GCD: S1—A Software, Uni-Processor Solution

(Laboratory Session 2, EDAN15)

Flavius.Gruian@cs.lth.se

February 28, 2017

1 Introduction

In this laboratory session you will have to implement and compare two pure software solution of a Greatest Common Divisor (GCD) algorithm for N numbers. Furthermore, the software has to run on a single processor system, based on a MicroBlaze soft core, using the available Xilinx FPGA platform. Once having a correctly functioning system, you will be required to report the device utilization of your solution along with power, energy estimates and speed measurements for various compiler optimization levels. The development and testing will be carried out using the *Xilinx Vivado/SDK* and the *Digilent Nexys-4* evaluation board.

2 Greatest Common Divisor

The greatest common divisor of two integers a and b, denoted by $gcd(a, b)$, is the largest integer that divides both. There are a number of algorithms that can be used for determining the $gcd(a, b)$, some faster, some more elaborate. For example, a naïve method is to test all numbers downwards from a or b – whichever is smallest – and stop when you found c, that divides both a and b. Better methods are *Euclid*'s algorithm or the *binary algorithm* (which can be found in almost any algebra book or with a search on the internet).

Note, however, that you are required to implement an algorithm for finding the gcd of N numbers, where N (and the actual numbers) is specified at run-time. It is possible, of course, to use the gcd method for two numbers, considering that $gcd_3(a, b, c) = gcd(a, gcd(b, c))$.

3 Base Architecture

The support architecture for this assignment is similar to the one built in the tutorial lab (1st). The system should consist of one MicroBlaze core, a dual port local memory for instructions and for data, a peripheral bus (AXI) connected to a UART core and a `timer` core, to carry out the performance measurements.

Input data for your program is provided via the UART, using the following format: `N a1 a2 ... aN`. In other words, the first number received is the number of positive integers for which to compute *gcd*, followed by the actual integers separated by spaces. To read numbers from STDIN you may use the **getnum( )** function provided via the course web page.

4 Design Evaluation

There are several important measures that characterize a design. Performance, chip area (cost) and power consumption are arguably the most interesting for embedded systems. Throughout these labs we will focus on performance (or speed) and cost (cell count or device utilization in the FPGA case). Your job as a designer is to examine several solutions in order to maximize the performance and decrease the cost. Power and energy consumption should also be taken into consideration when comparing your solutions.

While the device utilization is easily determined from the Vivado implementation reports, design performance depends both on the algorithm and clock frequency. Finally, for a fixed clock frequency, the interesting parameter is the number of clock cycles required to perform a certain operation. For this it is enough to use a dedicated clock counter IP, available in Vivado under the name `axi_timer`. On the software side (SDK), you will need to make sure your programs enable the counter (with auto-load on) before you can start to read the counter values. You may configure the timer at runtime either by:

a **a low level approach:** directly writing at the addresses at which the IP registers are mapped, by using *xio.h* macros, or by

b **higher level API:** using more specialized macros/functions provided by the higher level device drivers for the timer core, see e.g. *xtmrctr.h.*

note In SDK there is an easy way to inspect API description and even examples for the IPs included in the architecture. These can be found in the BSP projects associated with the hardware platform, in the BSP Documentation folder.

Using higher level drivers is preferred since it makes the code easier to read. Some of the useful macros and functions in this case are

- XTmrCtr_Initialize
- XTmrCtr_Start

- XTmrCtr_GetValue

but you will need to figure out how to use these by looking at the examples (e.g. the polled version)!

5 Assignment

Your assignment revolves around designing and evaluating **two** GCD algorithms on the hardware architecture described above, based on a single CPU. More precisely you should carry out the following tasks:

1. Write two GCD algorithms (the naïve one plus another one) for N numbers (read from the UART, as specified in section 3).

   note In a later lab you will need to implement a GCD algorithm in hardware, so it is recommended to choose your algorithms wisely at this point. See **Final Remarks & Hints**.

2. Use the timer IP via code for measuring the clock cycles required by your solution. Make sure you can record the computation time rather than the time required to read the data from the UART. You should attempt to split the program in an "input" phase in which you read all the numbers, a "compute" phase in which you carry out the GCD algorithm and an "output" phase.

3. Record the device utilization for your design.

4. Record the number of clock cycles required by your solution for **all** the data sets provided on the course web page. Do this for different compiler optimization levels: O0, O1, and O2 (In SDK, these settings can be changed via **Project→Properties** for your application. Open the **C/C++ Build** and then **Settings** in the Properties dialog box. Find the *MicroBlaze gcc compiler/Optimization* folder under the *Tool Settings* tab.)

5. Record the power consumption and compute the energy consumption for a few input data sets and compiler optimization levels. Recall that $Energy = Power \times Time$, but for $Time$ you may use the number of clock cycles, since all future labs will run at the same clock speed.

6. Think of ways to optimize your design, in principle for minimizing the execution time and the device utilization. Attempt to put some of these ideas in practice.

7. Make sure you save enough of this data, required by your final lab report.

6 Final Remarks & Hints

- **print()** is only able to print strings. To be able to print numbers you should use **xil_printf()**, which is similar to the classic C **printf()**, except with reduced functionality.
- **print** functions are also using the UART, thus taking a rather long time to complete. Make sure your time measurement does not include printing, or the results will be dominated by the communication time.
- Considering the relatively long delay for test running a program on the FPGA (involving Hw synthesis and getting hold of one of the shared boards), try to develop the software as much as possible on another target (Linux, Windows, ...). It is much easier to identify faults if you know which parts do work correctly.
- Be aware that in the later laboratory sessions you will have to design, simulate, and test a GCD in hardware. Note that some operations are rather hard, costly or even infeasible to fully implement in hardware (i.e. *division* or *modulo*), and therefore you might need to change your choice of GCD algorithm at that point. Comparing different algorithms on different platforms is pointless, so you will have to choose your GCD algorithms (the non-naïve one) such that it is feasible in hardware, to make use of it later on. A good algorithm for hardware implementation is, for instance, the binary (bitwise) GCD, using only bit shifts and arithmetic operations.
- Use rather large data sets (around ten 4–digit numbers) which you should also store in files, to be able to test your future designs with the same input data. We recommend you use the data sets available on the course web page, to make it easier to compare your solution to other groups' solutions.