Hardware/Software Codesign II
Summer Term 2015

TECHNISCHE UNIVERSITÄT CHEMNITZ

Prof. Dr. Wolfram Hardt, Dipl.-Inf. Michael Nagler

Informatik

Lab 1

Tutorial Nexys 3 and Xilinx SDK

1. Introduction

In this lab, the FPGA board Nexys 3 from Digilent®with a Xilinx Spartan®-6 FPGA on it is used. The following interfaces and components will be utilised during this lab:

- Xilinx® LogiCORE IP MicroBlaze Micro Controller System[5]
- Serial RS232-Interface
- LEDs and 4-digit Seven Segment Display
- Push Buttons and Slide Switches

These development tools are used to program the FPGA device:

- Xilinx® Integrated Software Environment (ISE™) as integrated development environment (IDE) for hardware components.
- Xilinx® Software Development Kit (SDK) as integrated development environment for software components. The SDK is built on Eclipse[1].
  Note: A free version of ISE and SDK ("WebPack") is available at `http://www.xilinx.com` (Registration is required)
- HTerm[2] as terminal for the serial interface.

For a better overview of the board, it is depicted in Figure 1 with all ports being labelled. A block diagram can be found in Figure 2.

2. Requirements

In order to work through the lab in a feasible time, the following skills are required:

- Knowledge of VHDL and hardware design methods (a good basis is the lecture "HW Development with VHDL")
- Knowlegde of C (in the lab a GCC-C-Compiler and an Xilinx®adapted GCC is used)

[1] `https://eclipse.org/`
[2] `http://www.der-hammer.info/terminal/`

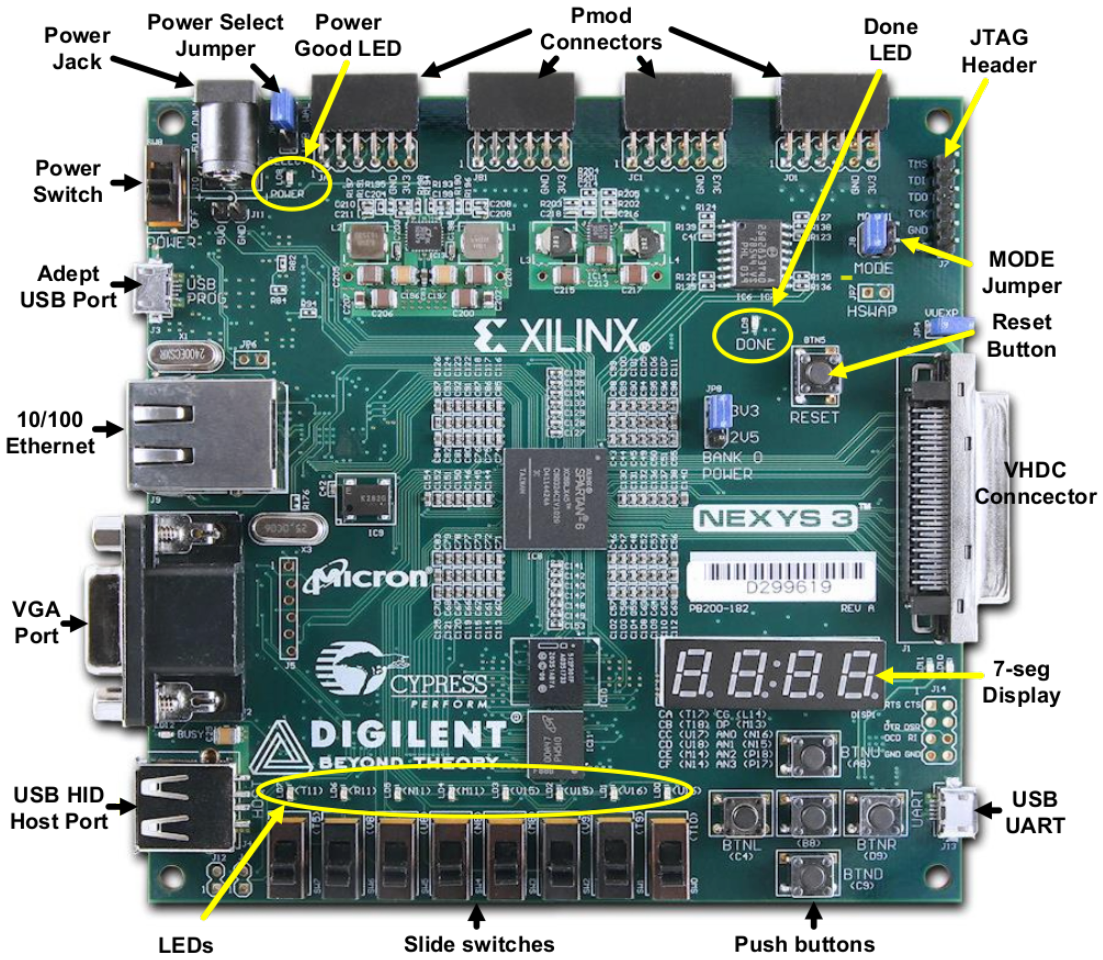


Figure 1: Depiction of the Board (from Nexys3™ Board Reference Manual [1])

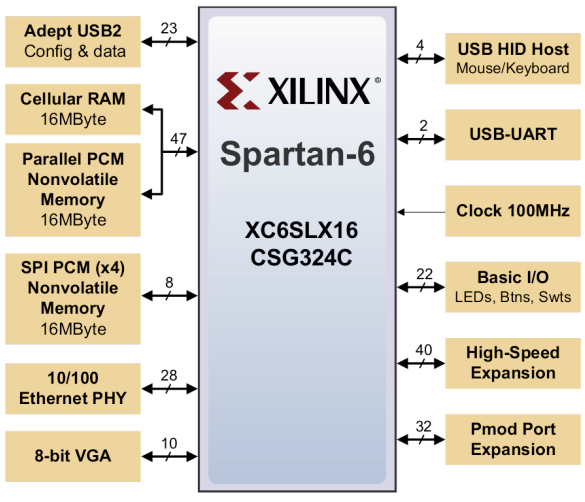


Figure 2: Block Diagram (from Nexys3™ Board Reference Manual [1])

3. FPGA

An FPGA (Field Programmable Gate Array) is a hardware device whose behaviour can be changed by programming it. This functionality is realised using so-called LUTs (Look Up Tables) and wiring between. LUTs are memories that store boolean function: The parameters to the boolean function are taken as address and the result of the function can be read from the storage element selected by this address. LUTs have more than two inputs (currently four to six inputs are common) so that more complex functions than e.g. `AND`, `OR` and `NOT` can be realised.

LUTs are part of CLBs (Configurable Logic Block). These CLBs can be connected dynamically.

The FPGA can also be partitioned into slices that constitute larger areas. These slices can be programmed independently, which enables partial reconfiguration of the FPGA. Current research addresses this reconfiguration.

Essential to an FPGA is the organisation as an array of CLBs. Memory technologies used in the LUTs and for the interconnections are amongst others SRAM, flash, and other non-rewritable technologies [2]. This means that there are also FPGAs that can only be programmed once.

The programming of the FPGA consists of writing the content of the LUTs and setting up the connections between LUTs and CLBs.

In this lab SRAM-based FPGAs are used. So in case of a power fail it loses its configuration. But it is programmable nearly unlimited times. Often another hardware device (ROM) is used from which the FPGA loads its configuration after turning on the power.

FPGAs are used especially in hardware design, because the designed hardware can be tested in a very early design stage with relatively high performance, which is not possible with simulation.

4. Outline of the Lab

The goal of this lab is to control a robotic arm. This implementation task is devided into three parts which make up three of the four labs. The following outline illustrates how the lab evolves:

1. Introduction to the board and the used development environment.
2. Implementation of a finite state machine for a serial RS232-interface.
3. Implementation of a software control on the Nexys3 for the robotic arm.
4. Implementation of a software that checks for collisions for the robotic arm.

5. Lab

In this part, basic knowledge and methods concerning the work with the Xilinx®Integrated Software Environment (ISE) and the Xilinx® Software Development Kit (SDK) are introduced. In later parts of the lab, this knowledge will be used, since a repetition would make the lab tutorials unnecessarily long.

5.1. Prepared Running Light

To demonstrate the operation of the FPGA board, the first step is to load a completely prepared running light onto the board. The project consists of three parts:

1. The top module: `top.vhd`. This module defines the interface (pinning) of the design and creates an instance of the LED component that realizes the running light.
2. The LED component: `led.ngc`. This component is given as a synthesized netlist and realizes the running light on the LEDs.
3. The constraints: `top.ucf`. This file realizes the connection between the virtual ports of the interface (VHDL file) and the physical pins of the FPGA.

The following steps are necessary to synthesize the project and to download it onto the board:

1. Turn on the power supply of the board so the board is ready to receive a configuration from the workstation.
2. Create a folder/directory `Lab1`. It is important that there is ***no white space in the path name***. It is also recommended to store it on a local hard disk because the network connection is too slow for working at a good pace.
   Hint: *Create the folder in your user directory. Remember to copy back all your data to your computer center home (drive H:) before you logout. So you will be able to access your data at home or when you login on another computer.*
3. Extract the archive `Task1.zip`
4. Start „Xilinx® ISE Project Navigator“.
5. Open the file `running_light.xise` using „`File` → `Open Project`“. This will open the project with the prepared running light.
6. In the project hierarchy select the top module (`top - Behavioural (top.vhd)`) by clicking on it.
7. In the window below run “`Configure Target Device`” by double-clicking it. This will synthesize the design, create a flashable bitstream and download it onto the board. ***Hint:*** *This step will take some time.*
8. When the generation of the bitstream is finished, a warning pops up, saying the *iMPACT* project file was found. After clicking *ok*, the *iMPACT* tool is started.
9. Double-click `Boundary Scan` in the left side of the *iMPACT* window.
10. Right-click in the main tab and select “`Initialize Chain`” (or use `CTRL+i`).

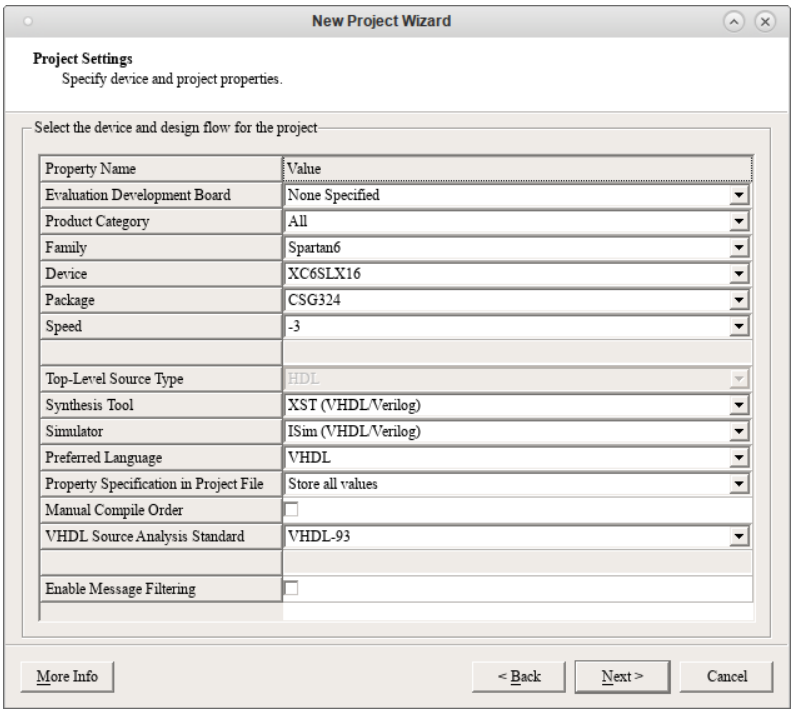
11. Click *Yes*
12. Select the bitstream `top.bit`.
13. Click *No*
14. Click *OK*
15. Right-click the FPGA symbol and select "`Program`"
16. When the orange *DONE-Led* lights up, hit the reset button `BTND (C9)` to start the running light.
17. Hold down button `BTNU (A8)` to reverse the direction of the running light.
18. In order to close the project select in the menu "`File` → `Close Project`"

5.2. Running Light in Hardware

Now the running light should be realized in hardware. The description language used in the lab is VHDL. With the Xilinx® ISE™ new hardware projects can be created and instantiated on the board.

First, a new hardware project is created:

1. Create a new folder/directory `Task2`.
2. In the menu select "`File` → `New Project`". Name it `Task2` and set as location the created folder `Task2`.
3. Click *Next*
4. Project Settings:



5. Click *Next*
6. Click *Finish*

Now right-click into the project hierarchy and create new source files. At this point two new files are needed:

1. **Create a VHDL file:**
   a) `[right-click]` → "`New Source...`" and select "`VHDL Module`"
   b) Enter a filename (e.g. `top.vhd`)
   c) Click *Next*
   d) Define the entity of the VHDL file:
   ```
   clk       in
   rst       in
   button    in
   led       out (Bus:  MSB = 7, LSB = 0)
   ```
   e) Click *Next*
   f) Click *Finish*
2. **Create a constraint file:**
   a) `[right-click]` → "`New Source...`" and select "`Implementation Constraints File`"
   b) Enter a filename (e.g. `top.ucf`)
   c) Click *Next*
   d) Click *Finish*

Now the behaviour of the running light has to be described using VHDL (`top.vhd`). To complete the design the virtual signals have to be connected to the physical pins of the board using the file `top.ucf`. The pin locations can be found on the board or in the master UCF (User Constraint File) on the Digilent® Website[3]. To test the design, select the top module in the hierarchy view and run "`Configure Target Device`" in the window below by double-clicking it.

Hint: *The EDK generates several log files during the synthesis. In these files, warning and error messages are stored. Furthermore, there are also summaries about how many FPGA resources are used and how many are available.*

If an error occurs, the error message on the screen contains an information saying that a more detailed description of the error can be found in the log file. The path to the log file is also given.

The log files can easily accessed in the "`Design Summary`" tab. If it is not opened, double-click "`Design Summary/Reports`" on the left side of the screen.

[3] https://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,897&Prod=NEXYS3

5.3. Running Light in Software

In the next step, the running light should be realized in software. Therefore, a new project is created and the MicroBlaze Micro Controller IP Core is added to it. Afterwards a software project will be created using the Xilinx® SDK.

5.3.1. Configuration of the Hardware Components

To be able to run a software component on the board, the hardware system, on which the software will finally be executed, is synthesized first. To do so, a new project `Task3` with a top module and a constraint file has to be created, as described in Section 5.2. Then the MicroBlaze IP Core is created using the Xilinx® Coregenerator:

1. `[right-click]` → "`New Source...`" and select "`IP (CORE Generator ...)`"
2. Enter a filename (e.g. `microblaze_mcs`)
3. Click *Next*
4. Select "`Embedded Processing` → `Processor` → `Microblaze MCS`"
5. Click *Next*
6. Click *Finish*

Then the generated IP core has to be configured using the dialog window that pops up when *finish* is clicked:

1. Tab *MCS*
   - Instance Hierarchical Design Name: `mcs_0`
     Hint: *This name has to be used when creating an instance of the IP core in the design (VHDL file).*
   - Input Clock Frequency: 100.0 Mhz
   - Memory Size: 16 kb
2. Tab *UART*
   - Enable Transmitter
   - Baud Rate: 9600 bps
   - Data Bits: 8
   - No Parity
3. Tab *GPO*
   - Enable General Purpose Output 1.
   - Number of Bits: 8 (for the LEDs)
   - Initial Value: `0x00000000`
4. Tab *Interrupts*
   - Use External Interrupts

- Number of External Interrupts: 1 (for timeout between LED update)
- Level or Edge: `0x0000`
- Positive or Negative: `0xFFFF`

Hint: *The meaning of the values for "Level or Edge" and "Positive or Negative" can be obtained by hovering the input fields.*

After the configuration of the IP core is done, click on *Generate*. When the generation is finished (this step will take some time), select it in the hierarchy view and run "`View HDL Instantiation Template`" in the window below. This will show an example for the *component declaration* and an *instantiation template* for the design (VHDL file). In the instantiation template the instance name `your_instance_name` has to be replaced with the *Instance Hierarchical Design Name* specified during the IP core configuration. The code templates are shown in listing 1.

Listing 1: Code Templates for MicroBlaze MCS IP Core

```
1  -------------------------------
2  -- Component Declaration
3  -------------------------------
4
5  COMPONENT microblaze_mcs
6    PORT (
7      Clk            : IN  STD_LOGIC;                     -- 100 Mhz Clock
8      Reset          : IN  STD_LOGIC;                     -- Reset (high active)
9      UART_Tx        : OUT STD_LOGIC;                     -- Pin for serial Transmitter
10     GPO1           : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Output Reg for LEDs
11     INTC_Interrupt : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);  -- Interrupt for LED Update
12     INTC_IRQ       : OUT STD_LOGIC                      -- Interrupt Received
13   );
14 END COMPONENT;
15
16 -------------------------------
17 -- Instantiation Template
18 -------------------------------
19
20 your_instance_name : microblaze_mcs -- change name of instance
21   PORT MAP (
22     Clk            => Clk,
23     Reset          => Reset,
24     UART_Tx        => UART_Tx,
25     GPO1           => GPO1,
26     INTC_Interrupt => INTC_Interrupt,
27     INTC_IRQ       => INTC_IRQ
28   );
```

With these code templates the hardware design can be finished. Connect all signals (Clock, Reset, UART) as well as the LEDs (General Purpose Output Register) to the appropriate ports of the MicroBlaze Core. Then create the interrupt signal every second. This signal has to be asserted until the reception of the interrupt is confirmed (`INTC_IRQ`). The reverse direction of the running light can be realized in VHDL as done in 5.2.

Hint: *The configuration of the IP core is saved in the file* ***`microblaze_mcs.xco`****. It is located in the project directory under* ***`ipcore_dir`****. This file is also added to the design automatically. It can also be used to reconfigure/recreate the core by double-clicking it inside the hierarchy view.*

5.3.2. Integration of Software Components

In order to be able to run software on the generated IP core, a Xilinx® SDK project has to be created for it. This is done in the following steps:

Create Board Support Package:

1. Create a folder/directory `workspace`.
2. Start "Xilinx® Software Development Kit".
3. Select the created directory as workspace location and click *ok*.
4. Close the welcome screen tab so the Project Explorer opens.
5. Create a Board Support "`File` → `New` → `Board Support Package`
6. Click *Specify*
7. Click *Browse* (Target Hardware Specification) and select the hardware specification file located in `ipcore_dir` (e.g. `microblaze_mcs_sdk.xml`).
8. Click *Finish*
9. Select `standalone` (Board Support Package) in the next dialog window and click *finish*.
10. Click *OK*

Create Software Project:

1. Click on "`File` → `New` → `Application Project`"
2. Give it a project name (e.g. `running_light`).
3. Use existing Board Support Package: `standalone_bsp_0`
4. Click *Next*
5. Select Template: *Hello World*
6. Click *Finish*

Now a software project with an example application, that sends the string *"Hello World"* over the serial interface, is created. It is displayed in the *Project Explorer* on the left side of the screen. This project can now be adapted, so it can be used to realize the running light. Therefore the following steps have to be done (see listing 2):

1. Create an Interrupt Service Routine (ISR) to handle the nterrupts.
2. Initialize the IO module to have access to the general purpose registers and the interrupts.
3. Register the intterupt handler for the MircoBlaze Micro Controller.
4. Connect and enable the created ISR.
5. Enable interrupts on the MircoBlaze Micro Controller.

The C code in Listing 2 can be used for the running light. Only the update of the data still has to be done. The IDs are defined as follows: The Device ID of the IO Module is defined

in the file `xparameters.h`. The interrupt ID depends on the priority of the interrupt. The MicroBlaze Micro Controller has up to 16 interrupts. The highest one (16) corresponds to bit 0 of the interrupt vector (`INTC_Interrupt`) in the VHDL description. Interrupt 15 corresponds to bit 1 and so on. The GPO Register ID `LED_REGISTER_ID` depends on the MicroBlaze configuration (see Section 5.3.1). As *General Purpose Register 1* was selected, the ID has to be 1, too.

Listing 2: Example C Code for the Running Light in Software

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xiomodule.h"

#define IO_MODULE_ID          XPAR_IOMODULE_0_DEVICE_ID
#define LED_INTERRUPT_ID      16
#define LED_REGISTER_ID       1

// IO Module for GPI, GPO, Interrupt
XIOModule ioModule;

// Variable holding current LED Data
u8 led_data = 0x0f;

void print(char *str);

// the ISR to handle the interrupt (STEP 1)
void ledUpdateISR ( void )
{
    // output to serial interface
    print("LED Data updated ...\n\r");

    // TODO: shift data

    // write data
    XIOModule_DiscreteWrite(&ioModule, LED_REGISTER_ID, led_data);
}

int main()
{
    init_platform();

    // Initialize (STEP 2)
    XIOModule_Initialize(&ioModule, IO_MODULE_ID);

    // Register Interrupthandler (STEP 3)
    microblaze_register_handler(XIOModule_DeviceInterruptHandler, IO_MODULE_ID);

    // Connect/Enable ISR for Seven Segment Display (STEP 4)
    XIOModule_Connect(&ioModule, LED_INTERRUPT_ID, ledUpdateISR, IO_MODULE_ID);
    XIOModule_Enable(&ioModule, LED_INTERRUPT_ID);

    // enable Interrupts (STEP 5)
    microblaze_enable_interrupts();

    // stay in main function
    while (1);

    cleanup_platform();

    return 0;
}
```

Hint: *In the upcoming labs it is also necessary to read values from the GPI Registers. The function* `XIOModule_DiscreteRead(&ioModule, GPI_ID)` *returns a 32-bit value that holds the read data. If the register is configured to have less than 32 bits, the remaining bits are filled up with zeros. The* `GPI_ID` *also depends on the MicroBlaze configuration*

(see 5.3.1).

5.4. Building the Project

In order to be able to run the software on the MicroBlaze Micro Controller, the ISE project has to be modified by hand. First, the Block Memory Map (BMM file) has to be loaded during the implementation of the design. This file is generated during the generation of the MircoBlaze IP Core and can be found in the folder `ipcore_dir`. It defines both type and amount of memory components that have to be used to realize the configured memory size of the micro controller and how the memory is mapped on these components:

1. Right-click "`Implement Design` → `Process Properties...`"
2. Property display level: *Advanced*
3. Category: *Translate Properties*
4. Property Name: *Other Ngdbuild Command Line Options* → `-bm` "`ipcore_dir/microblaze_mcs.bmm`"

The second step is to load the compiled ELF file during the bitstream generation:

1. Right-click "`Generate Programming File` → `Process Properties...`"
2. Property display level: *Advanced*
3. Category: *General Options*
4. Property Name: *Other Bitgen Command Line Options* → `-bd` "`workspace/running_light/Debug/running_light.elf`"

After these two steps are done, double-click on `Configure Target Device` so the design is synthesized and flashed on the FPGA. After pressing the reset button the running light should start.

Note: *The ELF file is regenerated automatically when a source is in the SDK is modified and saved. After that the bitstream in the ISE has to be updated: right-click on* ***`Generate Programming File`*** → ***`ReRun`****.*

6. Tasks

6.1. Start Using the Tools

Retrace the steps given in the three tutorials in Section 5.

6.2. Change the speed of the running light (HW)

Change the speed of the running light realized in hardware. Add two new buttons and double the speed when the first button is pressed or half the speed when the second button is pressed. ***Hint:*** *The button has to be debounced.*

6.3. Change the initial value of the running light (SW)

Add the switches to set the initial value of the running light. ***Hint:*** *The software of the MircoBlaze Micro Controller is restarted when the reset button is pressed.*

References

[1] Digilent, Inc.: *Nexys 3 reference manual*, April 2013. Link.

[2] Herrmann, Göran and Dietmar Müller: *ASIC - Entwurf und Test.* Fachbuchverlag Leipzig im Carl Hanser Verlag, 1. edition, 2004.

[3] Xilinx, Inc.: *XST User Guide for Virtex-6 and Spartan-6 Devices*, December 2009. Link.

[4] Xilinx, Inc.: *ISE In-Depth Tutorial*, April 2012. Link.

[5] Xilinx, Inc.: *LogiCORE IP MicroBlaze Micro Controller System Product Guide*, December 2012. Link.