Xilinx EDK Tutorial

Flavius.Gruian@cs.lth.se

March 19, 2013

This tutorial shows you how to create and run a simple MicroBlaze-based system on a Digilent Nexys-3 prototyping board. Such a system requires both specifying the hardware architecture and the software running on it. These two are then combined into one FPGA configuration, which is used to configure the Spartan-6 FPGA located on the Nexys-3 board.

note Please be aware that, as with any tool chain with many contributors, the software we use in the lab may contain bugs as well as discrepancies between different versions. The actions described in this tutorial were carried out for version 14.4, so there might be inconsistencies with your current tool chain. With a bit of flexibility, similar actions can be carried out in your lab setup.

note To save disk space, you can always remove temporary files from the project via **Project→Clean All Generated Files**. If you work in a common directory `C:/Temp`, we strongly encourage you to remove your project at the end of your session. You are of course welcome to save it in your home directory.

1 The Sample Design

This tutorial revolves around creating a system (hardware and software) that can output a simple message via the UART (in our case over USB-UART). The simplest of programmable hardware architectures that can be built involves a single processor, a MicroBlaze in our case, and some minimal support for it (memory, interconnect). On top of this, the functionality for communicating over the UART is needed. Furthermore, without great effort, one can add functionality for sensing and controlling other physical components present on the Nexys-3 board (such as push buttons, LEDs, switches). Later on, the tutorial will show you how to create a double processor hardware architecture, that will be useful in the following laboratory assignments.

Naturally, a programmable hardware architecture will require a program (software) to run on it. In a second phase, the tutorial will address the steps needed to build an initial software environment (libraries and programs) required to implement the simple functionality we mention before: sending a message through UART.

Finally, the last part of the tutorial describes how to finally configure the FPGA with the hardware and software you just built, how to run your design and actually display the output of the UART.

2 Creating the System Architecture (Hardware)

Hardware architectures are created using Xilinx Platform Studio (XPS), a GUI that helps you to specify

- which processors, memory blocks and other soft IPs (peripherals) to use
- how the different IPs are interconnected
- the memory map, i.e. for addresses for memory mapped IO/peripherals

The output from XPS is that part of the FPGA configuration that describes the hardware of your system.

2.1 Start Xilinx Platform Studio (XPS)

You can find the program in Windows start menu. **start→ all programs→ Xilinx Design Tools→ ISE Design Suite 14.x→ EDK→ Xilinx Platform Studio**

2.2 Create a new design

When you start XPS, you have the possibility of choosing between **Getting Started** actions or have a look at the **Documentation**. To easily create a basic system, choose **Create New Project Using Base System Builder**. This action can also be launched from **File→New BSB Project...** (You may also start from a completely empty project (**Create New Blank Project**), and gradually add the needed cores one by one. This is not recommended, however, since it can be very time consuming and error prone.)

- In the first wizard dialog, specify the project location and name.

  note Note that path cannot contain spaces. We recommend you use the local `C:/Temp/yourdir` for all your projects. Working on the local file system will speed up the tools, compared to using your remote user directory.

  We advise you create a new empty directory here, since XPS will create a lot of files in the project directory. Whether you choose to build an AXI based system or a PLB based system, you will also need to specify the location of the Board Support Package files for the Digilent Nexys-3 board. This tutorial will use the AXI flow in the following.

note The files should be available on your computer, along with the Xilinx tools. Otherwise the files can be downloaded from the Digilent website. Make sure you unpack the downloaded zip file.

note There are a couple of significant differences between the AXI and the PLB flows. The PLB flow can generate a system that includes most of the board resources, including the Micron_RAM and Numonyx_PCM off-chip memories, if one would need to use these. This is also about to be deprecated, since PLB is an older standard. On the other hand, the AXI flow uses a newer standard, and can generate systems with faster clock. However, support for off-chip memories is not automatically included. This might change in subsequent version of the Digilent BSP.

In the same dialog, you will have to set the *Set Project Peripheral Repository Search Path* to the path to the directory called *lib* from the unpacked zip-file. For instance, on the laboratory machines the support package is available in `C:/Xilinx/BSP`. If you plan to use an AXI System, this path should be set to `C:/Xilinx/BSP/Nexys3_AXI_BSB_Support/lib`

note If you forget to do this, or the path is incorrect you will not be able to select the Digilent Nexys-3 board in the following steps.

Click OK.

- In the next dialog box, **Board and System Selection**, select *Digilent* for **Board Vendor** and make sure the board name and revision match your board (*Nexys3, revision B*). Keep the rest of the options as they are (*Single Processor System*, optimize for *Area*). Click Next.

- In the next dialog box, **Processor, Cache, and Peripheral Configuration**, you can select the clock frequency for your system, the memory size and the initial IP cores (for accessing physical board input/output). We recommend you keep the 100MHz clock, increase the local memory to 32KB and remove the *Ethernet_lite* core from the included peripheral. Remove as well all the IPs starting with *Digilent*, or you will run into some harder-to-fix issues later on.

  note A fast clock frequency imposes rather strict constraints on the design timing, which may prolong the hardware synthesis time. Also, having more IP cores increases the demand on area and interconnect routing, which may also prolong the hardware synthesis time. If you want to reduce the time you wait for your design to synthesise, you might want to choose a slow clock and remove as many peripheral as possible (except the UART).

  At this point you may add other cores, such as the *axi_timer*, that may come in handy later on. You may also add cores after you generated the initial design. Click Finish.

You have now created your initial hardware platform. At this point you may simply run logic synthesis to obtain the first part of the FPGA configuration, as will be explained later.

However, this is a good time to explore XPS a bit. Let's start by looking at some details of the hardware architecture generated for you. Do not worry if you do not get all the abbreviations. Here is a partial list:

LMB	Local Memory Bus	CPU ↔ memory bus (point to point link)
AXI	Advanced eXtensible Interface	CPU ↔ peripherals (system bus)
GPIO	General Purpose IO	glue logic for the buttons/leds on the board
UART	serial port	text input/output of your application

There are a number of frames in the XPS window. The leftmost *Navigator* is a navigation menu. The bottom frame is the console useful to inspect the warnings and errors you might get during synthesis.

The frame right besides the *Navigator* contains two tabs, *Project* and *IP Catalog* (the tabs are in the bottom of the frame). The *Project* tab contains the files describing the generated system, as well as log files and the project options.

note It is worth having a look at the `MHS, UCF` and the `OPT` files. For instance the User Constraint File (`UCF`) binds external ports in your design to physical pins on the FPGA. It is important that the names in the design are the same as in this file, so do not rename the external ports. Also if you add or remove external pins (when adding/removing a GPIO for instance), this file must be edited.

The *IP Catalog* contains a tree of all the IP components you can use in your system.

The frames to the right in the XPS window are used for exploring and editing your hardware architecture. The *System Assembly View* tab in the middle view offers a structured way of exploring the architecture. The *Graphical Design View* tab contains a block schematic of the architecture, containing most of the signals and cores. The *Design Summary* tab gathers numerical information and output reports from the different steps of the synthesis process.

Select the *System Assembly*, which should contains a list/tree of your system components and their connections. Notice the three coloured buses. The two blue are LMBs connecting the processor and the memory controllers, one is used for instruction fetch (ilmb) and the other is used for data read/write (dlmb). The red lines are not buses, but illustrate the connection between the memory controller and the memory block(s). Finally the green line is an AXI bus, connecting the processor to the peripherals. All peripherals are memory mapped, thus they can be addressed through read/write to memory addresses (see the *Addresses* tab at the top of this view). You may connect/disconnect components to/from a bus by clicking on the circles. Double click on a component to get a dialog window in which you can change its parameters, view its VHDL code or even the documentation as PDF.

Feel free to explore more of the XPS environment at your leisure.

2.3 Building your hardware

You are now ready to create that part of the FPGA configuration that describes your hardware architecture. This *synthesis* step is complex, thus time-consuming the first time (depending on your machine performance and the design constraints, this might take 10 minutes or more). Subsequent changes to the architecture may require only partial rebuilds, so they might take less time, depending on the extent of the changes (changing one IP parameters, reconnecting one signal are minor changes).

Assuming your hardware is fixed for now (you may always return to XPS and change things, which will be detected in the rest of the design flow), you are ready to move on to writing the software. For this, Xilinx provides the SDK, which is an Eclipse based environment. All you need to do is select the **Export Design** icon in the *Navigator* menu, and click on the *Export & Launch SDK* button in the pop-up. This will carry out the hardware synthesis (may take some time, as mentioned before) and start the Eclipse Xilinx SDK environment.

3 Creating the Software

The Xilinx Software Development Kit (XSDK), which is based on Eclipse, may be either invoked from the XPS, or found alongside XPS in **start→ all programs→ Xilinx Design Tools→ ISE Design Suite 14.x→ EDK→ Xilinx Software Development Kit**.

note At launch, XSDK prompts with a request for a workspace to use (as in standard Eclipse). It is recommended you select a brand new workspace for each hardware design, since using a pre-existing workspace might overwrite certain essential parts of the workspace. It is up to you to select a suitable path here.

In XSDK there are in principle three kinds of projects:

hardware platform specification describes the hardware platform, and it is automatically created when exporting a project from XPS and is updated whenever the hardware architecture is changed.

board support package describes the software environment set-up for the specific hardware architecture, including the drivers, libraries and operating system. Any number of such projects may be created for a given hardware.

application describes an specific program that can run on a specific hardware using a specific board support package (BSP). When creating a new application one must select the hardware platform to run the application as well as a BSP. Alternatively, a new BSP may be created automatically when you create a new application.

Once the XSDK starts, the hardware platform project is already available. Let us now create an simple application (and, of course, a required BSP). In the XSDK menu select **File→ New → Other ...**. Select then **Xilinx → Xilinx C Project** from the type of projects.

In the following pop-up, you will have to give the name, select a template application and a BSP for your new application, along to the processor to run it on (if several are available). Select for instance *Hello World* as a template, *standalone* (no OS support), and a new BSP (you may reuse this for other projects), and keep the defaults for any other parameters. Once you close the pop-up, automatic generation starts, which will take a couple of moments.

Two new projects should now appear in your **Project Explorer**, one for the application, and one for the BSP. Feel free to inspect the settings for both. Note that the framework is based on Eclipse, so the way of compiling projects, examining its output and the source files should already be familiar to you.

4 Running the System on the FPGA

At this point both the hardware and the software specifications of your system are ready. This system will be emulated on the Nexys-3 board, which contains a Spartan-6 FPGA.

note You should be aware that both what we refer to as hardware and software are in this case implemented as one FPGA configuration (ultimately memory contents). The FPGA thus acts as your hardware, which runs your software. A true silicon flow, which produces an ASIC from your hardware, would take a far longer turn around time, but should produce a much faster and smaller system.

4.1 Connecting through the USB-UART

When you run your system it is nice to see some output, i.e. the result of `print()`. Nexys-3 uses a USB based serial communication, a USB-UART. The hardware we created earlier maps the Nexys-3 physical pins that control the UART to signals used inside our system, by the `axi_uartlite` core. The BSP created earlier maps the C `stdin` and `stdout` to the `axi_uartlite` registers, and also uses the right libraries for communicating via the serial. Finally the application we created outputs *Hello World* via a `print()` instruction. In brief, all is set for our board to output something through the USB-UART. We now only have to display these messages.

For this reason, one can use any terminal program outside the XSDK, or use the XSDK built in terminal capability, via **Window→ Show View→ Terminal**. Push the connect button (right toolbar on the Terminal tab) and select *Serial* in the connection type, and make sure the following parameters are set for the serial (the port may vary however):

Port	COM3 **(may differ on your machine!)**
Bits per second	9600
Data bits	8
Stop bits	1
Parity	None
Flow control	None

This must in fact be the same as the configuration the one used for the `axi_uartlite` core in the hardware architecture. You should now be able to see any output produced through the USB-UART cable.

4.2 Configuring the FPGA

Now you are about to run the system and you need a FPGA board, connect it (you need two micro-USB cables, one for configuring and powering the board and one for the USB-UART) and switch on the power.

Although there are number of ways to download the configuration on the FPGA, the simplest way to do this is through the XSDK, via **Xilinx Tools→ Program FPGA**. In the **Program FPGA** pop-up, you have to specify three parameters.

Bitstream is the hardware generated by the XPS. (this should be already set properly)

BMM File is the memory map of the on-chip RAM. This file basically describes how the RAM blocks are organized into a contiguous memory space. (this should also be set properly already)

Software Configuration allows you to specify which executable is executed on which processor. The `bootloop` is basically an empty endless loop. Here you should choose the `.elf` file of your application.

Pushing *Program* will download the configuration (hardware and software) onto the FPGA (see **note** if this fails), and directly start executing your program (the `.elf` you specified earlier). You can check the output of your program in the **Terminal** tab, lower part of the window. If you used the *Hello World* template, you should directly see the `Hello World` message in this window.

note If configuring the FPGA through the XSDK fails, there is an alternative way to download the configuration onto the FPGA. This uses the tools provided by *Digilent*, in particular **Adept**. Find and run **Adept** via the Windows start menu. The tool should automatically identify your board as Nexys-3 and display in its main window the FPGA chip located on the board. You can now assign a configuration for the FPGA, located in your XSDK project directory, hardware platform folder, called `download.bit` by using the *Browse* button. Note that you might run into another `bit` file in the same directory, `system.bit`. This is not a complete configuration, since it does not contain the software. In order to obtain the `download.bit` file, you may need to attempt to configure the board via the XSDK first, which will build the `download.bit` automatically. You should now be able to configure the FPGA via the *Program* button.

Now push the **B8** button on the Nexys-3 board (the middle button, out of the five buttons group). This is assigned as *reset* button, which will cause the program to run again, producing yet another `Hello World`.

Congratulations, you have just built and ran your first (?) Nexys-3 system!

5 That was easy, what's next?

You should have by now created and run your first system. Use the remaining time in the first lab to explore the XPS program. The following labs will contain much more work and you will not be able to finish unless you have prepared ahead, i.e. created the system. In the next lab you will need a system consisting of a MicroBlaze, a UART (axi_uartlite) and a timer (axi_timer). If you have time you can start to create this system. Read the documentation of the timer. To find the documentation, right click on it in the *IP Catalog* tab in the left XPS frame.

6 Power and energy consumption

Once you have your system, you can estimate its power and energy consumption. In embedded systems (and not only) power and energy are very important since they determine the cooling, power supply and battery lifetime.

To get a rough estimate of your system's power (meaning the power consumption of the FPGA emulating it) you can use the XPower Analyzer tool that comes with Xilinx ISE, found in **start→ all programs→ Xilinx Design Tools→ ISE Design Suite 14.x→ ISE Design Tools→ 64-bit Tools→ XPower Analyzer**. Once XPower starts, choose **File→ Open Design** to specify the design you want to examine. In the dialog that appears, the design file is necessary, which is the `system.ncd` file located in the your XPS project directory, in `implementation` subdirectory. This file is the result of hardware synthesis, so it will only be present if you went through the synthesis step already. You should at least specify the Physical Constraints File, `system.pcf`, present in the same directory.

note To make the analysis more accurate, more of the optional files should be provided.

Choose OK to start the analysis, and pay attention to the output comments. Also inspect the tables produced in the main window, since these gather the power figures for your design.

Once you have the total power for the design, you may multiply this with the time it takes to execute any of your applications (you will need to use a timer core and instrument the program to record the number of clock cycles used) in order to obtain the energy consumption for that program.

7 Dual-processor systems

Let's continue by creating a dual-processor system, that will be needed in lab 3. You may either extend the uni-processor system you are currently using, or extend a dual-processor system created with the wizard you used before (please see the final section for that). The first solution focuses on using FIFOs to communicate between processors, while the second uses shared memory/mutexes or mailboxes to the same end. These mechanisms do not exclude each other and can be used at the same time, granted your system contains the hardware support. Finally by the end of this section you should be able to have *three* different ways of sending data between two processors:

1. FIFOs (Fast Simplex Link)
2. mutex/shared memory (Micron_RAM)
3. mailbox

7.1 Starting from a single processor system

IN XPS, start by creating a system containing only one MicroBlaze and a UART using the BSB wizard (or continue working with the system you just created). To add a second processor you need to manually duplicate all components and

parameters in the processor/memory subsystem. Look at the block diagram to see how the memory and CPU are connected (CPU↔ LMB↔ LMB controller↔ memory block). Notice that MicroBlaze uses a Harvard architecture, where data and instructions have separate pathways. On the other hand, the memory blocks are dual ported, meaning that you can connect both the instruction and data bus to the same memory block. To add IP components to your design, select the IP from the *IP Catalog* tab in the left XPS frame, and double-click on it. The newly added IP should appear in the *System Assembly View* tab, where you can also rename the instance. Now add the following:

1x `Processor→microblaze`, which will automatically add and connect the following:

  1x `Memory Block→bram_block`,

  2x `Memory Controllers→lmb_bram_if_ctrl`,

  2x `Buses→lmb_v10`,

2x `Fast Simplex Link Buses→fsl_v20`.

The component associated with the new processor should match match the way your first processor is connected. This means that the memory block should be connected to both controllers. The two controllers should be connected each to its own LMB. The processor should be connected to both LMBs (one for data, one for instructions). The processor should also be connected to the common AXI (or PLB) bus. You can check this in the *System Assembly View* tab. Make sure that the *Bus Interface* tab is selected. All buses are shown to the left (vertical lines). Simply click on a circle/box to connect/disconnect a component instance to a bus. One can only connect ports on an instance, so click on the + to see the component ports first.

The Fast Simplex Link (FSL) is a uni-directional point-to-point connection which will be used to communicate directly between the processors. To use them, you must add FSL ports to the processors. To do this, double click on the processor in *System Assembly View*, which will bring up the MicroBlaze **Configuration Wizard**. In the pop-up, push the *Advanced* button, then locate and select the *Buses* tab. Here you can select the number of *Stream Links*, where each link is a duplex link (which means, each needs two FSL buses). Make sure the *Stream Interface* is set to FSL (not AXI). You may now close the pop-up via *OK*. At this point, your processor should have two extra interfaces, called SFSL0 and MFSL0 (standing for slave and master FSL), if you expand its interfaces (click *+*) in the *System Assembly View* tab. You can now connect these interfaces to the two FSL buses you added previously. Repeat the procedure for the second processor, in order to connect the two processors via the FSL buses.

Your system should be looking something like in Figure 1 (obtained by **Project→ Generate Block Diagram Image**).

You also need to connect the clock and reset signals for any IPs you have added to your design. Switch from the *Bus Interfaces* to *Ports* tab in *System Assembly View*. First connect the reset signal of the newly added processor to the `proc_sys_reset_0::MB_Reset` signal (the same as the initial processor). Connect the two FSLs you added to the proper clock and reset nets. It suffices to link `FSL_Clk` to `clock_generator_0::CLOCKOUT0` and `SYS_Rst` to

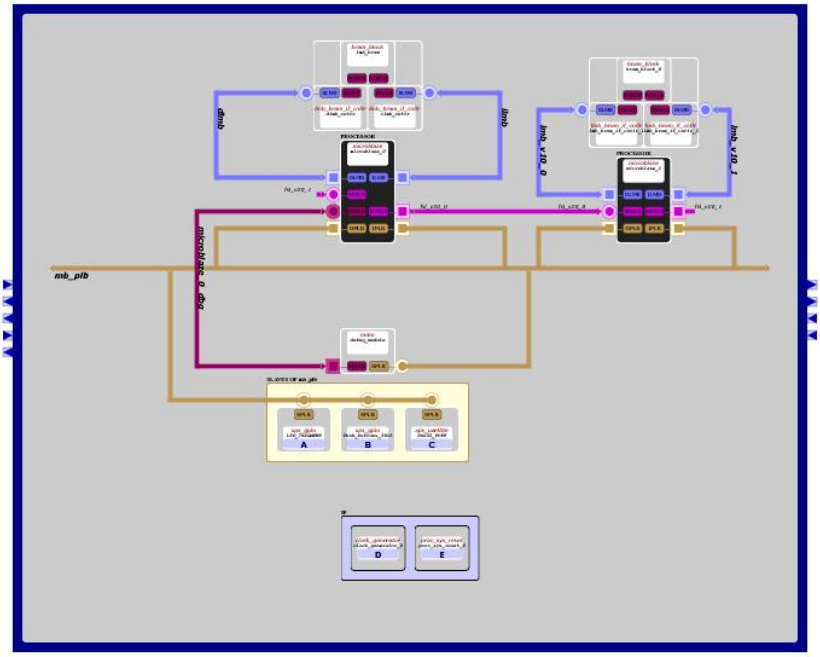

Figure 1: Dual MicroBlaze system view

`proc_sys_reset_0::Bus_Struct_Reset`: select the *Connected Port* for the signal you want to connect and choose the appropriate signal from the drop list. Connect the two LMB buses you added similarly to clock and reset. Note that names may differ in your tool version, so the best hint is to examine how the initial IPs are connected to signals (e.g. `ilmb`).

note Making sure all the new cores are connected to the clock and reset signals is essential. Unconnected signals may not be easy to detect during synthesis, and your system might not function properly, or at all.

Moving on to the memory layout of the newly added processor, in *System Assembly View*, switch the tab from *Ports* to *Addresses*. This gives an overview of the address map of the whole system, how each processor sees the peripherals and memory. To make sure all the peripherals are mapped to some memory address, you can generate this address map automatically, by clicking on the small button located in the upper right corner, the same level with the *Addresses* tab (if the mouse pointer hovers over it, it says *Generate Addresses*).

note Peripherals with unassigned addresses are usually bad news, since they cannot be addressed from anywhere. If after the automatic generation of addresses there are still such peripherals, it can only mean that they are not connected to a bus (AXI or PLB).

note Note that the address spaces for each peripheral cover a certain memory size. In most cases the same registers from a peripheral are mapped to a number of addresses inside this space. For the `ilmb/dlmb` controllers,

however, these sizes directly control how much memory your program has available to use. By changing these sizes, you can give more memory to your software, up to a limit imposed by the resources available on the FPGA.

note If you want to access the same peripheral from the two different processors, you have to make sure it is connected somehow to both processors. You may choose, for instance, to connect both processors to the same system bus (AXI or PLB).

At this point the dual-processor hardware architecture should be set properly. You should now synthesise and export this to XSDK.

7.2 Dual-processor software

The XSDK flow for two processors looks similar to the flow for one processor, so we will not detail it further here.

You might want to display messages from your second processor, which means setting its STDIN/STDOUT to the UART. If you already created your application for the new processor, you will have a BSP associated with it. Right click on the BSP project associated with the new processor and select *Board Support Package Settings*, which will bring up a pop-up. Select the *standalone* item from the list, and choose the right peripheral for the STDIN/STDOUT. If you cannot select your UART, it might mean that the processor does not have physical access to it, e.g. the UART is not connected to the processor via a bus (AXI or PLB).

The system should now be ready to run. You should download it, run it, and check its output.

7.3 FSL communication

Once you have got both CPUs running (messages from both show up on the terminal), it is time to do something useful. Let's make them talk to each other, through the FSL. Macros for reading and writing from/to the FSL 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 FSL port number, starting from 0 (both for read and write). An example application (assuming there are FSLs in both directions):

MicroBlaze 0:

```
#include <mb_interface.h>
#include <xutil.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>
#include <xutil.h>
int main(void){
  int i;
  while(1){
    getfsl(i, 0);
    xil_printf("ping %d\n\r", i++);
    putfsl(i, 0);
  }
}
```

7.4 Using the wizard to create a dual-processor system

When you created your single processor system using the wizard in XPS, at one point you were asked to choose between single processor or dual-processor systems. At the same step you may choose dual-processor system to create an architecture that contains two processors, a few peripherals, including a mailbox, a mutex and a Micron_RAM cores. At the time of writing this tutorial, the AXI flow for the dual-processor system did not work, but the PLB flow did, so you might want to use the latter.

In order to see an example of using the mutex and mailbox from your software, you can use the XSDK template applications, when adding a new project. The *Peripheral Tests* template will detect mailboxes or mutexes connected to your system bus and add code for these. Have a good look at this code, if you intend to use mailboxes, or mutex/memory instead of FSLs in lab 3.