GCD: SM—A Software, Multi-Processor Solution

(Laboratory Session 3, EDAN15)

Flavius.Gruian@cs.lth.se

February 28, 2017

1 Introduction

In this laboratory session you will have to implement and evaluate again a pure software solution of a *gcd* algorithm for N numbers. However, unlike the previous approach, the support architecture should be a multi-processor system. In particular, you are supposed to build a system with (at least) two MicroBlaze processors, working on the same data set. As in the previous laboratory session, once having a correctly functioning system, you are required to report the occupied FPGA resources of your solution along with speed measurement. The development and testing will be carried out using *Xilinx Vivado/SDK* and the *Digilent Nexys-4* evaluation board.

2 Base Architecture

The support architecture for this assignment should consist of at least two MicroBlaze core with associated local memory (for data and instructions), a peripheral bus (AXI) accessible from all processors, connected to an shared UART core and a timer.

note You may use more than two processors if you wish, however be aware that the FPGA can host a limited number cores. Expect also an increase in the time taken to do the hardware synthesis.

2.1 Add another MicroBlaze in Vivado

Adding more MicroBlazes to your system is relatively easy in Vivado. Before doing any changes to your previous design you might either save it in safe place, or choose to save a checkpoint via **File→Write Checkpoint...**, where you can return later. To add an IP (a processor), in Flow Navigator select **IP Integrator** which will bring up your Block Design Diagram. Push the **Add IP** button, as you did in the tutorial, and select a MicroBlaze processor. Click **Run Block Automation** to configure the processor and finally **Run Connection Automation** to connect it to the rest of the system. Make sure the new processor is also connected to the peripherals via the AXI bus.

2.2 Using Multiple Processors

To achieve a good performance, you will have to split the functionality between the multiple processor instances. You will need to specify which source code will be run on each of the MicroBlazes. Furthermore, the processors have to work on the same data stream provided via the UART in the same format as in the previous laboratory session (GCD:S1). We recommend you use one processor to read in the data, and then distribute it to all processors. Also once the computation is done you will need to gather information and maybe do some processing again, before outputting the result.

Sharing data between two MicroBlazes can be achieved in several ways. One way is to use shared memory, e.g. Block RAM (onchip) or off-chip memory, connected to the peripheral bus via their associated memory controllers (e.g. axi_bram_if_cntrl). Mutually exclusive access in this case can be implemented, if necessary, through a hardware mutex. Vivado also provides a hardware Mailbox core, which can be handy to communicate data between two processors.

Another way of sharing data between processors is via directional point-to-point links. For MicroBlaze these are called *Stream Links* and are implemented using the *AXI streaming* protocol. For this lab, we recommend you use this way of sharing data between processors, and therefore the description in the following focuses on an architecture using Stream Links. (You may of course use other ways of communicating between processors, but expect less help from your lab assistants in that case.)

2.3 Add Point-to-Point Links

MicroBlazes can support up to 16 duplex Stream Links. To add one, in your Block Design Diagram, double-click a MicroBlaze to bring up its **Re-customize IP** dialog box. Select the **Advanced** button in the top toolbar, and then the **Buses** tab in the left, to bring up the connections configuration. In **Stream Interfaces** section, change the *Number of Stream Links* from 0 to 1, meaning one duplex (one read and one write port) link. Click **OK**.

The processor instance you just changed now has two extra ports `S0_AXIS` and `M0_AXIS` which correspond to the read (**S**lave) end of a link and the write (**M**aster) end of another link. These should be connected to the other processor instance you have in your design.

note You may have noticed by now that the blocks in your Diagram typically have the input (read) ports on the left and the output (write) ports on the right.

Now add a Stream Link to your other processor using the same procedure as above. Finally, using the mouse, click-drag connections from the slave port of one processor to the master port of the other (and vice-versa). Your processors are now connected via a duplex Stream Link. Note that there is no buffering in this configuration: the master cannot send more than one value before the slave reads that value! Sometimes it is beneficial to be able to send several values before the receiver actually starts reading. For that you could use an `AXI4-Stream Data FIFO` IP as a buffer between a master AXIS port and a slave AXIS port. The size of the buffer can be adjusted via the **Re-customize IP** dialog box.

note After making changes in your design, it is recommended to **Validate Design (F6)** in order to check for any inconsistencies. The messages are usually pretty informative.

Once you are done modifying the design, you will need to run again through synthesis, implementation and generate bitstream, and finally to export this to SDK in order to be able to use your new architecture. SDK will prompt you that the hardware changed and you will need to reload it from file, and also regenerate the BSP.

2.4 API for Stream Links

Special instructions are available to write to/read from an Stream Link, both blocking and non-blocking, wrapped by the macros in the
`.../microblaze_0/include/mb_interface.h` or
`.../microblaze_0/include/fsl.h` headers. Note that both the master and slave links associated to the same duplex channel on a MicroBlaze have the same identifier.

note For large amounts of shared data, shared memory might be a better choice, while for small messages the solution is FSL. Combining the two is also possible and useful in many cases.

2.5 A Sample Dual-Processor Application

Once you have got both MicroBlazes running (prints from both show up on the terminal), it is time to do something useful. Let's make them talk to each other, through the Stream Link. Macros for reading and writing from/to the SL are already available in a header file that should be included in your sources `mb_interface.h` (in the `include` folder of the BSP project associated with your application). The commonly used macros are `putfsl(data, port)` and `getfsl(data, port)`. Both are blocking, `data` is the data to read/write. For the read/get operation it must be a variable, which then will be updated to contain the value read, `port` is the SL number, starting from 0 (both for read and write). An example application (assuming there are SLs in both directions):

MicroBlaze 0:

```
#include <mb_interface.h>

int main(void){
  int i = 0;
  while(i<10){
    putfsl(i, 0);
    getfsl(i, 0);
    xil_printf("pong %d\n\r", i);
  }
}
```

MicroBlaze 1:

```
#include <mb_interface.h>

int main(void){
  int i;
  while(1){
    getfsl(i, 0);
    xil_printf("ping %d\n\r", i++);
    putfsl(i, 0);
  }
}
```

3 Design Evaluation

In the final lab report, you are required to describe your solution choice, and again report both the occupancy of FPGA resources as well as the number of clock cycles required by your solutions to carry out the tasks, and the power and energy figures. Again, use the timer core to record the clock cycle count. As in the previous laboratory assignment, the time spent for reading data in is not important. However, the time spent in communicating between processors should be included. In this sense, it is recommended to split the program again in an "input phase", when all data is read in (to one of the processors) and no computation is carried out, a "computation phase", when the *gcd* is calculated (and data exchanged between processors) and an "output phase", when you display the final result to STDOUT (using for example **xil_printf()**). The time you should record is for the "computation phase" only.

4 Assignment

Your assignment to build a hardware architecture as described above and evaluate a GCD algorithm on that architecture. More precisely you should carry out the following tasks:

1. Write an M-processor ($M \geq 2$) GCD algorithm for N numbers (read from the UART, as specified in section 2).
2. Use a timer core with the necessary code for measuring the clock cycles required by your solution.
3. Record the device utilization for your design.
4. Record the number of clock cycles required by your solution for the data sets available on the course web page. Do this for different compiler optimization levels: O0, O1, and O2.

5. Record the power consumption and compute the energy consumed for various input data and optimization levels.

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 include this data in the final laboratory report.

5 Final Remarks & Hints

- Note that two processors connected to the same peripheral bus share its resources (IPs). For instance the UART core may be shared resource. Concurrent calls to **getnum()** are likely to yield very strange results. You will have to either synchronize the data input somehow or use only one of the processors to access the UART.

- The timer core may be also a shared resource. However, the processors will only read its value, without changing any data, except at start-up. Make sure you initialize the core only once, from a single processor. Furthermore, it could be easier to have only one processor keeping track of time.

- It should be clear by now that it is easier to think in terms of master/slave processors. It is recommended to use a single master processor that controls the data input, distribution, gathering, and output.

- Sometimes it is desirable to synchronize processors/tasks, using for example "rendezvous" mechanisms. This is easily achieved with two mutexes. Alternatively, with SL it is also possible to have two processors starting to execute almost at the same time from chosen points, as shown next:

```
MicroBlaze One              MicroBlaze Two

#include "mb_interface.h"   #include "mb_interface.h"
⋮                           ⋮
non-synchronized            non-synchronized
⋮                           ⋮
(SL must be empty)          (SL must be empty)
putfsl(data12, 0);          getfsl(var12, 0);
getfsl(var21, 0);           putfsl(data21, 0);
⋮                           ⋮
synchronized                synchronized
⋮                           ⋮
```