time, lack of flexibility, the need for knowledge and tools.

The current and the next laboratory sessions are centred on adding a custom hardware accelerator to the *gcd* uni-processor system built in the first lab. There are a number of ways in which a custom hardware accelerator can be added into

the basic system. Although you could use any methiod, we will provide support for a memory mapped accelerator, using the AXI bus. A wrapper for a slave AXI-light core will be provided for you to this end. Thus, your core should be able to connect to the MicroBlaze via the already existing peripheral AXI bus.

note Other methods to insert the core into the system are possible – such as connecting it via point-to-point links (AXI Streams). Note also that the functionality of the skeleton we provide is rather limited, and could be extended with interrupts, Direct Memory Access (DMA) to fetch data from the memory, etc. If you opt for any of these features or another way of connecting into the system, you will need to implement the interface on your own. Vivado does provide some support for creating new IPs with a certain bus interface that should be very useful.

3 Interface Specification

The intention is to use memory mapped registers to communicate with the processor. This means that the processor will see some of the registers of your core at certain addresses in the address space. To *send* data to the core, the processor will write at a given address. To *receive* data, the processor should read from a (same or another) given address. It is customary to provide both data and control registers for your core – for instance use a control bit to check by polling whether a computation is done and the data is valid. To use the provided AXI-light wrapper, you will need to drive/monitor the following signals from your hardware (see the *user_logic.vhd* file):

- **clk** The clock signal for your core (in, 1 bit).
- **rstN** The reset signal, active low (in, 1 bit).
- **wren** Active (high) if a write has been issued (in, 1 bit).
- **waddr** The address of the register to write the data to (in, ? bits).
- **wdata** Actual data to write (in, 32 bits).
- **rden** Active (high) if a read has been issued (in, 1 bit).
- **raddr** The address of the register to read the data from (in, ? bits).
- **rdata** The actual data read (out, 32 bits).
- **busy** Active (high) if the core is busy (out, 1 bit).

The protocol for writing and reading data using the signals given above is straightforward. When (rd/wr) enable goes high, the address will contain the register address. For writes the data will also be present in *wdata*. For reads, you will need to provide the data from the right register. Finally, the *busy* signal is not really used at this moment by provided AXI skeleton, but you might find some uses for it (e.g. interrupts, or waiting for the core to finish). Note that the number and deciding how to use the registers is up to you. One idea is to have a control register containing the core status, that can be polled by the processor, which reports whether the computation is complete. You could also use this register to tell the core that the data has been loaded and it can start computing. Or you might start computation as soon as you wrote data in a specific register.

4 Finite State Machines in VHDL

Finite State Machines (FSMs) are useful for modeling in many situations. Your hardware needs to wait for data, read it, process it, and finally write the result back – we have already identified three states that you should further refine to fit your choice.

Looking at FSMs from a hardware perspective, one can identify the state (registers, memory) and the logic for computing the next state and outputs. Following this pattern, in VHDL FSMs are rather easy to model and synthesize. A helpful template for writing an FSM is given below – notice that the SYNC_PROC models the state register(s) while COMB_PROC the logic generating the next state and the outputs:

```
-- insert the following between 'architecture' and 'begin' keywords
type STATE_TYPE is (S0, S1, S2, S3);
signal CS, NS: STATE_TYPE;

-- insert the following after 'begin'
SYNC_PROC: process(Clock, Reset)
begin
  if  (Reset = '1') then
    CS <= S0;
    -- other state variables reset
  elsif rising_edge(Clock) then
    CS <= NS;
    -- other state variable assignment
  end if;
end process;

COMB_PROC: process(CS, <more inputs>)
begin
  -- assign default signals here to avoid latches
  case CS is
    when S0 =>
       -- assign outputs here
       -- assign the next state depending on various conditions
    -- have a 'when' for all states
  end case;
end process;
```

State encoding can be enforced by assigning the ENUM_ENCODING attribute. For example, for a *one-hot* encoding of the states the following lines should be inserted after the STAT_TYPE declaration:

```
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of STATE_TYPE: type is "0001 0010 0100 1000"
```

For binary or gray-code state machines, the encoding can be specified in the same way.

note More templates are available in Vivador: **Tools** → **Language Templates...**

5 Design Evaluation

For this laboratory assignment, it is recommended to use Vivado to write and simulate your VHDL code. You should create a new project and add the `user_logic.vhd` module file modified by you. Test benches (input stimuli) will be required to simulate the behavior of your core – unfortunately these need to be written manually in VHDL. Nevertheless, there are a number of online test bench generators that can help you by generating an initial skeleton you will need to fill in: `https://www.doulos.com/knowhow/perl/testbench_creation/`

To determine whether your design functions correctly, you will use simulation. At this point, it is interesting to examine both the correctness of the result and also the number of clock cycles required by your model to carry out the operations. Since these operations will most likely yield different results and timing for different input data, you should test the design for various data sets (meaning also more than two numbers!). Additionally, if possible, try to give a worst case number of cycles required by your design – an upper bound for carrying out the computations. Although you will have to initialize your design via a reset and other signals, you should only account for the time passed between the moment the (first) data is available on the input and the moment the final result is written at the output.

6 Assignment

1. Hardware/software partitioning — Decide how much to transfer to a hardware core and how much should still be done by software. For example, you may choose to implement in hardware the *gcd* of two numbers, or N numbers, or only a modulo operation, etc.
2. Choose a good algorithm for the part you decided to implement in hardware. Regardless of the problem, there are usually several ways of finding a solution. Do not fall into the trap of assuming that what seems to be fast or easy to do in software has the same features in hardware (see **Final Remarks & Hints**).
3. Create a new project in Vivado, the same way it was done in the introductory lab. Add a new design source, called `user_logic.vhd`, and using the **Define Module** dialog box (pops up once you added the source), define the **I/O Port Definitions** as mentioned above (see also Figure 1). This is the top module of your design, for now.
4. Write your own hardware in the architecture section for the **user_logic** entity in the `user_logic.vhd`. You may use more files and modules, but make sure you include them for the next lab.
5. Simulate the design for several input data sets. It is recommended to do this via a test bench, an additional VHDL module that generates the required input signals and tests the outputs. Or you can simply force the signals to various values during simulation.
6. Synthesize your `user_logic.vhd` file and make sure you do not get any warnings for latches.
7. If possible, try to estimate and report the worst case number of cycles for 2 or N numbers.

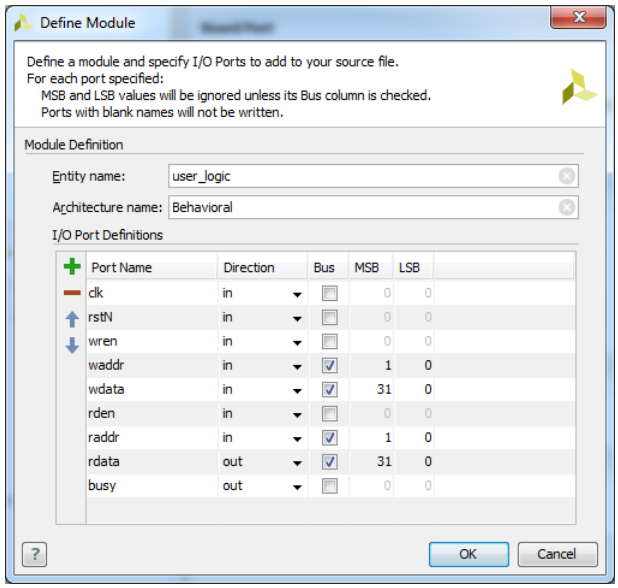

Figure 1: Define Module Dialog Box

7 Final Remarks & Hints

- Division and modulo is hard/costly to fully implement in hardware. There are *gcd* algorithms which use only subtractions and maybe shifts. These are preferred for hardware implementation.
- Think your design through, draw any FSMs you plan to implement before actually starting to write the code.
- The more you implement in hardware, the faster your design will be and the less you will have to work in the next laboratory session. If you are still pondering how much or what to implement in hardware, talk to your lab assistant.
- It is a good idea to implement a control register, which contains various flags the processor can read and set. The number of data registers is up to you and the functionality you want to implement, but it might be a good idea to keep it low.
- It is not required to use all the input signals given in the interface above, nor set all the output signals, if your software is not supposed to make use of them.
- Using files (to read from) in your VHDL test bench is totally acceptable and recommended, especially if your stimuli data is large, as in some of the test sets. You may pre-process the provided sample data files to fit your test bench.