Lecture 4 notes - Linux OS creation over MPSoC

Main author: Frederik Nyboe - Instructor: Thor

Fall 2025

1 Overview

The goal of developing robotic applications with Multi-Processing System-on-Chip (MPSoC) is to combine the versatility of high-level software with the power of custom Field-Programmable Gate Array (FPGA) circuitry. Several tools exist for building advanced FPGA systems, however integrating them into an operating system (OS) opens the door for a vast mess of complexity. This is what we will try to master in this and the following lectures.

Having developed custom circuitry for the FPGA, an interface between the FPGA and the CPU is needed. In MPSoC system, this interface is **Advanced eXtensible Interface (AXI)**. AXI defines a hardware bus and a communication protocol which maps **registers** (FPGA side) to **physical memory** (CPU side), allowing the CPU to perform read/write operations into the FPGA, disguised as normal memory operations (i.e. read/write to a pointer).

In a bare-metal application (a stand-alone C-code running on the CPU with no operating system), this is straight forward; memory operations are performed directly on physical addresses and thus only the address assigned to an FPGA entity's AXI interface is required. However, any high-level OS uses a **Memory Management Unit (MMU)** to map between **physical address space** and **virtual address space**. A process running in an OS will by the MMU be assigned a range of virtual addresses which the MMU at run-time will map to physical memory; however, depending on memory requirements from other processes, the target physical memory addresses might be changed, replaced, fragmented, assembled, or even moved to swap memory by the MMU. The MMU will keep track of this, such that the application developer doesn't need to worry about it.

In Linux, this means that a **user space** application (which covers most applications we are developing and running) can not directly access the physical memory we have assigned to our AXI FPGA modules. Only code running in **kernel space** can do that; we therefore need a **kernel driver** running in kernel space to recognize our memory mapped FPGA module and expose a user space interface which our application can utilize. Luckily, there exists a kernel driver called **generic User space I/O (UIO)** which we can apply to any AXI based FPGA module we develop.

We are therefore tinkering with the Linux OS to be running on our MPSoC on the kernel level. Furthermore, the modifications we apply to the OS are different depending on what circuit we have running on the FPGA. Changing something on the FPGA which is mapped to the CPU via AXI is from the point of view of the OS equivalent to changing the hardware on which the OS is running. We therefore need to **build** a custom version of the OS for the specific set of AXI mapped modules we are synthesizing for the FPGA.

Xilinx provides a neat tool for building Linux OS for the MPSoC: **Petalinux**. Under the cover, Petalinux uses **Yocto Linux**, which is a tool for building Linux distributions for custom hardware. Petalinux appends to this functionality by supplying easy integration with MPSoC and FPGA specific hardware and firmware. Petalinux can be instructed to load a specific kernel driver for a specific hardware using a **device tree** file.

Petalinux, however, builds a very scraped Linux distribution which requires a good understanding of Linux to efficiently work with. Furthermore, the complexity of a Petalinux project quickly increases when wanting to port to a specific board, such as the **Ultra96-V2**, which we will use in this class. Multiple different vendor-specific files and objects needs to be properly included. We have therefore in our group built the **MPSoC4Drones** tool (it works also for non-drone applications), which will provide a template Vivado project and a template Petalinux project, and automatically handle all builds. Furthermore, it will build a Ubuntu-based distribution on top of the built Petalinux kernel, such that we can work in a friendly environment once we have flashed the OS onto the Ultra96-V2.

1.1 A Note About Linux

As roboticists, we will most likely encounter different Linux distributions throughout our careers. I have never heard about a robot running Windows or Mac OS (it probably exists though), and I would never built one running anything but a version of Linux. Linux is a very powerful tool, and there are tens of thousands of developers building extremely useful software relying on Linux for different purposes - and a lot of it is freely available as open source, meaning less development effort for roboticists wanting to integrate various functionality. One example is **Robot Operating System (ROS)** which is a middle-ware for robots basically handling all of the nasty complications one can imagine in robot software development (we will learn about ROS in a later lecture). ROS is tightly integrated with the Linux (command line) workflow, as is most other Linux-based tools. Finally, Linux is completely transparent meaning we can modify and change things as we like - for instance leading to the possibility to integrate with custom FPGA systems on the MPSoC.

Not convinced yet? The entire world-wide robotics and drone research and industry communities are using Linux. Both Universal Robots (UR), Mobile Industrial Robots (MIR), and Tesla cars are running Linux.

2 Development Environment Setup

We will be using **Ubuntu 20.04 LTS**. You can install it either natively on your computer (preferred) or in a virtual machine. System requirements for running the tools: lots of RAM (≥ 8 is best), lots of hard drive (the tools alone are about 100 GB, and an MPSoC4Drones project after a full built is around 50 GB). If these requirements are hard to meet for some of you, you can team up - you will be working in groups anyways. Another option for those who have desktops at home is to use TeamViewer. A third option is to install the tools on an external USB-3 (A or C) hard drive. If you have already installed the tools in a Windows partition, you can remove this installation to free up space.

2.1 Password-Less `sudo`

In order to work properly with the tools in this course, set up password-less sudo by entering the following in your terminal:

```
echo "$USER ALL=(ALL) NOPASSWD: ALL" | sudo tee -a /etc/sudoers
```

The changes might require you to log out and in again.

2.2 Xilinx Tools Installation

The Xilinx 2020.2 Unified Installer can be found here or direct download here. Login with your AMD account, fill out the export declaration, and download the file `Xilinx_Unified_2020.2_1118_1232_Lin64.bin`.

Open a terminal, and install a few dependencies:

```
sudo apt install -y xterm texinfo gcc-multilib libncurses5-dev libncursesw5-dev
sudo dpkg --add-architecture i386
sudo apt update
sudo apt install -y zlib1g:i386
sudo apt install -y net-tools autoconf libtool zlib1g-dev build-essential gawk
sudo apt upgrade
sudo ln -s /usr/lib/x86_64-linux-gnu/libtinfo.so.6 /usr/lib/x86_64-linux-gnu/libtinfo.so.5
```

Assuming the file has downloaded to your `Downloads` folder and that you want to install the tools in the folder `/tools`, open a terminal (`ctrl-alt-t`), and do the following to start the installation:

```
sudo mkdir /tools
sudo chown -R $USER:$USER /tools

cd /home/$USER/Downloads
sudo chmod +x Xilinx_Unified_2020.2_1118_1232_Lin64.bin
./Xilinx_Unified_2020.2_1118_1232_Lin64.bin
```

Press Next, login with your AMD account, press Next, tick "Vitis", press Next, untick "Add-On for MATLAB and Simulink", "DocNav", "Install devices for Alveo and Xilinx edge acceleration platforms", and "Versal ACAP", press Next, tick "I Agree" three times, press Next, press Next, press Install. This will take a long time and requires continuous internet connection.

When the install is finished, start the installer once again by

```
cd /home/$USER/Downloads
./Xilinx_Unified_2020.2_1118_1232_Lin64.bin
```

This time, choose "Petalinux" instead of "Vitis". This installation is shorter. Finally, we need to install cable drivers. This is done by issuing

```
cd /tools/Xilinx/Vivado/2020.2/
cd ./data/xicom/cable_drivers/lin64/install_script/install_drivers/
sudo ./install_drivers
cd
```

2.2.1 Issues with Petalinux Installation - not relevant if you have previously installed the dependencies

If you have issues with missing libraries for the Petalinux installation, do the following:

```
sudo apt install -y xterm texinfo gcc-multilib libncurses5-dev libncursesw5-dev
sudo dpkg --add-architecture i386
sudo apt update
sudo apt install -y zlib1g:i386
sudo apt install -y net-tools autoconf libtool zlib1g-dev build-essential gawk
sudo apt upgrade
```

And restart the Petalinux installation.

2.2.2 Installing More Board Files

If during the installation the proper board files have not been installed, open Vivado, press ”Help” in the top bar, then press ”Add Design Tools or Devices”. Then proceed to install the required board files.

2.3 Git Setup

Make sure you have git installed:

```
sudo apt install git
```

In order to work with Git and GitHub, you need to assign an SSH key to your GitHub account, following this guide. Finally, you must specify your credentials:

```
git config --global user.email "<email>"
git config --global user.name "<GitHub username>"
```

2.4 Adding Swap Space

The tools can be quite heavy on the RAM. If your computer runs out of RAM during a build, worst case your computer will freeze badly for a while. One solution is to limit the number of jobs during the run. Another excellent option is to add swap space to your computer. Swap space is a file on the hard disk in which the MMU can temporarily put some of the contents of the RAM, such that you don’t run out, at the cost of speed. 16 GB of swap space can be added as follows:

```
sudo swapoff -a
sudo rm -f /swapfile 2> /dev/null
sudo fallocate -l 16G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
echo "/swapfile swap swap defaults 0 0" | sudo tee -a /etc/fstab
```

Verify that the swapon is active:

```
sudo swapon --show
```

2.4.1 Alternative modify swap space

This move swap spacve to the main memory, this might take several minutes

```
sudo swapoff -a
```

Create an empty swapfile, insert desired GB

```
sudo dd if=/dev/zero of=/swapfile bs=1G count=<GB>
sudo chmod 0600 /swapfile

sudo mkswap /swapfile
sudo swapon /swapfile
```

A swapspace of the desired size should now be mounted.

2.5 Installing QEMU

Install QEMU:

```
sudo apt install -y qemu-user-static
```

3 MPSoC4Drones General Workflow

The MPSoC4Drones framework[1] handles everything related to developing for the Ultra96-V2. The Readme on the GitHub repository along with this section will help you get started and provide an overview. It is highly recommended to read the Readme and the linked article (also on Itslearning). The general idea is to copy the template repository into a new repository for any new MPSoC development project.

Create a new repository from the MPSoC4Drones template by clicking "Use this template". Then clone your newly created repository by

```
git clone <url to repository>
```

You can now start working with MPSoC4Drones. Enter the repository folder and create a new branch:

```
cd <repository folder>

git checkout -b lec4
```

Assuming you have installed the Xilinx tools as specified above, source the tools:

```
source /tools/Xilinx/Vivado/2020.2/settings64.sh
source /tools/Xilinx/PetaLinux/2020.2/tool/settings.sh

source ./scripts/settings.sh
```

The MPSoC4Drones commands are now ready at your disposal. To set up the project:

```
mp4d-setup
```

You can open the target Vivado project, which you will use to develop your system:

```
vivado ./hdl/projects/u96v2_sbc_mp4d_2020_2/u96v2_sbc_mp4d.xpr
```

This project contains the basic block design template. Do not remove anything from the block design. All of your FPGA implementations will be added to this block design, and this project will be where we work with the FPGA design. You can develop and save as you like.

Note 1 *When adding source files to you Vivado project, create the file in the project directory* `src` *folder:*

```
touch <project directory>/src/<new file>.vhd
```

Then add the file to the Vivado project without copying the file by in Vivado clicking the +-sign over the Source browser, tick "Add or create design sources", click "Next", click "Add Files", browse to the file you have created `<project directory>/src/<new file>.vhd` *and select it, untick "Copy sources into project", and click "Finish". This will help MPSoC4Drones commit your changes to the Git repository later. You can then modify the file using the Vivado editor or in an external editor as the file will be linked.*

Once you would like to continue with the build, go to the terminal (where you have sourced as before and entered the project folder), and issue

```
mp4d-build -V
```

to build the Vivado project. This will synthesize, implement, generate bitstream, and export the hardware specification file. If the build fails, enter Vivado to look at the errors. If you would like to test your design in a bare-metal application, you can open Vitis and create a project using the exported hardware specification file `<project directory>/hdl/projects/u96v2_sbc_mp4d_2020_2/u96v2_sbc_mp4d.xsa`.

When everything is good, you can proceed. This is the time at which you configure your device tree to load the UIO kernel driver for your AXI devices, which is done by modifying the file

```
gedit <project directory>/src/system-user.dtsi
```

When the device tree file has been modified, it is time to configure the PetaLinux project to use your hardware design and your device tree file. Issue the command

```
mp4d-build --petalinux-config
```

to configure the Petalinux project, and

```
mp4d-build -P
```

[1] https://ieeexplore.ieee.org/document/9836055
https://github.com/DIII-SDU-Group/MPSoC4Drones

to build the Petalinux project. Finally, we build the Ubuntu file system by

```
mp4d-build -U
```

The builds can be easily done sequentially by

```
mp4d-build -V -P --petalinux-config -U
```

or simply

```
mp4d-build
```

For any changes to the hardware, it is a good idea to perform the full build before modifying the kernel configuration, and then rebuild from the `--petalinux-config` step. Subsequent builds will be significantly faster.

The SD-card can then be flashed. See the GitHub Readme for information on partitioning of the SD-Card. Having partitioned the SD-card as specified, insert it into the computer, and it can be flashed using

```
mp4d-package
```

4 Hardware Accelerated Memory Copy System

In this lecture, we will be building a custom embedded memory copy system. An FPGA system copies data from one memory address range (the *input range*) to another memory address range (the *output range*), such that anything written to the input range by a process running on the CPU in Linux will be readable from the output range by another process. The copying of the data is one-way, effectively implementing hardware accelerated process-to-process communication. A diagram of the system is seen in Fig. 1.

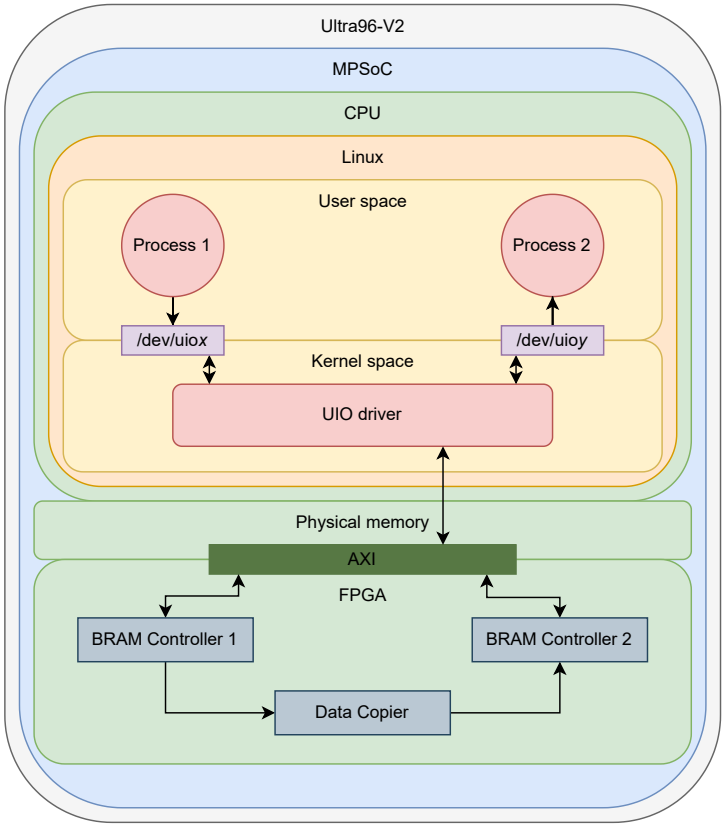

Figure 1: Conceptual diagram of the Embedded Hardware Accelerated Memory Copy System

The exercise will showcase the following points:

1. Development of an AXI-connected FPGA system with awareness of the address range with Vivado;
2. Testing the FPGA-CPU interface in a bare-metal application with Vitis;
3. Configuring a Petalinux build and setting up the kernel to load the generic UIO driver for AXI-connected FPGA modules;
4. Building and flashing the image using MPSoC4Drones;

5. Booting the Ultra96-V2 and accessing it through SSH; and
6. Writing a C++ application to test the functionality.

As the build process takes a long time, build products are provided for the steps along the way.

4.1 Vivado

First, open a terminal, enter your project directory, and source the tools:

```
cd <project directory>

source /tools/Xilinx/Vivado/2020.2/settings64.sh
source /tools/Xilinx/PetaLinux/2020.2/tool/settings.sh
source ./scripts/settings.sh
```

Download the provided `data_copier.vhd` file and add it to the project by putting it in the `src/` folder, assuming it has been downloaded to the `Downloads/` folder:

```
mv /home/$USER/Downloads/data_copier.vhd ./src/
```

Inspect the file:

```
less ./src/data_copier.vhd
```

The entity header defines the ports for interfacing with two separate BRAMs as well as a clock input:

```
entity data_copier is
    port (
        clk         :   in  STD_LOGIC;

        bram0_din   :   in  STD_LOGIC_VECTOR(31 downto 0);
        bram0_wen   :   out STD_LOGIC_VECTOR(3 downto 0);
        bram0_addr  :   out STD_LOGIC_VECTOR(31 downto 0);

        bram1_dout  :   out STD_LOGIC_VECTOR(31 downto 0);
        bram1_wen   :   out STD_LOGIC_VECTOR(3 downto 0);
        bram1_addr  :   out STD_LOGIC_VECTOR(31 downto 0);

        bram_en     :   out STD_LOGIC;
        bram_rst    :   out STD_LOGIC
    );
end data_copier;
```

Here, `bram0` will be the input and `bram1` will be the output. In the architecture header, a counter is defined:

```
    architecture Behavioral of data_copier is
        signal cnt  :   UNSIGNED(31 downto 0)   :=  (others => '0');
```

This counter signal drives the behavior of the module. A process increments the counter on rising clock edge:

```
    cnt_process: process(clk)
    begin
        if (rising_edge(clk)) then
            cnt     <=  cnt + "1";
        end if;
    end process;
```

The BRAMs are then driven by concurrent statements:

```
    bram_en     <=  '1';
    bram_rst    <=  '0';

    bram0_wen   <=  (others => '0');
    bram1_wen   <=  (others => STD_LOGIC(cnt(1)));
```

```
bram1_dout  <=  bram0_din;

bram0_addr(31 downto 10)    <=  (30 => '1', others => '0');
bram0_addr(9 downto 2)      <=  STD_LOGIC_VECTOR(cnt(9 downto 2));
bram0_addr(1 downto 0)      <=  "00";

bram1_addr(31 downto 10)    <=  (30 => '1', others => '0');
bram1_addr(9 downto 2)      <=  STD_LOGIC_VECTOR(cnt(9 downto 2));
bram1_addr(1 downto 0)      <=  "00";
```

Open your MPSoC4Drones Vivado project:

```
vivado ./hdl/projects/u96v2_sbc_mp4d_2020_2/u96v2_sbc_mp4d.xpr
```

Add the `data_copier.vhd` file to the project from the `src/` folder - make sure to untick the "Copy file into project" box. Open the block design and drag the `data_copier` entity into the design. Connect it to the existing BRAM as in Fig. 2.

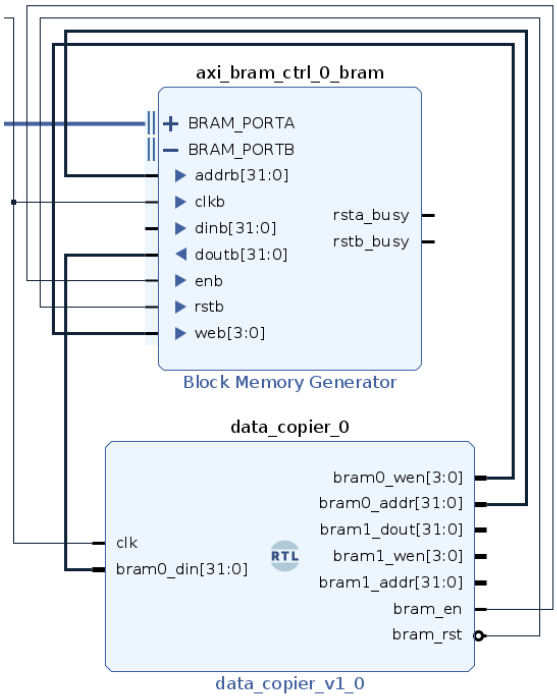

Figure 2: Connecting the `data_copier` to the input BRAM

Copy-paste the `axi_bram_ctrl_0_bram` block to get a new one. Connect it's `PORTB` to the `data_copier` block as in Fig. 3. Add then an instance of the IP `AXI BRAM Controller` to the design and double click it and set the number of BRAM interfaces to 1. Connect the BRAM interface to `BRAM_PORTA` of the second `Block Memory Generator`, as in Fig. 4.

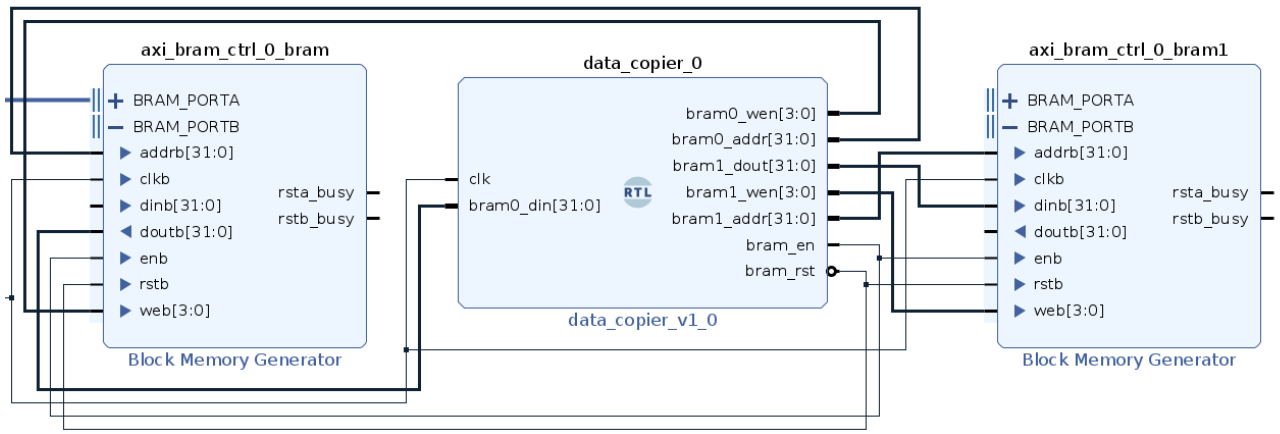

Figure 3: Connecting the `data_copier` to the output BRAM

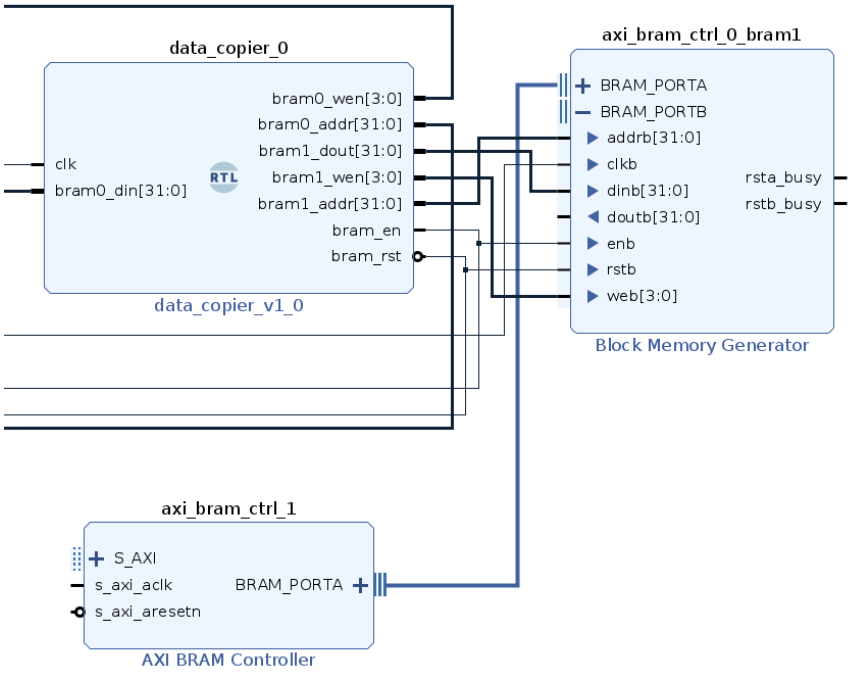

Figure 4: Connecting the `AXI BRAM Controller`

Then, press the green ”Run connection automation” button at the top of the block design view and press ”Ok” in order to automatically connect the second `AXI BRAM Controller` module to the AXI network. Press then the ”Regenerate layout” button, and your design should look like Fig. 5. Open the ”Address Editor” tab in the top of the block design view. You can here left-click the unassigned `axi_bram_ctrl_1` and assign it. In the view, you can assign the base address for entities. Make sure they do not overlap. Afterwards, you can go to the ”Address Map” view in order to inspect you address map.

In a terminal, enter the project directory, source the tools, and build the Vivado project:

```
mp4d-build -V
```

4.2 Vitis

Having built the Vivado project, you can test the functionality in a Vitis bare-metal project. In your project folder, create a `vitis/` folder:

```
mkdir <project directory>/vitis
```

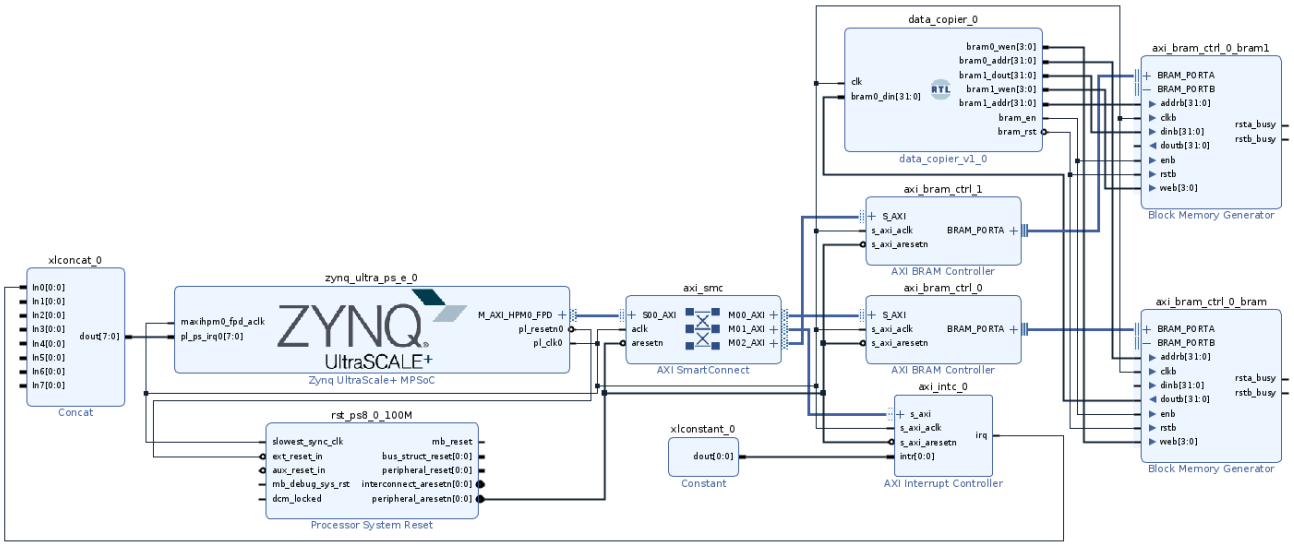

Figure 5: Full block design

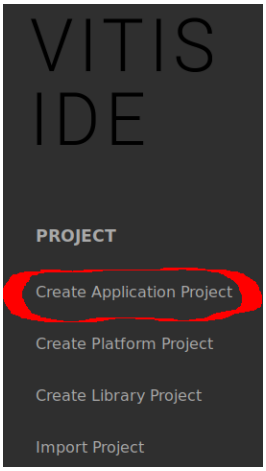

Figure 6: Create new application project in Vitis

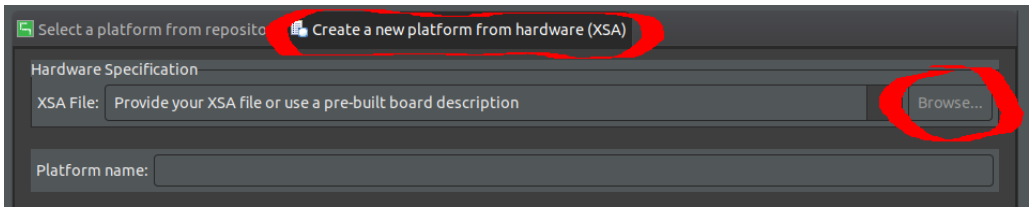

Figure 7: Create new platform project in Vitis

Then open Vitis and select `<project directory>/vitis` as your Vitis workspace. Referring to Fig. 6, create a new application project, press ”Next”, press the tab ”Create a new platform from hardware (XSA)”, press ”Browse”, Fig. 7, and find the XSA-file `<project directory>/hdl/projects/u96v2_sbc_mp4d_2020_2/u96v2_sbc_mp4d.xsa`. Press ”Next”, name your application project `mem_cop_test`, press ”Next”, press ”Next”, select ”Hello World”, and press ”Finish”. If you at a later time have built a new hardware specification (XSA-file), you can right-click the platform project in the Vitis Explorer view and select ”Update Hardware Specification”.

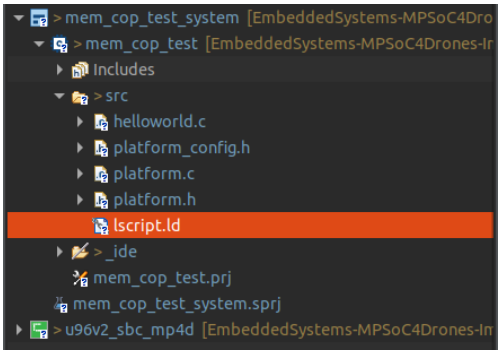

Figure 8: Open linker script

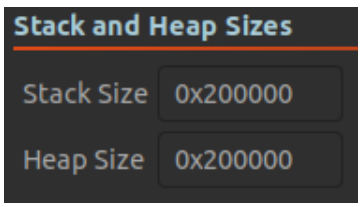

Figure 9: Increase stack and heap size

When creating a new Vitis project for the Ultra96-V2, we need to do two things. First, we need to allocate more stack and heap memory for the application in the linker-script, such that we don't run out of memory - the board has 2 GB of RAM, so no need to be cheap. Double-click the linker script to open it, Fig. 8, and increase the stack and heap size as in Fig. 9. Press "ctrl-s" to save the linker script and close it.

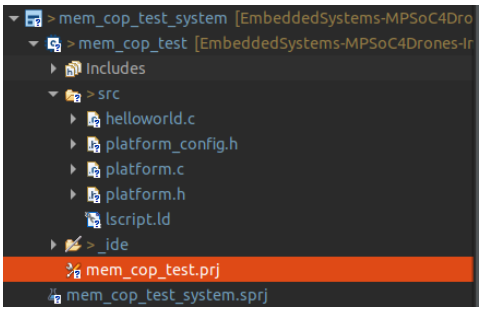

Figure 10: Open project file

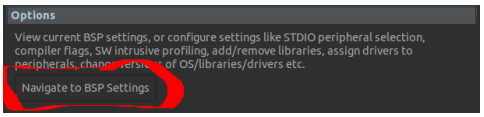

Figure 11: Navigate to BSP Settings

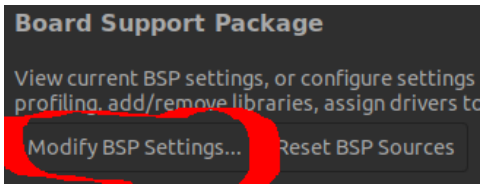

Figure 12: Modify BSP Settings...

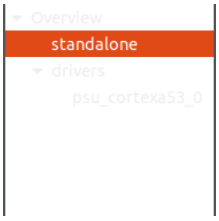

Figure 13: standalone

Figure 14: Modify stdin and stdout

Secondly, we need to modify the board-support-package to use `psu_uart_1` as `stdin/out` in order to output information over the JTAG-to-serial adapter to your computer. Open the project file, Fig. 10, press "Navigate to BSP Settings", Fig. 11, and press "Modify BSP Settings...", Fig. 12. In the left drop-down bar, press "standalone", Fig. 13, and change "stdin" and "stdout" to `psu_uart_1`, Fig. 14. Save and close the files, and press "ctrl-b" to build the project.

You can now write your program in the `src/helloworld.c` file. To access a *single* BRAM in a bare-metal application, include the "xbram.h" file in the main file:

```
#include "xbram.h"
```

Define the memory access macro:

```
#define BRAM(A)      ((volatile u32*)px_config->MemBaseAddress)[A]
```

Define the BRAM objects:

```
XBram            x_bram;
XBram_Config     *px_config;
```

Initialize the BRAM:

```
px_config = XBram_LookupConfig(XPAR_BRAM_0_DEVICE_ID);
int x_status = XBram_CfgInitialize(&x_bram, px_config, px_config->CtrlBaseAddress);
```

The BRAM can then be accessed as follows:

```
uint32_t addr = 0x12
uint32_t data_read = BRAM(addr)

uint32_t data_write = 1234;
BRAM(addr) = data_write;
```

Now, write a test application which tests the functionality by writing to one BRAM and reads from the other BRAM. You need to initialize separate BRAM objects for each BRAM, and use different device IDs in the initialization method call. Output data through serial using `xil_printf(...)`.

4.3 Petalinux

When the functionality has been verified, you are ready to continue with the image build. You need to configure the kernel to load the UIO driver for the two BRAMs in your design. This is done by editing the device tree file. From the project directory, open the user device tree file:

```
gedit src/system-user.dtsi
```

Append the following to the end of the file:

```
// Custom IP map to UIO
&axi_bram_ctrl_0 {
    compatible = "generic-uio";
};

&axi_bram_ctrl_1 {
    compatible = "generic-uio";
};
```

This will tell Yocto to associate the generic UIO driver with the specific hardware modules. Notice that the names match the names of the AXI-connected blocks in the Vivado project.

Re-setup and rebuild from the Petalinux-step by entering the following in the terminal (tools sourced):

```
mp4d-setup -P -f
mp4d-build --petalinux-config -P -U
```

This step will take a while, after completion the fully built image is provided.

4.4 Booting and SSH

Plug the SD-card into the Ultra96-V2, flick the ”boot from SD”-switch, connect to your computer using the JTAG-to-serial converter, power the board, and open a `screen`-session:

```
screen /dev/ttyUSB1 115200
```

assuming the JTAG-to-serial converter shows up on your computer as `/dev/ttyUSB1`. Observe the system output as it boots. First, we need to fix a small bug, in order to be able to use to use sudo and work in the home directory. Login with the credentials `root` and `mp4d`. Enter the following:

```
chown root:root /usr/bin/sudo && chmod 4755 /usr/bin/sudo
sudo chown -R mp4d:mp4d /home/mp4d

exit
```

When the session has exited, you can login with the ordinary credentials `mp4d` and `mp4d`. On your computer, install a DHCP-server:

```
sudo apt install -y isc-dhcp-server
```

Then open Settings, go to Network, next to your ethernet interface press the plus-button to add a new connection profile. Give the connection a name, tick ”Connect automatically”, under both IPv4 and IPv6 select ”Shared to other computers”. Plug the provided USB-ethernet adapter into one of the USB slots of the Ultra96-V2. Connect the ethernet cable to your computer. In the Ultra96-V2 serial session, a ethernet connection can be set up by entering the following commands:

```
sudo nmcli connection add con-name eth-con ifname enx806d97376320 type ethernet
sudo nmcli dev set enx806d97376320 managed yes

sudo mv /usr/lib/NetworkManager/conf.d/10-globally-managed-devices.conf  \
    /usr/lib/NetworkManager/conf.d/10-globally-managed-devices.conf_orig
sudo touch /usr/lib/NetworkManager/conf.d/10-globally-managed-devices.conf

sudo systemctl restart NetworkManager

sudo nmcli con up eth-con
```

You should now have internet connection on the Ultra96-V2 through the ethernet connection to you computer. You can test it by:

```
ping google.com
```

You can check your assigned IP address with

```
ifconfig -a
```

See the IP address found under the `enx806d97376320` interface. You can now open as many SSH session from your computer to your board as you like with

```
ssh mp4d@<IP address>
```

where `<IP address>` is the IP address obtained before. You can now close your serial connection.

4.5 Testing the System

After having logged in to the Ultra96-V2, we can observe the UIO device files by entering into the terminal

```
ls /dev | grep uio
```

One device file will appear per each UIO-mapped hardware device, numbered sequentially. Some of these will be related to the power-management unit. In order to know which device file relates to your AXI-mapped FPGA module, you need to read the name of the UIO class, which is done by entering

```
cat /sys/class/uio/uio<uio number>/name
```

where `<uio number>` is the number of the UIO device you want to check. Check all the numbers. This will never change, so you can note down the numbers with names corresponding to your AXI BRAM Controllers. We will in the following assume that the input BRAM has UIO number 0 and that the output BRAM has UIO number 1.

You will now write two applications corresponding to Process 1 and Process 2 in Fig. 1. If you prefer to write the application on your computer and then transfer it to the board, you can transfer it using the command `scp`. Otherwise, you can develop it directly through your SSH connection.

In your SSH session, create a folder:

```
mkdir /home/mp4d/mem_copy_test
cd /home/mp4d/mem_copy_test
```

For open the BRAM in an easy way, we use a small utility class from GitHub[2]. Pull the source to the folder:

```
git clone https://github.com/DIII-SDU-Group/BRAM-uio-driver.git
```

Inspect and familiarize yourself with the files `BRAM-uio-driver/src/bram_uio.cpp` and `BRAM-uio-driver/src/bram_uio.h`. You will use this class for BRAM interface, and the way it opens the UIO device file is the same method you will use for opening UIO device files in the remainder of the course.

If you are not familiar with the `vim` editor, your can install `nano` which is easier to use:

```
sudo apt install nano
```

Create the writer application:

```
touch writer.cpp
nano writer.cpp
```

Add to the file:

```
#include "BRAM-uio-driver/src/bram_uio.h"

#include <iostream>
#include <string>

int main(int argc, char *argv[]) {

    BRAM writer(<writer BRAM UIO number>,
                <BRAM size>);

    // ...
```

[2] `https://github.com/DIII-SDU-Group/BRAM-uio-driver`

```
    std::cout << "Hello world!\n";

    return 0;

}
```

Modify the application to properly write a sequence of data to the BRAM. You can build the writer application by

```
g++ writer.cpp BRAM-uio-driver/src/bram_uio.cpp -o writer
```

and run it by

```
./writer
```

Write the reader application in a similar fashion to read from the second BRAM and perform a test of the system. Bonus: Make the writer application modify the contents of the BRAM each second and make the reader application output the data from the second BRAM to the terminal each second. Open a second SSH session, run the writer application in the first session, and run the reader application in the second session.