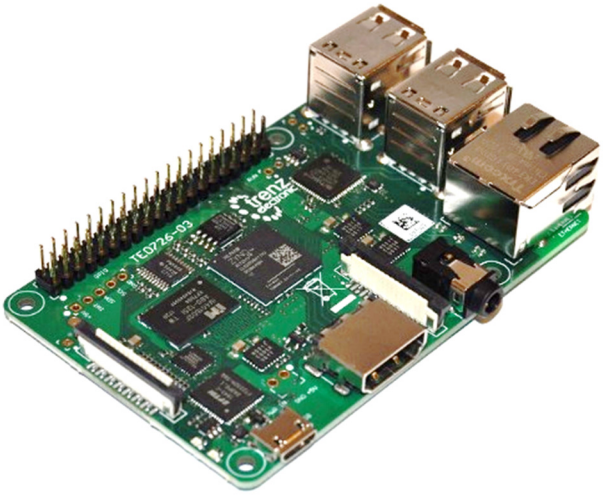
Mittuniversitetet
MID SWEDEN UNIVERSITY

Experiments on SoC design for Embedded Computing



This publication along with supporting design files can be downloaded from the course homepage,
http://apachepersonal.miun.se/~bentho/ec/index.htm

Outline

1. Experiment 1: Getting to know the development tools

1.1. Introduction

This first experiment will guide you through the process of synthesizing and implementing a reference HW design using Vivado. This reference design is originally provided by the board manufacturer Trenz Electronics GmbH and can stream video from camera to monitor. You will use a pre-built Linux operating system that is tuned to work with this custom developed HW.

1.2. The reference hardware design

A tutorial [3], user guide 981 from Xilinx can additionally guide you through the following steps:

a) Download *zynqberrydemo1* hw-design from the course home page. All design files are bundled in a zip archive. Open this design in Vivado tool and explore the set of IP-components used.

b) In addition to VHDL modelling, Vivado uses a schematic input called block design allowing users to easily connect IP-components. If you select a block design in the Vivado design file browser and right click mouse, there will be an option, *Generate output products*. This means that the tool automatically generates a VHDL structural model for the connectivity as described in the block design. This operation must be performed on block design *zsys.bd*

c) Right click on *zsys.bd* once again and select *Create HDL Wrapper …* This function will generate a top-level VHDL file for your whole design.

d) Processing of your complete VHDL model is divided into three steps: *Synthesis*, *Implementation* and *Programming* for generation of a bit stream. You should be familiar with these steps from previous course module on VHDL modelling.

e) From *File* menu in Vivado, select *Export Hardware*. Make sure to select *Include Bitstream*. This operation will create a workspace for software development in SDK and a hardware description file along with the bit stream will be included.

1.3. Prepare ZynqBerry for Linux OS

Firstly, you will program the onboard FPGA configuration memory with boot loaders and logic configuration. Secondly, you will prepare a flash memory card with a bootable Linux image.

a) Download file *BOOT.BIN* from course home page.

b) Connect ZynqBerry to your host computer using an usb-cable. Open hardware manager located on the left down side of Vivado panel. Press “Open Target” and then “Auto Connect”. You should now see a symbol labelled xc7z010, equal to the part number of the on board Zynq platform. See Figure 1A. Right click this symbol and press “Add Configuration Memory Device”. Choose *s25fl128s-3.3v-qspi-x4-single* as the onboard memory device. On the panel for configuration memory device properties, you should select *BOOT.BIN,* previously downloaded. Right click the symbol for the spi flash memory and start programming procedure. You should be familiar with this programming operation from previous course module VHDL but if you need further guidance, please read Xilinx user guide [X].

c) Download *bootimage.7z* from course home page. Unzip this archive and copy all files to the micro flash card that fits into the card holder of the ZynqBerry board. Disconnect usb-cable to ZynqBerry and attach flash card into the holder.

1.4. Start your embedded Linux computer

You will now start up your Linux embedded computer and view streaming video from the Rasberry Pi camera to a monitor. You will log on to your computer using console line commands.

a) Connect an hdmi video cable from ZynqBerry to a video monitor. Connect an usb-cable to your host computer. An image will be seen on the video screen while ZynqBerry is booting Linux from flash memory card. After some time, real-time video will start to stream from the RaspBerry Pi camera to the video monitor.

b) When you connect the usb-cable from ZynqBerry to your computer, an additional com-port will appear. Keep in mind that com-ports will show up as file devices in your Linux workstation. To know the number of that port, issue command *ls /dev/ttyUSB** on your workstation, use this command both before and after your ZynqBerry has been connected to the usb-port. This will reveal which file device is mapped to communication with ZynqBerry. From the *File* menu in Vivado design suite, select *Launch SDK*. The graphical interface for the software development tools will show up on your screen. Find the *SDK Terminal* window. See Figure 1D. Click the green +button and connect to one of the com-ports showing up when you connected the usb-cable (Baudrate *115200*, Data bits *8,* Stop bits *1*, Parity *None*, Flow control *None*). See Figure 1C. Press return key within the terminal window and you should see a system prompt on the terminal,

plnx_arm login:
Type *root* for user name and then *root* again as password.
You are now logged in and you should be able to explore the file system by typing commands *cd /* and then *ls*. You will see a file structure that is typical for a Linux system. Type command *reboot* and you will be able to see what is printed during whole booting sequence.

1.5. Create your first "Hello World" application

You will now learn how to create your first program executing under Linux. You will further learn how to remotely deploy, debug and execute your Linux application on your target platform ZynqBerry. Information from Xilinx on the same topic can be found in user guide 981 and section "Debugging Zynq Applications with TCF Agent" [4].

a) In the "File" menu of SDK panel, select "New" and then "Application Project". A new panel will appear. Select a name for your new application and make sure to select Linux as your OS platform. See Figure 1B. Press button "Next" and then select the template "Linux Hello World".

b) Now is the time to hook up your ZynqBerry to Ethernet. Plug in the Ethernet cable into the corresponding connector on your ZynqBerry. In the *SDK Terminal* window, type command *ifconfig* and find out the ip-address of your ZynqBerry board. Look at Figure 2A where a green marker shows the detected ip-address. Be aware of that this ip-address might be assigned differently by a DHCP-server every time you restart ZynqBerry or re-connect it to Ethernet.

c) On the SDK panel, select "Run" menu and then "Debug configurations …". A new panel will show up. See Figure 2B. Select "System Debugger on Local" at the left side. Select "Linux Application Debug" and press "New" to set up a connection between your host computer and your ZynqBerry. Another panel shown in Figure 2D will appear. You need to fill in the same ip-address as was previously detected for your ZynqBerry. Port number should always be 1534. Before you close this panel, it is advisable to test the connection between host computer and ZynqBerry. Then press "OK" button. Figure 2C shows how name of project, local file path to compiled executable and remote file path to executable on target hardware is specified. It is enough to open the browser showing available software applications, select one application and all data will be filled in automatically. The path */mnt* on ZynqBerry is where the debugger will download the executable for debugging or running the application remotely. The debug session is finally started when you press "Debug" at the down right-side of panel. A new perspective will open where e.g. source code and console output can be viewed as you execute your code line by line. See Figure 3.

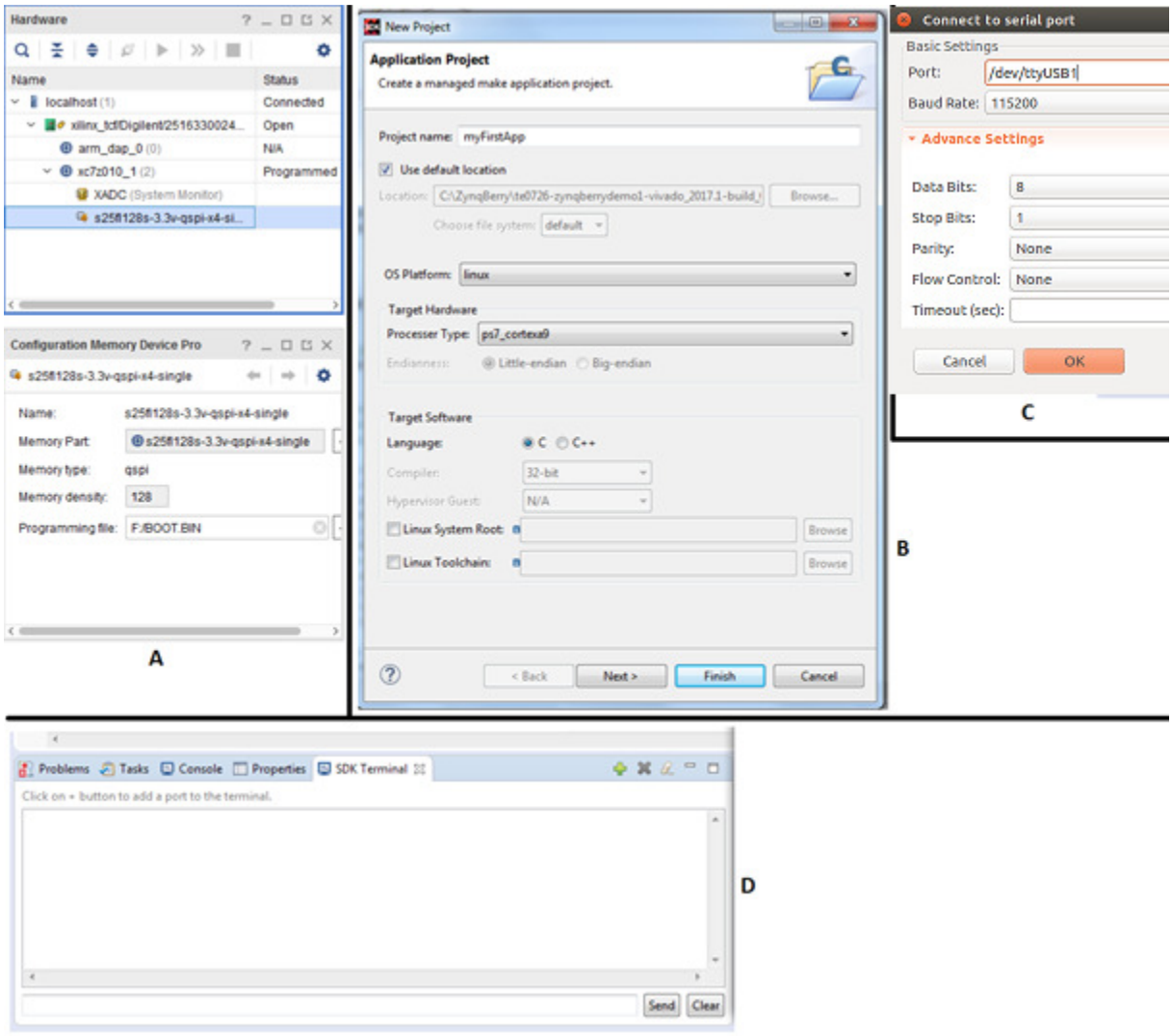


Figure 1. Selection of panel views.

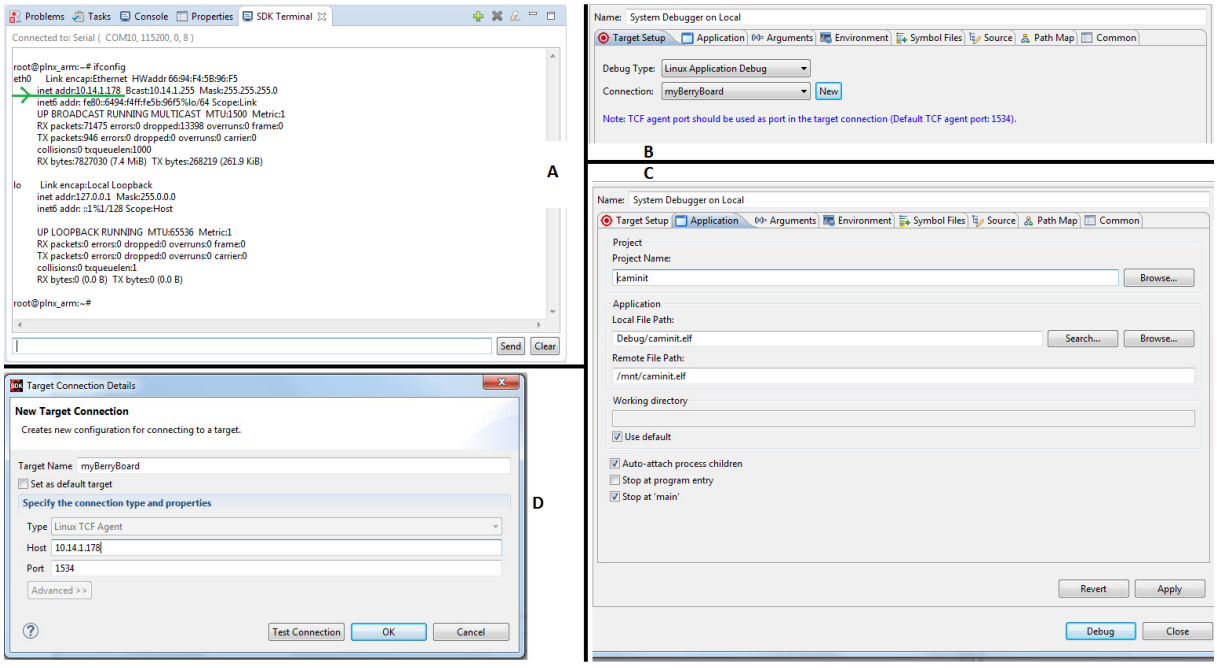


Figure 2. Selection of panel views.

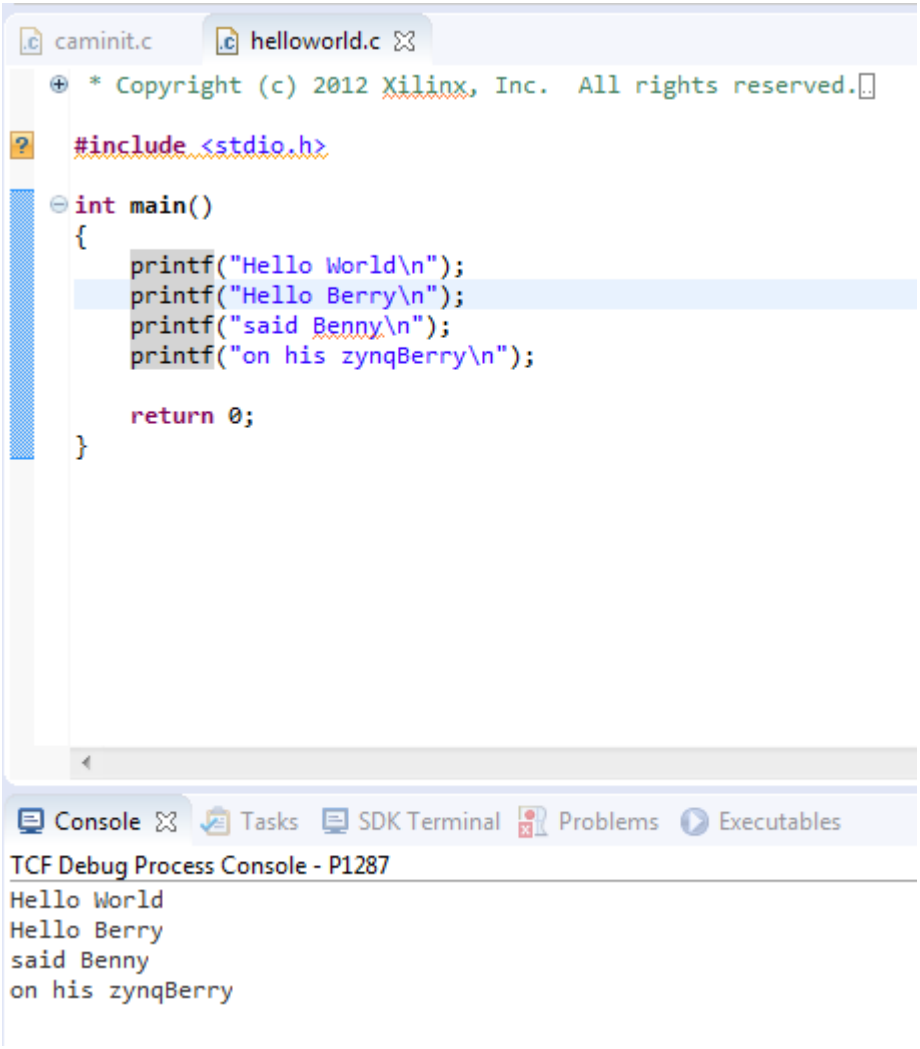


Figure 3. Debug perspective showing source code and console output.

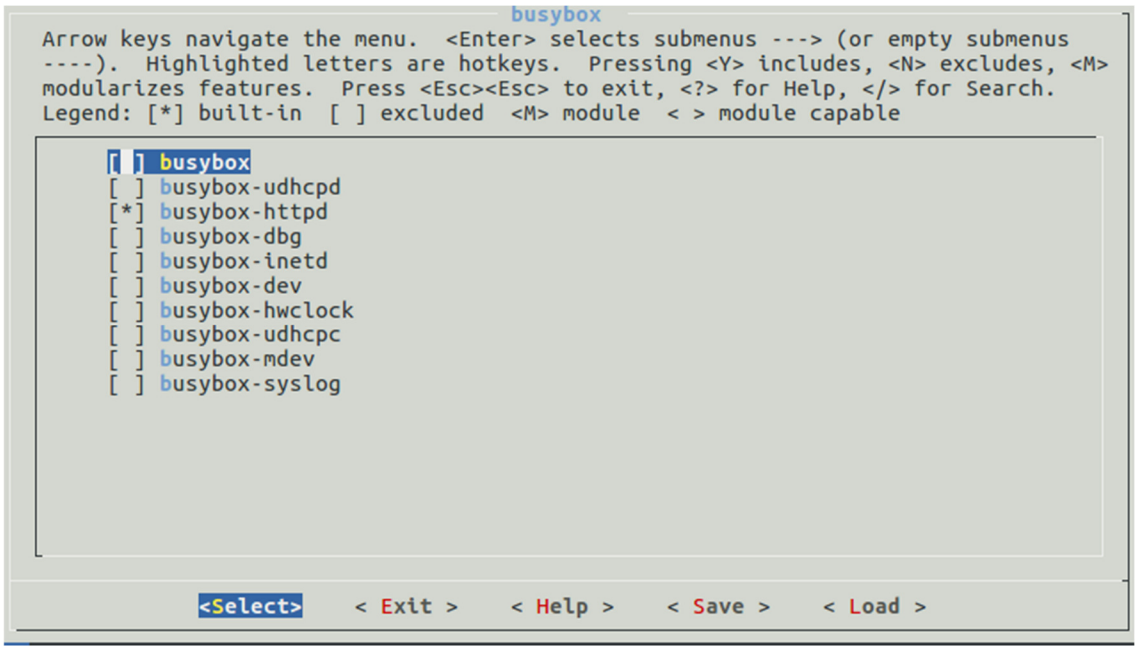


Figure 4. Preferred settings for Busybox Webb server.

Experiment 2: Configuration of the Linux system

1.6. Introduction

This experiment will guide you through the process of configuring your Linux system to host a Webb server. Petalinux tools are installed for you under Ubuntu Linux operating system. A quick reference guide for a set useful Linux commands can be found in [9]. Petalinux is a set of tools (installed under Ubuntu) used to generate a customized Linux system that just fits the need of your embedded system. Petalinux tools is a simplification of the more general Yocto method for generation of Linux systems [6]. An introductory tutorial to working with Petalinux can be found in Xilinx user guide 1156 [7] .

1.7. Create, configure and compile a Petalinux project

A board support package file carries all the necessary settings for the Petalinux tools to generate a bootable Linux system on a specific hardware. Board support package files are available on Xilinx home page for several experimental boards. A .bsp file for ZynqBerry board was generated from the demo projects supplied by Trenz Electronics.

a) Download the board support package file, *zynqBerry.bsp* for Petalinux projects on ZynqBerry. This file is listed on the course home page. Create a directory *petaProj* under your home directory and locate the downloaded .bsp file in this directory. Copy the hardware description file, *zsys_wrapper.hdf* from the Vivado project used in *Experiment 1* and locate this file in directory *~/petaProj*.

b) Issue command *cd ~/petaProj* and then command *echo $PETALINUX*. A path to the Petalinux software package should now display in the terminal window, */opt/pkg/petalinux.* If not, then type *source /opt/pkg/petalinux/settings.sh.*

c) Create a new Petalinux project by issuing the command, *petalinux-create –t project –s ./zynqBerry.bsp*
Change to the new subdirectory that was created for this project, *cd petaDemo1*

d) Import the hardware description file from *Experiment 1*.
To do so, issue command *petalinux-config --get-hw-description=../*
A graphical user interface is shown where you are given possibility to configure various functionalities for the Linux kernel to be generated. Just exit this panel without changing any settings and without saving.

e) You are now going to configure Petalinux for using a Webb server called Busybox. To do this, you should configure the Linux root file system by issuing command, *petalinux-config –c rootfs*
Another graphical panel is opened. Select *Filesystem Packages*, then *base* and then select *busybox*. The proper settings for the *busybox* panel can be viewed in Figure 4. When done with *busybox* settings, you should save and then exit the graphical panel.
Now you are ready to build the Linux operating system by issuing command, *petalinux-build*
This operation might take a while, so go and get a cup of coffee!

1.8. Prepare your boot medium with the new Linux image and a Webb page

You are now ready to prepare an updated content of the micro flash memory card.

a) Copy file *~/petaProj/petaDemo1/images/linux/image.ub* to the flash memory card
You should thus overwrite previous version of the same file.

b) Download *www.7z* from course home page. This archive contains two files for a simple Webb page and one additional startup shell script file for the Linux system. (You can edit the supplied html file, *index.html* using any preferred tool to give it your own personal appearance) Copy all three files to the micro flash memory card.

c) Assemble the memory card into the holder on ZynqBerry and connect the usb cable. You should see an image during booting and then live video after some seconds.

d) Connect to ZynqBerry using the SDK terminal as described in *Experiment 1*. Type command *ifconfig* to find out the ip-number of ZynqBerry.
Use a Webb browser on your host computer and open page *http://ip-address* where *ip-address* should be the ip-number assigned to ZynqBerry. Can you now see your Webb page?

1.9. Saving your Petalinux project

A compiled Petalinux project occupies approx. 6 Gbyte of data and it might take long time to make a backup copy. There should be usb-3 ports available, which might result in reasonable times. But if still too long time,

a) Issue command *petalinux-build –x mrproper* to clean your Petalinux project and reduce its storage size. (All settings will be kept and you can recompile the project (about 17 minutes) next time you need to use it)

b) Save your Petalinux project to a preferred medium, usb memory or home account.

2. Experiment 3: Prepare your first custom designed IP

2.1. Introduction

This experiment will guide you through the process of designing a multiplier IP module. This IP should have an AXI4-Lite slave interface to allow the multiplier to be connected to your ZynqBerry Linux computer. It is advisable to read about Xilinx implementation of the AXI protocol before you start working on this task [10]. The multiplier will have four 32-bit registers mapped to the memory space of your embedded Linux computer. Two of those will be used for operands and the other two for multiplier result. You will learn how to connect this multiplier IP to the reference design you have been using in *Experiment 1* and *2*. *Peek* and *Poke* commands will be used for initial tests of the multiplier IP.

2.2. Create a subproject for the design of a custom peripheral IP

Start your work by opening the hardware reference design used for *Experiment 2* in Vivado.

a) You should now add a template for a new peripheral IP to the user IP repository. In Vivado, select *Tools* and then *Create and Package New IP…* Press *Next* on the first panel that appears. A second panel shown in Figure 5A will appear. Select the AXI4 peripheral. A third panel shown in Figure 5B will appear. Give your IP a name e.g. *myIP* on the first line and then press *Next*. A fourth panel shown in Figure 6A will appear. *Enable interrupt support* should be deselected. Press Next and a fifth panel shown in Figure 6A will appear. Mark selection *Edit IP* and then press *Finish*. A new rtl development project will show up in Vivado having its own project file located in the user repository of the reference design.

b) Explore the two automatically generated VHDL files that build up the peripheral IP template. One of the VHDL modules implements the AXI4-Lite interface while the other one is a simple top level wrapper. The model for the AXI4-Lite has four slave registers, *slv_reg0* to *slv_reg3*. Those four registers will be mapped to the ZynqBerry computer memory address space upon connection. Locate the processes that handles reading and writing of those registers. All four registers are by default read-and-write registers. Make changes to the model so that *slv_reg2* and *slv_reg3* becomes read only registers. The definition of those register values (writing) should be handled by the multiplication output. That is why you should now continue with modeling of the multiplication.

2.3. Add and verify a VHDL model for the multiplier hardware

You are going to add a synthesizable model for a simple multiplier hardware. This core functionality could instead be e.g. signal processing of sensor data or a co-processor for vector data computations. The multiplier function is chosen for its simplicity while you are concentrating on learning how to define user peripheral IPs.

a) Add a new empty VHDL source code file to the peripheral IP project created at 2.2a and then write the following entity description of a multiplier module,

```
entity myIPfuntion is
   generic (
       -- Width of operands and result
       DATA_WIDTH      : integer    := 32
   );
  Port (
   clk : in std_logic;
   op1 : in std_logic_vector(DATA_WIDTH-1 downto 0);
   op2 : in std_logic_vector(DATA_WIDTH-1 downto 0);
   prodL : out std_logic_vector(DATA_WIDTH-1 downto 0);
   prodH : out std_logic_vector(DATA_WIDTH-1 downto 0)
  );
end myIPfuntion;
```

b) Write a synthesizable architecture description in VHDL that computes the products of *op1* and *op2*. The least significant part of result should be in *prodL* and most significant in *prodH*. Make sure that the generated multiplier is built from FPGA on-chip multipliers and not a gigantic combinatorial network.

c) Add another VHDL file for simulation and create a simulation test bench. Simulate and verify the multiplier module's functionality.

d) Now, you are ready to instantiate the multiplier hardware into your peripheral IP. Instantiate entity *myIPfunction* as a component in the AXI4-Lite interface model that you edited in section 2.2b). Connect *op1* and *op2* to slave registers *slv_reg0* and *slv_reg1*. Connect *prodL* and *prodH* to *slv_reg2* and *slv_reg3*.

Connect also the AXI bus clock to the multiplier. This arrangement will allow any software application to write operands to *slv_reg0* and *slv_reg1* and then read result of multiplication in registers *slv_reg2* and *slv_reg3*.

c) You should now make your custom peripheral available by pressing *Package IP* in the project manager panel of Vivado.

2.4. Add your multiplier IP to the reference demo design

a) Close the peripheral IP development project and reopen the reference demo design from *Experiment 1*.
b) Open the main graphical block design by pressing *Open Block Design* in the *IP INTEGRATOR* panel.
c) Right click somewhere in the diagram and press *ADD IP…* Search and select the multiplier IP that you have previously created. The symbol for your IP will now appear somewhere in the diagram. On top of the diagram, the choice of *Run Connection Automation* will appear. Run this design assistant and your IP will be automatically connected and mapped to the microprocessor system's memory space.
d) The block diagram window has a tab for editing address mappings, press *Address Editor*. You can reduce the allocated space to the minimum value, 4K. Make a note of the *OffsetAddress* assigned to your IP. This is the physical memory address to the first of the four 32-bit registers.
e) Synthesize and implement your new hardware design and then generate bitstream.
f) From *File* menu in Vivado, select *Export Hardware*. Make sure to select *Include Bitstream*.

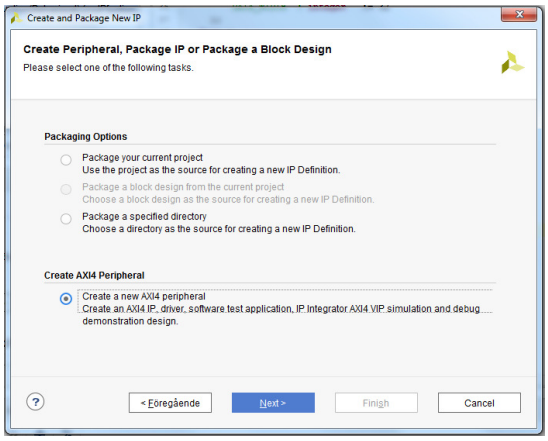
2.5. Build and install a new Petalinux system on ZynqBerry

You are now going to build a new Petalinux system that can support your added custom hardware.

a) Copy files *zsys_wrapper.bit* and *zsys_wrapper.hdf* located in your Vivado/SDK project libraries to the Linux environment for Petalinux tools, copy to directory *~/petaProj*
b) Download an executable First Stage Boot Loader (FSBL) from the course home page, download *zynq_fsbl.elf* to directory *~/petaProj*
c) Before using Petalinux tools, remember to firstly setup the working environment,
cd ~/petaProj/petaDemo1 and then command *source /opt/pkg/petalinux/settings.sh*
d) You should now import the new hardware description file that was exported from Vivado project. Type shell command *petalinux-config –get-hw-description=../*
You can just exit and close the kernel configuration menu that pops up.
e) The commands *Peek* and *Poke* are user software applications that firstly need to be activated before you can use them on your ZynqBerry board. For this you need to configure the root file system. Issue command *petalinux-config –c rootfs*
f) Select *apps* and activate selection *peekpoke*. Save configuration end exit.
g) Build the new Linux system by issuing shell command *petalinux-build*
h) Generate a new boot system, *BOOT.BIN* by issuing shell command,
petalinux-package --boot --format BIN --fsbl ../zynq_fsbl.elf --fpga ../zsys_wrapper.bit --u-boot –force
This command merges the FSBL with fpga configuration bit-file and the second Linux loader u-boot.
i) You will find the new versions of *BOOT.BIN* and *image.ub* files in directory
~/petaProj/petaDemo1/images/linux/
Program the ZynqBerry on board flash memory with *BOOT.BIN* and copy the new version of *image.ub* to the flash memory card.
j) Restart ZynqBerry and make sure it boots up as expected.

2.6. Verify function of the multiplier IP

a) Launch the SDK development environment from Vivado. Then start the SDK Terminal and connect with ZynqBerry as shown in Figure 1C and Figure 1D. Logon the Linux system using *root root*.
b) Assume for example that the *OffsetAddress* of your multiplier IP is *0x43c40000*. You previously made a note of this address, see 2.4d. Then issue commands:
poke 0x43c40000 255 and then *poke 0x43c40004 4* which means that you want to compute 255·4
Issue command *peek 0x43c40008*. This command should now respond with *1024* if the multiplier hardware is working correctly?

A) 

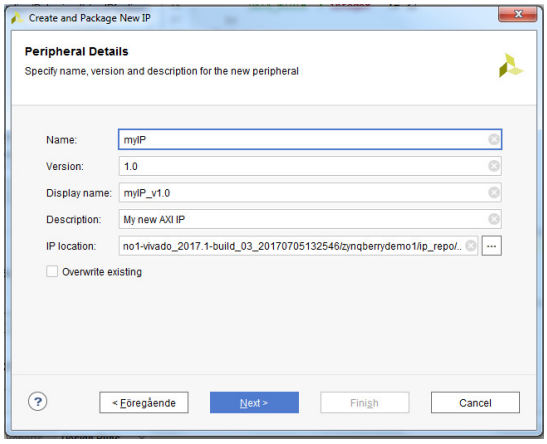
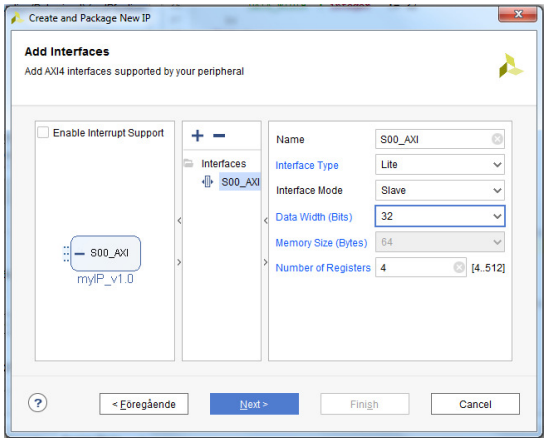
B) 

Figure 5. Package IP menus.

A) 

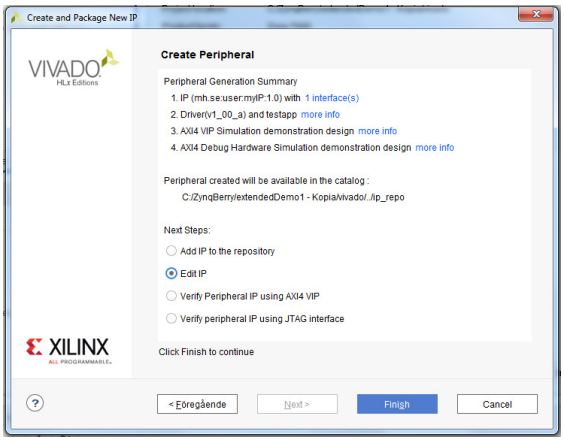
B) 

Figure 6. Package IP menus.

3. Experiment 4: How to access your custom IP from a software application

3.1. Introduction

This custom designed hardware IP that you have designed in previous experiment is physically mapped to the address space of the microprocessor using the four 32-bit registers. These registers are directly available for the Linux kernel software. But how to access those physical registers within a software application running in user space? The simplest method is to map the physical address space with Linux user address space for a small limited range of addresses. This will provide access to your IP registers using memory read and write operations. This simple method is only possible to use whenever you know that there are not more than one application competing for register access.

The second method that you will explore in this experiment is to interface with your IP hardware using a User I/O framework (UIO). The UIO framework provides a simple method to implement a Linux device driver for your custom IP. This device driver performs three major tasks: (1) communication with a custom IP hardware, (2) interaction with the kernel and (3) communication with user applications through a device. A device in Linux is simply handled as a file by the user. Interrupt handling is implemented as blocking read, which means that the execution of a file read operation within the user application will be stalled until an interrupt occurs. However, our experiments on UIO will be limited to not using interrupts. In the case of using UIO, your IP will be protected against conflicting simultaneous write operations.

3.2. Interface with your multiplier IP using direct access to physical address space

a) Start your work by opening the "Hello World" application that you created in task 1.5.

b) Locate the source code for Linux shell commands *peek* and *poke*. You will find the code within the PetaLinux project that you created in task 2.5. Hint: use Linux command *grep* to find files that contains the keyword *peek*. Change current location on your Ubuntu machine to,
~/petaProj/petaDemo1/project-spec and then type command, *grep –R –l peek*

c) Open the found source code file and study how the physical memory is mapped to user address space. You will find out that a function *mmap* is used for the mapping and this function operates on the device */dev/mem.*

d) Create two separate source code files: *peekpoke.h* and *peekpoke.c* in your SDK project. These two files should implement two functions for reading and writing to addressed memory locations:
- int peek(unsigned addr, unsigned *prod);
- int poke(unsigned addr, unsigned val);

e) You previously made notes on the physical register addresses when working with your multiplier hardware, see 2.4d. Include function calls into you "Hello World" program to *poke* for writing and *peek* for reading. Verify the functionality of your 32 bit multiplier.

3.3. Configure Petalinux project for interfacing with your multiplier IP using an UIO device driver

You can continue working on the same "Hellow World" program code as in previous task 3.2, just adding an alternative method to access the same hardware. But before adding any new code lines in "Hello World", you should start by making a few changes to the Petalinux project**.**

A Device Tree Specification (.dts files) is an important element of a Petalinux project. The purpose of DTS is to specify the hardware system that the Linux kernel is running on. A Device Tree Compiler (DTC) is used to compile such specifications into a binary representation (.dtb files) that can be read and interpreted by the Linux operating system kernel at boot time. This will allow the same kernel machine code to run on any Arm processor system having varying sets of peripherals and coprocessors. A community is specifically working on development of device tree mechanisms and DTS formats [11]. Thomas Petazzoni, *free-electrons.com* has published a presentation on a Linux forum where the mechanisms of device trees are explained [12].

a) Change current directory on your Ubuntu machine to
~/petaProj/petaDemo1/components/plnx_workspace/device-tree-generation
Open file, *plnx_arm-system.dts* using a text editor e.g. *gedit*
Scroll down this file until you will find the following record,

```
myMult@43c40000 {
        compatible = "xlnx,myMult-1.0";
        reg = <0x43c40000 0x10000>;
        xlnx,s00-axi-addr-width = <0x4>;
        xlnx,s00-axi-data-width = <0x20>;
};
```

A record in DTS, such as this one is describing an instantiation of a memory mapped hardware module, *myMult*. The base address for this instantiation is *0x43c40000*. *compatible = "xlnx,myMult-1.0"* means that a device driver *myMult-1.0* is interfacing the user space with the *myMult* hardware. This device driver is just an empty template generated by Petalinux. Instead of writing our own custom device driver, we are going to make use of the UIO framework. The driver specification must then be changed to, *compatible = "generic-uio"*. However, you should not do this directly in the opened .dts file.

b) Change current directory on your Ubuntu machine to *~/petaProj/petaDemo1/project-spec/meta-user/recipes-bsp/device-tree/files*
Open file, *system-user.dtsi* using a text editor e.g. *gedit*. The file extension *.dtsi* is used for include files that are included into the DTS. At the end of *system-user.dtsi* you should add the following record,

```
&myMult_0 {
  Compatible = "generic-uio";
};
```

This is a record that allows editing of an existing record where `&myMult_0` is referring to a record previously labelled by `myMult_0`. In this case, the property `Compatible` is changed from "xlnx,myMult-1.0" to "generic-uio". What is important here is that *system-user.dtsi* is not generated automatically by the Petalinux tools. This file is provided as a mean for users to add changes.

c) You will now configure the Linux kernel to include the UIO driver as a Loadable Kernel Module LKM. Change current directory on your Ubuntu machine to *~/petaProj/petaDemo1* and then run command *source /opt/pkg/petalinux/settings.sh* to set up the Petalinux environment if that has not been done before. Then run command *petalinux-config –c kernel*
A configuration panel will show up. Select *Device Drivers* and then mark *Userspace I/O drivers* with an <M> for module. Select *Userspace I/O drivers* and then mark *Userspace I/O platform driver with generic IRQ handling* with an <M> for module. You should now save your settings and exit the configuration panel.

d) To allow the UIO driver to load properly into the Linux kernel we need to change the kernel boot arguments. Run command *petalinux-config* and then select *Kernel Bootargs* on the panel. Disable *generate boot args automatically*. You are now allowed to edit a line with boot arguments that should include the previously automatically generated arguments with the additional, *uio_pdrv_genirq.of_id="generic-uio"*
Have a look at Figure 7. Save changes and exit the configuration panel.

e) Build a new Linux image by running commands *petalinux-build –x mrproper* and then *petalinux-build*
The first command using argument *–x mrproper* is used to rebuild the kernel from scratch after editing DTS [8].

f) Copy the new image *~/petaProj/petaDemo1/images/linux/image.ub* to the flash memory card on ZynqBerry. Restart ZynqBerry and log on to the Linux system using SDK terminal.

g) Type command *ls /dev* and make sure that you can find file *uio0* among the list of devices.

h) Type command *more /sys/class/uio/uio0/name* and you will find out that device uio0 is associated with hardware *myMult*.

i) Type command *modprobe –l* which is a command that will list all LKM as .ko files. Among those files you will find, *kernel/drivers/uio/uio_pdrv_genirq.ko* and *kernel/drivers/uio/uio_xilinx_apm.ko*
A .ko file is a binary object code file that executes in kernel space. The list of LKM is probably long but there is only a few that are loaded into the kernel and being active. Loaded LKM can be listed by command *lsmod*

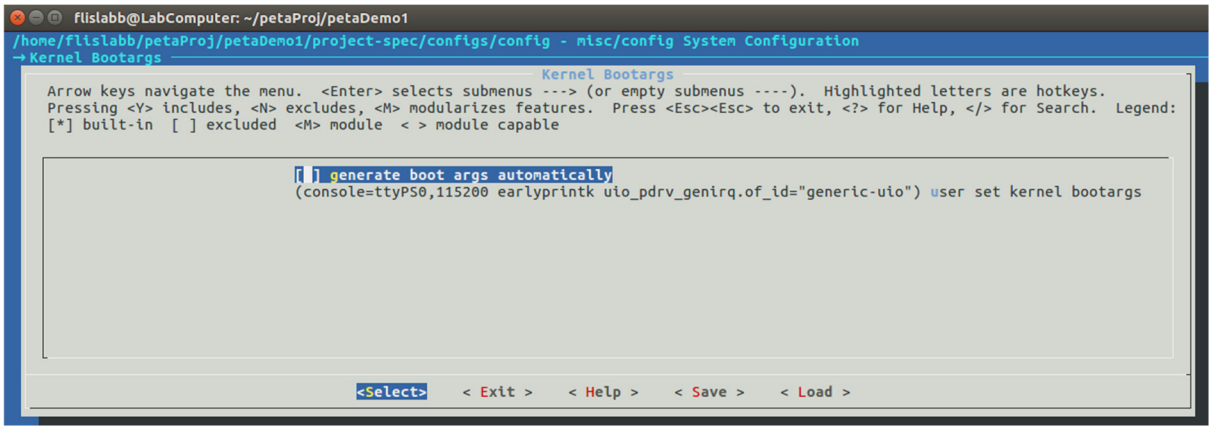


Figure 7. Configuration panel for custom kernel boot arguments.

3.4. Write a Linux software application for accessing your custom IP

Accessing your custom IP is done from a Linux application by firstly opening */dev/uio0* as a file.

a) `fd = open("/dev/uio0", O_RDWR);` is a function that returns an integer value larger than zero if the opening went well. Include also code for error handling in case the file open didn't succeed.

b) Next step is to map the file device to memory,
`ptr = mmap(NULL, MYMULT_MAP_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);`
This function returns a pointer that should be declared as void*. You can access guidelines on how to use this function if you simply search the Internet for *mmap()*.

c) Reading and writing to registers in your custom IP is now only a matter of accessing memory locations. Writing an unsigned integer to the second register can for example be done as,
`*((unsigned *)(ptr + OP2_OFFSET)) = 10;` where the offset to the first register `OP2_OFFSET` is four.

d) At the exit of your application, it is advisable to also release the mapping of user memory space by calling function `munmap(ptr, MYMULT_MAP_SIZE);` `MYMULT_MAP_SIZE` is the size of the memory mapping. Remember also to close the file device.

e) Write a simple software application similar to 3.2. Sven Andersson has published a code example *LED_DimmerUIO.c* in his blog, issue 42 that can give you guidance on how to write the code [5]. Verify the functionality of your multiplier IP.

4. Experiment 5: Interface with an external I2C temperature sensor

4.1. Introduction

This experiment will guide you through the process of connecting an external temperature sensor to the general purpose I/O breakout connector and interfacing a software application with that sensor. Xilinx has included device drivers into Petalinux environment to be used with the I2C interface on Zynq platforms. A complete list of all Xilinx device drivers can be found at [16]. The hardware description file .hdf generated in Vivado will adapt depending on if e.g. the I2C interface is activated or deactivated in the Arm processor IP. The Linux device tree specification will be adapted accordingly when the Vivado .hdf file is imported into Petalinux environment. Any I2C interface will appear as a file interface in the target system under directory */dev*. You will write a software application that continuously read temperature values from the external temperature sensor, displays values on the terminal and writes data to a log file.

4.2. Investigate the ZynqBerry hardware design for interfacing with I2C devices

You should preferably start by studying the ZynqBerry hardware manual [13] and look for the description on I2C interfaces. You will find out that an I2C multiplexer and interrupt controller circuit (TCA9544A from Texas Instruments) is connected to one of the two Zynq I2C ports, I2C1. This multiplexer splits the I2C1 interface into four separate interfaces: 0) connects to the general I/O breakout, 1) connects to a display interface, 2) connects to HDMI and 3) connects to the camera interface. A device driver is also available for the multiplexer/controller such that those four different I2C interfaces will appear under target directory */dev* as separate files. Which file device that maps to which I2C interface can be found out by studying the target directory structure */sys/class/i2c-dev*. As for example, command *more /sys/class/i2c-dev/i2c-2/name* will indicate that file device */dev/i2c-2* is connected with channel 0 on the multiplexer which in turn is connected to the I/O breakout connector. See also circuit diagrams for ZynqBerry [17]. Figure 10 in appendix A shows a photo of the general I/O breakout connector having signal names printed on it. Do not consider those names because they are relevant only for the RaspBerry Pi computer. Instead use the list of signal names and Zynq device pins that are attached on each side of the photo. *I2C-SCL* and *I2C-SDA* are the clock and serial data signals that you should use for communication with an external temperature sensor.

4.3. Connect the sensor hardware

A temperature sensor, TC74A0-3.3VAT from Microchip [14] is a device that is recommended for this experiment. However, there is a great range of compatible temperature sensors on the market that could be used. Read the data sheet carefully and connect, ground, power, clock and serial data accordingly. This sensor device is working with 3.3 V logic and power supply. Figure 8 shows an example of how this connection can be done. Rout short wires and make it as neat as possible, no rat nest please!

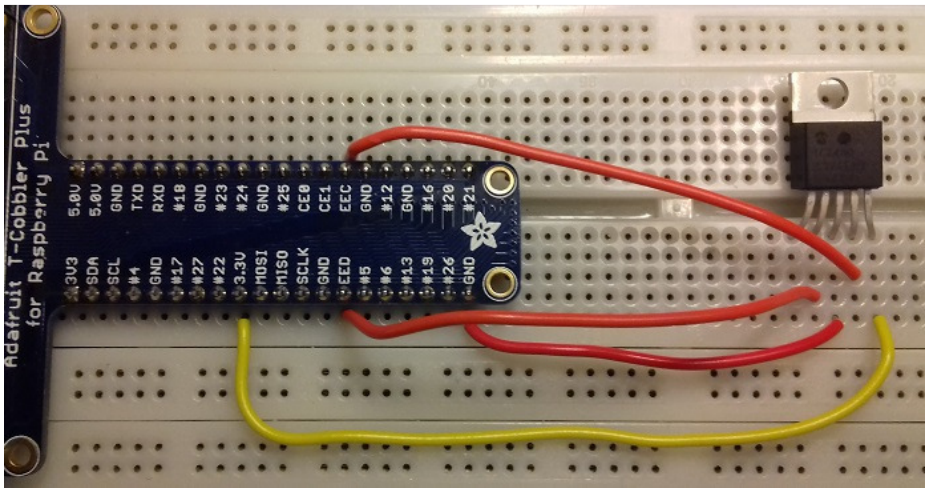


Figure 8. Temperature sensor connected to the i2c bus available on the I/O breakout connector.

4.4. Investigate the device tree specification for I2C configurations

Change current directory on your Ubuntu machine to
~/petaProj/petaDemo1/components/plnx_workspace/device-tree-generation
Open file, *plnx_arm-system.dts* using a text editor e.g. *gedit*
Scroll down this file and you will find records for the two I2C interfaces at addresses *0x4000* and *0x5000*. The latter has additional records for the multiplexer and its related four I2C channels.

4.5. Write a Linux software application for temperature logging

Create a new empty Linux application in Vivado SDK to be developed into a temperature logger. Attach a new source code file to this application having the main{} procedure. Proper include files to use are:

```
#include <stdio.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <time.h>
```

a) In 4.2 you most likely came to the conclusion that */dev/i2c-2* is connected with your temperature sensor. The first thing to do is to open this file device for read and write operations,

```
int i2c_file;

if ((i2c_file = open("/dev/i2c-2", O_RDWR)) < 0) {
            perror("Unable to open i2c control file");
            return 0;
}
```

b) The second step is to associate the open file with your temperature sensor that has a specific hardware address on the I2C bus.

```
if (ioctl(i2c_file, I2C_SLAVE, TEMP_SENS_ADDR) < 0) {
            perror("Selected i2c device is not available");
            return 0;
}
```

The temperature sensor address should be defined as 0x48 in accordance with data sheet [14].

c) Reading a temperature value involves two separate operations: 1) write an 8 bit command and 2) read an 8 bit temperature value. The command for reading the data register should be 0x00 according to data sheet. The described sequence is captured by the following lines of code,

```
buffer[0]=0x00;
write(i2c_file, buffer, 1);
read(i2c_file, buffer, 1);
```

The connected temperature sensor is using a protocol called SM bus [15], which is a derivate from the I2C specification.

d) Write source code for a complete software application that continuously reads temperature values from the external sensor, displays values on the terminal and writes data to a log file. A suitable period of time in between measurements is one second. Hint: use function *usleep()* to delay the execution for one second inside a while-forever-loop. Can you see a small change in temperature if you put your finger on the sensor?

5. Experiment 6: Package, interface and use a custom peripheral IP for a range sensor

5.1. Introduction

It is strongly recommended that you should finish doing all previous experiments before starting with this peripheral IP. You have previously designed and verified digital hardware for range sensing using an ultrasonic transmitter and receiver. This previous work was part of the course module VHDL. In this experiment, you should reuse that VHDL code and package it into a reusable peripheral IP component. This IP should interface with the rest of the Arm based microprocessor system using the AXI4Lite protocol.

5.2. Package IP

This task is almost to repeat what you previously learned in sections 2.2 and 2.3. Instead of the previous multiplier, you should use your VHDL-model for the range sensor. Keep in mind that you need to add two additional signals to this IP allowing for connection with the external ultrasonic sensor.

5.3. Prepare the hardware system

You should now create a very similar hardware system as in section 2.4 except that instead of including a multiplier IP, you should include the range sensor peripheral IP to your design. What more is, the two signals that need to be connected with the external ultrasonic sensor must be assigned to the proper fpga I/O-pins. Figure 10 allows you to choose any two free I/O pins and then give the proper constraints in the Vivado project constraint file.

5.4. Customize the Linux system

Prepare a Petalinux project for interfacing application software with the range sensor IP using a UIO device driver. This is the same procedure as described in previous section 3.3.

5.5. Software application for testing

You should now write c-code for a software application that continuously reads and displays range data on the console output. This task is more or less the same as in section 3.4.

6. Experiment 7: Prepare a Webb service for a remote metrology station

6.1. Introduction

It is strongly recommended that you should finish doing all previous experiments before starting with this Webb service. Previous experiments 5 and 6 guided you on how to connect with an external temperature sensor and how to connect with an external ultrasonic range sensor. You learned how to write software applications for continuous access and display of sensor data. Section 1.8 in experiment 2 guided you on how to activate a Webb server on ZynqBerry and how to publish a home page on that server. In this final grand experiment, you should reuse previous results and create a remote metrology station that can continuously measure range values and temperatures, create trend curves for those two parameters online such that they can be viewed remotely on a Webb browser. The goal is also that ZynqBerry should no longer have to be connected to your host computer, using a debugger for running applications. ZynqBerry should instead become a standalone embedded system that starts to log data and present trend curves directly when you connect it to power.

6.2. Hardware system

The configurable hardware system as it is modeled in Vivado can be the same as in experiment 6. In addition, you should physically connect the external temperature sensor as described in section 4.3 for experiment 5.

6.3. Linux operating system

You should reuse the Petalinux project that was customized to generate a Linux system for use with the range sensor in experiment 6. Make sure that the Petalinux settings for Webb server pointed out in section 0 is corresponding to Figure 4. If necessary, rebuild and use a new Linux image *image.ub*.

6.4. Write a software application for a sensor data logger

You should write c-code for a software application *logger* that continuously reads temperature and range values from the two externals sensors. The time between each sample of data should be approximately 5 seconds. Sensor values along with time stamps should continuously be written to a log file, */srv/www/myLog.csv*
The location */srv/www* on target environment is where the Webb server is reading data html data from. Every line in the file *myLog.csv* represents one sample of data and should have the following format,
hour minutes seconds,temperature,range
Be aware of that there is now separating comma sign between hour, minutes and seconds. Verify the functionality of the logger software by inspecting the written text file.

6.5. Install the logger software as a software application in Linux

You have created and verified the logger application using Vivado/SDK environment in previous section 6.5. The debugger functionality of this development environment was used for this verification, also handling downloading of executable code to the target environment, ZynqBerry. However, when you want to create a standalone Linux computer that also includes the developed application, then an installation procedure must be done in the Petalinux project used to generate the Linux image.

a) Return to the Petalinux project from section 6.3. You should now create a template for an application called *logger*. Run command *petalinux-create –t apps –n logger –enable*
b) Replace the template source code file
~/petaProj/petaDemo1/project-spec/meta-user/recipes-apps/logger/files/logger.c
with the developed logger code from 6.4
c) The application *logger* will be compiled when the Linux system is built. Type command,
petalinux-build
d) Copy the new Linux image, *image.ub* to the flash memory card and restart ZynqBerry. Make sure that the logger application is present at location */usr/bin* on ZynqBerry.
e) You should now be able to start your logger application by simply running command *logger*
Verify that a file *myLog.csv* is created at location */srv/www* and that it contains proper data.

6.6. Prepare and verify a Webb service for data logging

The html file that holds the Webb page on your ZynqBerry needs an update to provide the Webb service. The new version should include a Java script for reading log file and generating graphics for two trend curves.

a) Download new Webb files for the metrology station available on the course home page. Copy those new files to the flash memory card.
b) Open in a text editor the shell script *init.sh* located on the flash card. You will find out that now there is a command, *logger &* included that will start your data logger application upon system startup. The additional character "&" instructs the shell command interpreter to start the logger software as a separate process in the background. This enables users to still have access to command prompt.
c) Open the new *index.html* in a text editor so that you can inspect the embedded Java code for graphing of logged data.
d) Insert flash card into ZynqBerry and restart.
e) Check for assigned IP address using *ifconfig* and open a Webb browser with that address. Do you see the Webb page with trend curves for range and temperature?

7. Experiment 8: Analysis

7.1. Introduction

The purpose of this experiment is to apply the knowledge that you have gained from reading the course literature and from attending the introductory lectures. This experiment is more on the thinking and writing level and do not require any additional work in the lab. The outcome should be drawings and text that can be included into the final project report.

7.2. Design methodology

You should analyze and comment on the design methodology that you have used in this set of experimental works on SoC design.

7.3. Circuit technology

You should analyze and comment on the circuit technology used for the ZynqBerry board and compare it with other circuit technologies and other combinations of circuit technologies available on the market, give pros and cons.

7.4. Hardware architecture

You should analyze and comment on the hardware architecture used for your metrology station in previous experiment 7. Analyze busses, peripherals, memories and the Arm processor architecture. The Xilinx SoC device XC7Z010 includes a dual-core Arm Cortex-A9 processor architecture [19].

8. Examination

8.1. Learning goals

After completion of the course module on SoC design, student should know how to:

- **a)** Use tools for software-hardware co-design using reusable IP-components,
- **b)** Write c-code for an application supported by an operating system,
- **c)** Embed custom developed hardware and software drivers into a reusable IP-component,
- **d)** Analyze the selected design methodology for a given project assignment,
- **e)** Analyze the selected circuit technology for a given embedded system,
- **f)** Analyze the hardware architecture for a given embedded system.

8.2. Oral presentation

Every student group should prepare a 20 minutes oral presentation (15 minutes talk and 5 additional minutes for questions) of the whole set of experiments. This presentation must have a disposition corresponding to a full technical report. This means that you need to give an introduction, define the problem, present methods and results, as well as also analyse the results, draw conclusions, and point out possible alternative methods. A seminar will be held at the end of the course, where all student groups will give their oral presentations. All group members must participate actively in the presentations. The oral presentations must be based on PowerPoint slides. The first slide must have the names of all group members written on it. Power point slides for the given presentation are then sent to the teacher the day before the seminar.

8.3. Report

Every student group should hand in a full project report. The overall, as well as the verifiable goals for your work should as always be pointed out at the beginning of your report. In this case, goals should correspond to the learning goals as defined in the syllabus. See section 8.1. In the discussion section of your report, you should validate to what extent you have met those goals.

References

[1] ZynqBerry project at Hackaday, https://hackaday.io/project/7817-zynqberry
[2] Trenz Electronic Gmbh, https://shop.trenz-electronic.de/en/
[3] Xilinx, Vivado Design Suite Tutorial, Embedded Processor Hardware Design, UG940
[4] Xilinx, Petalinux SDK User Guide. Application development guide, UG981
[5] Sven Andersson is a consultant in FPGA/ASIC design who writes a blog, http://svenand.blogdrive.com/
[6] The Yocto project, https://www.yoctoproject.org/
[7] Xilinx, Petalinux Tools Documentation, Workflow Tutorial, UG1156
[8] Xilinx, Petalinux Tools Documentation, Command Line Reference, UG1157
[9] Linux Quick Reference Guide, http://danleff.net/downloads/linux/linux_quick_ref_card.pdf
[10] Vivado Design Suite, AXI Reference Guide, UG1037
[11] Community for the development of a device tree standard, https://www.devicetree.org
[12] Explanation of device tree, Thomas Petazzoni, https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf
[13] ZynqBerry hardware manual, https://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/TE0726/REV03/Documents/TRM-TE0726-03.pdf
[14] Data sheet for I2C temperature sensor, http://ww1.microchip.com/downloads/en/DeviceDoc/21462D.pdf
[15] Specification of SM bus, http://www.smbus.org/specs/smbus20.pdf
[16] List of available Linux device drivers from Xilinx, http://www.wiki.xilinx.com/Linux+Drivers
[17] Schematics for ZynqBerry,
[18] https://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/TE0726/REV03/Documents/SCH-TE0726-03M.PDF
[19] The Arm A9 architecture, https://developer.arm.com/products/processors/cortex-a/cortex-a9/documentation

Appendix A - Experimental equipment in the laboratory

1.1. Hardware platform

An experimental board "ZynqBerry" developed as an open source crowdfunded project [1] and sold by Trenz Electronic GmbH [2] is used for all experiments in this compendium. See device (A) in Figure 9. This board has the same form factor and the same I/O as the more known Raspberry Pi computer. ZynqBerry is designed with a System on Chip device from Xilinx being a heterogeneous combination of Field Programmable Gate Array, two Arm processors and I/O-interfaces. This architecture allows users to define both custom designed hardware and software.
Device (B) is a colour camera sold as an accessory for the Raspberry Pi computer. This camera can also connect with its corresponding camera port on the ZynqBerry (A). Both devices (A) and (B) are embedded into a casing (C).

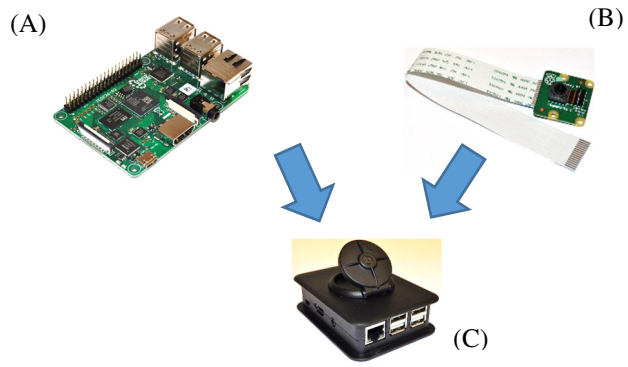
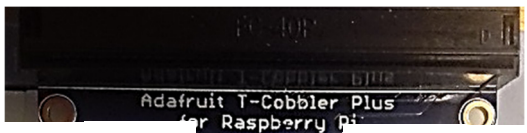


Figure 9. Experimental equipment

1.1.1. Pinout of expansion header



Zynq Pin	Signal		Signal	Zynq Pin
--	3.3V	3V3 5.0V	5V	--
K15	GPIO2	SDA 5.0V	5V	--
J14	GPIO3	SCL GND	GND	--
H12	GPIO4	#4 TXD	GPIO14	M12
--	GND	GND RXD	GPIO15	N13
G11	GPIO17	#17 #18	GPIO18	H11
G12	GPIO27	#27 GND	GND	--
H13	GPIO22	#22 #23	GPIO23	J11
--	3.3V	3.3V #24	GPIO24	K11
H14	GPIO10	MOSI GND	GND	--
J13	GPIO9	MISO #25	GPIO25	K13
J15	GPIO11	SCLK CE0	GPIO8	L15
--	GND	GND CE1	GPIO7	L14
--	I2C SDA	EED EEC	I2C SCL	--
N14	GPIO5	#5 GND	GND	--
R15	GPIO6	#6 #12	GPIO12	M15
R13	GPIO13	#13 GND	GND	--
R12	GPIO19	#19 #16	GPIO16	L13
L12	GPIO26	#26 #20	GPIO20	M14
--	GND	GND #21	GPIO21	P15

Figure 10. Signal names and mapping to Zynq device pinout.

1.2. Software tools for SoC design

Vivado - The Xilinx Vivado tool is used for modelling, implementation and verification of the HW system.

SDK - Xilinx Software Development Kit (SDK) is an integrated environment for development of software executing on the two Arm processors. A set of connected IP-components described as a hardware system in Vivado can be exported into SDK where SW-drivers for the corresponding HW-functions will be available for the programmer.

Petalinux SDK – Xilinx Petalinux SDK is a software tool that can generate a Linux operating system that can run on the embedded Arm processors. This OS is adapted to the set of functionalities incorporated in the HW system. Hardware description files generated in Vivado and Petalinux tool configurations steer this adaption to the targeted custom designed HW system.

Important - Both Vivado and Petalinux SDK must have the same version number. This is very important. The design files published on course home page as well as printouts of software panels in this document is developed for version 2017.1