UNIVERSIDAD
POLITÉCNICA
DE MADRID

Heterogeneous HW/SW acceleration of a video game in a re-configurable MPSoC

Master Thesis

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Centro de Electrónica Industrial

Master in Industrial Electronics

February 2019

Author:

David Lima Astor

Supervisors:

Eduardo de la Torre

Leonardo Suriano

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

RESUMEN

La evolución de la tecnología en la última década ha provocado la aparición de los llamados MPSoCs. Estos nuevos dispositivos computaciones se caracterizan por incluir varios elementos de procesamiento tanto *software* como *hardware* dentro de la misma oblea de silicio. Debido a esto, son denominados sistemas heterogéneos, y su uso ofrece grandes ventajas, como la aceleración de aplicaciones que hacen un uso óptimo de cada tipo de procesamiento. Debido a esta complejidad intrínseca, la metodología de diseño que se utiliza para implementar aplicaciones es compleja y requiere del uso de herramientas nuevas de co-diseño, como son las herramientas de Síntesis de Alto Nivel (HLS).

El Proyecto Fin de Máster tiene como objetivo reproducir esta metodología con le objetivo de implementar el juego DOOM en el kit de evaluación zcu102 de Xilinx, que incluye una Zynq UltraScale+ MPSoC. Para ello se realizan los pasos que se muestran en la figura A.

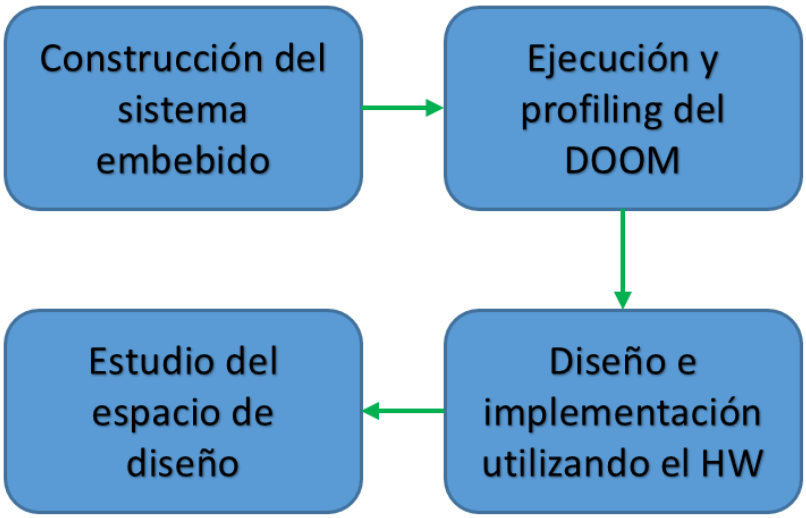


Figure A.: Etapas del proceso de diseño realizadas en el Trabajo Fin de Máster

Construcción del sistema embebido Creación de un sistema basado en Linux para la administración de los recursos de la plataforma. Para ello se utilizan las herramientas de PetaLinux y Yocto, que gracias a su uso permiten crear un script personalizado que genera todo el sistema. Debido a esto, puede ser utilizado y seguir desarrollándose para nuevos trabajos futuros.

Ejecución y *profiling* del DOOM Ejecución del juego como si se tratase de un programa *software* más en los microprocesadores que están ejecutando el sistema operativo Linux. Realización de un *profiling* para identificar las tareas o funciones que más se ejecutan.

Diseño e implementación utilizando el *hardware* Implementación de la función identificada con mayor uso de la CPU en la estructura *hardware* o FPGA del dispositivo. Ejecución del juego utilizando la función aceleradora.

Estudio del espacio de diseño Exploración del espacio de diseño a partir de las diferentes configuraciones que se pueden realizar del acelerador *hardware*. Realización de una comparación entre las aceleraciones frente a potencia consumida por la plataforma.

Estructura temporal del trabajo

La figura B corresponde con el gráfico Gantt correspondiente a la realización del proyecto. Del que se extraen los siguientes sucesos:

- Gran duración del período de PetaLinux, debido a que también se cuenta el período de familiarización con la plataforma zcu102. Además de la realización de tutoriales y guías para poder generar el sistema.
- Intento de uso de Yocto para corregir las limitaciones de PetaLinux.
- Diseño del script que genera todo el sistema, a partir de los pasos realizados por las herramientas anteriores.
- Ejecución rápida del juego y su análisis gracias a que se contaba con la experiencia de su realización en un computador de propósito general.

- Diseño de la implementación *hardware* lenta debido a las iteraciones con las herramientas de Síntesis de Alto Nivel y familiarización con el uso de un bloque DMA.

- Ejecución de la exploración del espacio de diseño, requiriendo generar muchos archivos binarios. Las medidas de FPS requirieron de cambios adicionales en el código del juego.

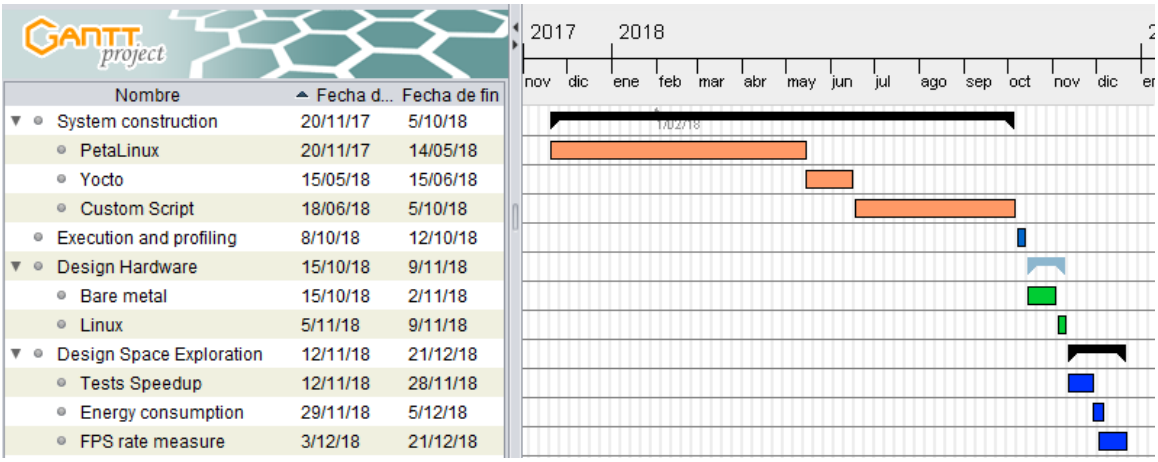


Figure B.: Gráfico Gantt de la evolución temporal de las tareas del proyecto

SUMMARY

The evolution of the technology in the last decade has led to the appearance of the so-called MPSoCs. These new computational devices are characterised by including several processing elements, both software and hardware, within the same silicon wafer. Due to this, they are called heterogeneous systems, and their use offers great advantages, such as the acceleration of certain tasks, that make optimal the use of each type of processing. Due to this intrinsic level of complexity, the design methodology used to implement applications is complex and required the use of new co-design techniques, such as the High-Level Synthesis (HLS) tools.

This Master Thesis aims to reproduce this methodology with the main objective of implement the DOOM game using the Xilinx zcu102 Evaluation Kit, which includes a Zynq UltraScale+ MPSoC. To do so, the steps showed in figure C.

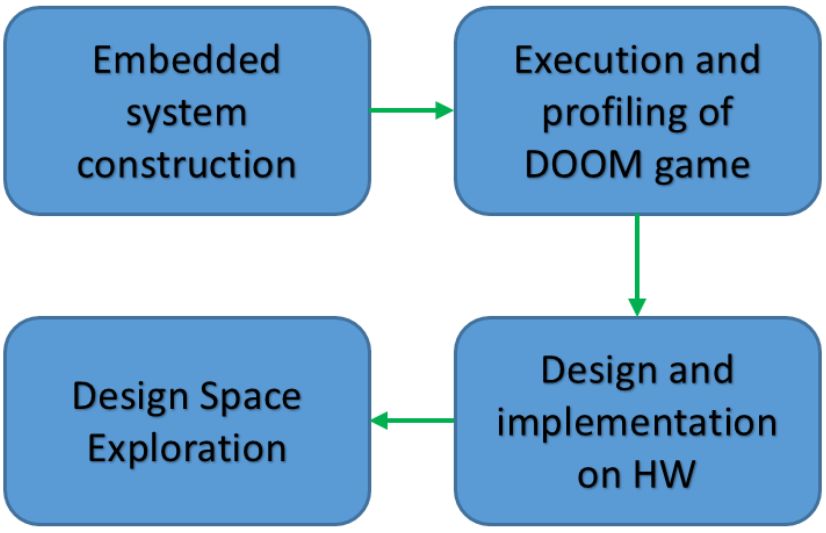


Figure C.: Stages of the design process carried out in the Master Thesis

Embedded System Construction Creation of a Linux-based system for managing the resources of the platform. For this, PetaLinux and Yocto tools are used, and thanks to their use allow the creation of a custom script that generates the entire system. Because of this, the image of the embedded system can be used for future lines of work.

DOOM Execution and Profiling Execution of the game using the microprocessors of the device as it is another program running on the Linux Operating System. Profiling of the game in order to identify the tasks that occupy the CPU the majority of the time.

Design and implementation on HW Implementation of the identified function with the greatest use of the CPU resources in the hardware fabric or FPGA of the device. Execution of the game using the accelerated hardware module.

Design Space Exploration Exploration of the design space from the different hardware configurations that can be made of the accelerator. Making a comparison between the achieved speedups and the total consumption of the platform.

Temporary structure

Figure D corresponds to the Gantt Chart related to the realization of the Master Thesis. From which the following events are extracted:

- Great duration of the PetaLinux period, caused by the familiarisation process with the zcu102 platform. In addition to the realization of tutorials and guides to be able to generate the embedded system.
- Attempt to use Yocto to correct the limitations of PetaLinux.
- Design of a script that generates the whole system, from the steps carried out by the previous tools.
- Fast execution of the game and its profiling thanks to the fact that it was already done in a general purpose computer.

- Slow design process of the hardware implementation due to the iterations with the High-Level Synthesis tools and the familiarisation with the DMA block.
- Execution of the Design Space Exploration, which required the regeneration of a lot of binary files. The FPS measurements have required also additional code changes in the game.

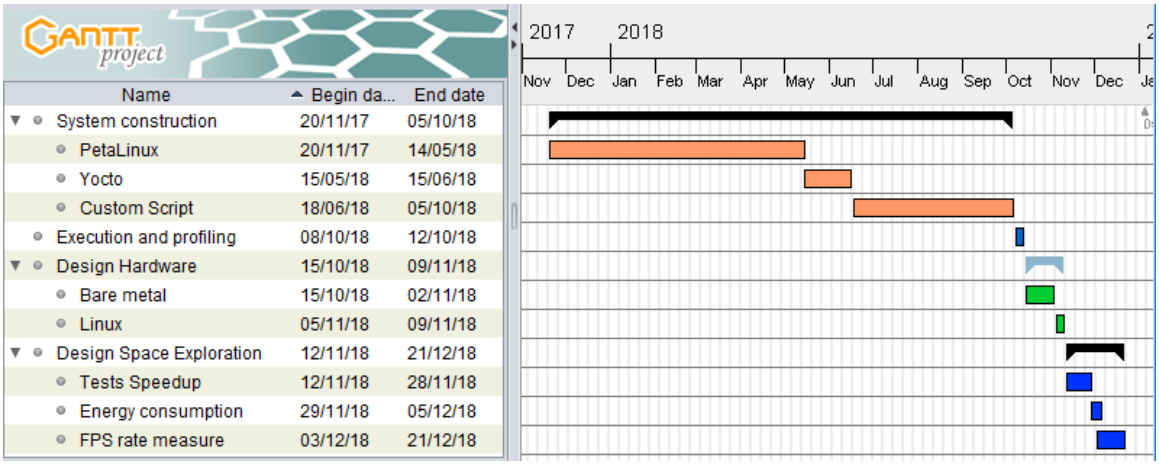


Figure D.: Gantt chart of the temporal evolution of the Master Thesis

1. INTRODUCTION

During the last decades, embedded systems have evolved thanks to the evolution of the technologies (figure 1.1), allowing the manufacture of silicon wafers with an increasing number of transistors. One of the results of this evolution is the appearance of the *Multi-Processing System-on-Chip* (MPSoC), which are called heterogeneous systems because they include both software and hardware processing elements within the same silicon wafer.

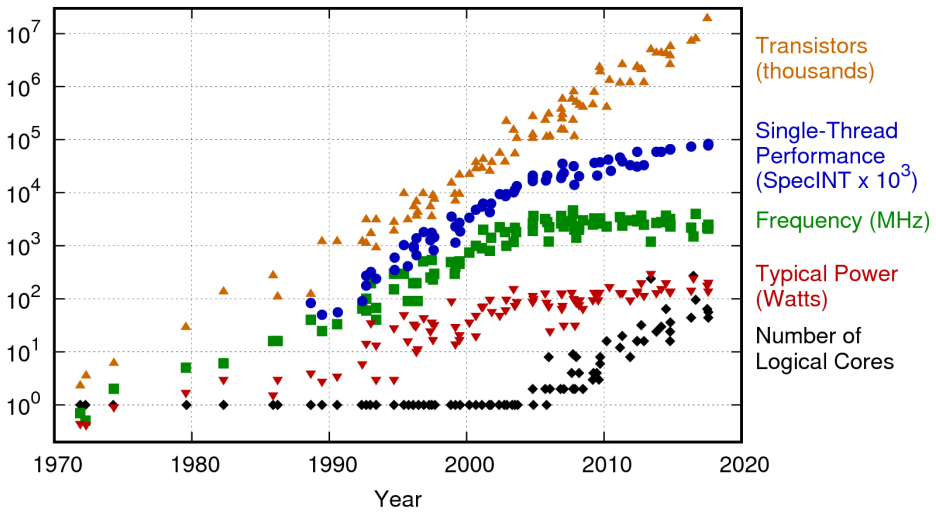


Figure 1.1.: Trends in the evolution of microprocessor technology in the last 42 years [1]

As a result of this increasing complexity, it is necessary to use complicated tools and design techniques to use these systems optimally. This causes that the designers have to dedicate more time to explore the design space and study the different options. Despite this, with the use of these systems, numerous advantages are obtained such as the acceleration of processes or greater reliability in the result of an operation. Due to these reasons, heterogeneous MPSoC systems are starting to be used in fields where there are hard requirements of reliability and latency, such as the aerospace or the biomedical.

This Master Thesis aims to show the methodology that must be carried out in order to design an application using these new heterogeneous devices. To do so, a video game is used to identify the task that is executed the longest, in order to implement it using another processing element, such as the FPGA. The process ends with the study of the design space of this implementation.

The objective of this chapter is to introduce the reader in the context of the Master's Thesis. First, the main concepts related to the project are defined. Next, the motivation and its main objectives are explained. Finally, the derived contributions and the structure of the document are presented.

1.1 Heterogeneous Systems and MPSoCs

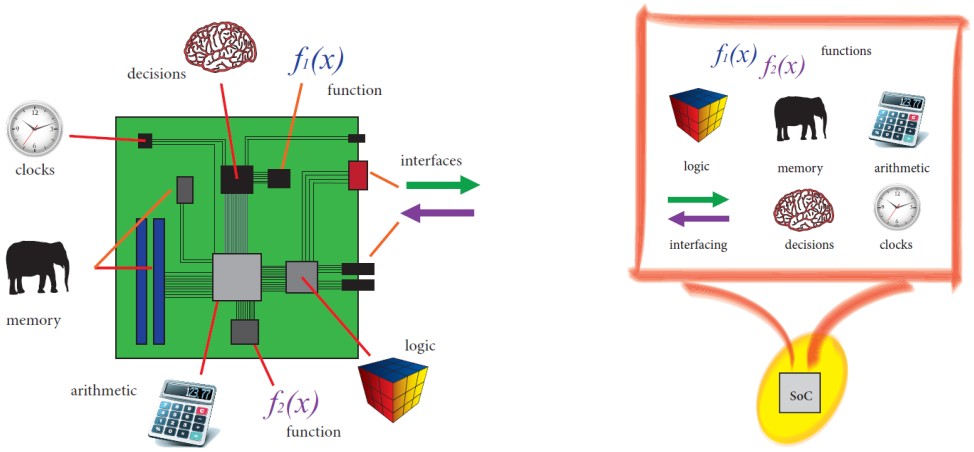
Heterogeneous systems are those that use both software and hardware processing elements in the execution of their tasks. These systems are characterised by taking advantage of the best of each type of processing:

- Software: Use of microprocessors for the execution of tasks mainly of logical condition or data movements, such as the evaluation of a variable and decision making.
- Hardware: Use of programmable logic for the execution of arithmetic tasks on data in parallel and ideally allowing concurrent access. An example of this are graphics tasks and image processing, such as the application of a filter on an image.

These systems have been evolving since they were based on *Printed Circuit Boards* (PCB) that contained microprocessors and *Field-Programmable Gate Array* (FPGA) in different silicon chips (figure 1.2(a)). Nowadays, thanks to the evolution of technology, these components can be included in the same silicon chip, resulting in what is known as *System-on-Chip* (SoC) (figure 1.2(b)).

When there are programmable elements in this SoC, it is renamed to *System-on-Programmable-Chip* (SoPC), and at the moment in which several processing blocks that can be software or hardware are included, it becomes an MPSoC.

We must emphasise that the MPSoC platforms allow the reprogramming of their resources so that they can perform several different tasks after their deployment. There are currently new areas of research related to the implementation of run time



(a) Traditional PCB (b) SoC

Figure 1.2.: Comparison between PCB and SoC devices [2]

FPGA reprogramming processes and the implementation of evolutional and adaptable hardware [3] [4].

The task related to the programming of these devices is very complex because it involves various levels of abstraction. This is because there are elements programmed in hardware description languages, like VHDL, and also in high-level languages, like C/C++ or Python. Therefore, in the design methodology that is executed in these platforms, the use of *High Level Synthesis* (HLS) tools is very important [5].

High-Level Synthesis tools

The High-Level Synthesis is an automated process that creates, from an algorithm description, the digital hardware that implements this behaviour. These tools are designed to break the productivity gap [6] that produces the design of low-level abstraction that belongs to hardware design.

These tools take advantage of the powerful and fast description of algorithms of the high-level languages, at cost of losing the optimisation offered by the low-level design. Despite this disadvantage, there a lot of good things with the use of these tools [7], such as the hard reduction of the time to design a prototype application.

1.2 Motivation and Main Goals

The MPSoC platform allows executing the tasks optimising the different steps that must be carried out. In this way, the software resources must be used for control and decision making tasks, while the hardware is used for the implementation of arithmetic instructions. The correct use of these elements results in optimisation in terms of power consumption and latency, as well as a possible general acceleration of the application. Due to all these advantages, they are starting to be used in fields where strict latency and reliability specifications are required, such as the aerospace and biomedical areas.

The main objective of this project is to implement the DOOM game on the Xilinx ZCU102 platform [8], using the microprocessors for the control flow, and the FPGA to implement the task identified as the most executed in the game. Therefore, the main objectives of the project are listed below:

- Prepare and deploy a Linux-based system on the Xilinx ZCU102 platform, which must include a package manager and a desktop environment.
- Run the DOOM game application using only the microprocessors. Make the profiling to identify the tasks or functions that occupy the CPU the majority of the time.
- Implementation of the identified tasks using other resources, freeing the CPU load and trying to accelerate DOOM execution.
- Exploration of the design space of the application, making a comparison between the achieved speedups and the total consumption of the platform.

1.3 Contributions

The main contributions of this Master Thesis are:

- The creation of a custom Linux-based Operating System to run upon a Zynq UltraScale+ platform in order to manage, also, hardware accelerators. The entire process has been automated in a `bash` script, resulting in an image to mount on an SD card ready to use. This system also provides a package manager and a desktop-like environment to be user-friendly.

- The analysis of the execution of a video-game (frames per second vs. energy consumption) on the platform running the created custom OS.
- The generation of a hardware IP for accelerating graphics functions of a video-game.
- The identification a method to split a single video frame in a set of independent slices: each of them can be so processed in parallel to the others. The Design Space Exploration is so carried out taking into account another parameter: the number of hardware accelerators in the FPGA.

1.4 Document structure

The document is composed of 5 chapters and 1 appendix. It is organised as follows:

- Chapter 2 describes the system and its components.
- Chapter 3 explains the methodology followed in order to achieve the main goals.
- Chapter 4 describes the tests performed and presents the results obtained.
- Chapter 5 presents the conclusions obtained and the possible future lines of research that can be developed from this project.
- Appendix A includes the code of the custom script used to generate the embedded operating system.

2. SYSTEM DESCRIPTION

The purpose of this chapter is the description of the used platform and its components. First, the device and its most important components are presented, each of them with a brief description. Next, the embedded software that runs on the board is described from the stages bootloaders to the Operating System that runs on it. Finally, the version of the game used in the project is justified.

2.1 Xilinx Ultrascale+ MPSoC platform

Xilinx is the manufacturer of the Zynq family of devices, which are the new generation of *All Programmable System On Chip* (APSoC). This group of chips have the characteristic of combining "hard" processors such as the Arm Cortex™-A53 with the traditional FPGA programmable logic. The inside architecture is completed with industrial *Advanced eXtensible Interface* (AXI) interfaces, which gives the opportunity of communicating with low latency and high bandwidth. This interface allows each block to give its best, without the communications becoming bottlenecks [2].

Figure 2.1.: Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [8]

To make the project, the ZCU102 Evaluation Kit (figure 2.1) based on the Zynq Ultrascale+ XCZU9EG MPSoC has been used. This platform allows rapid prototyping of a multitude of applications that may be related to automotive, industry, video or communications [9].

All Zynq devices have the same architecture formed by the *Processing System* (PS) and the *Programmable Logic* (PL), being the first based on at least one dual-core processor Arm Cortex-A9 and the second based on the structure of an FPGA. The most important elements of the device are represented in figure 2.2 and are briefly described below [10]:

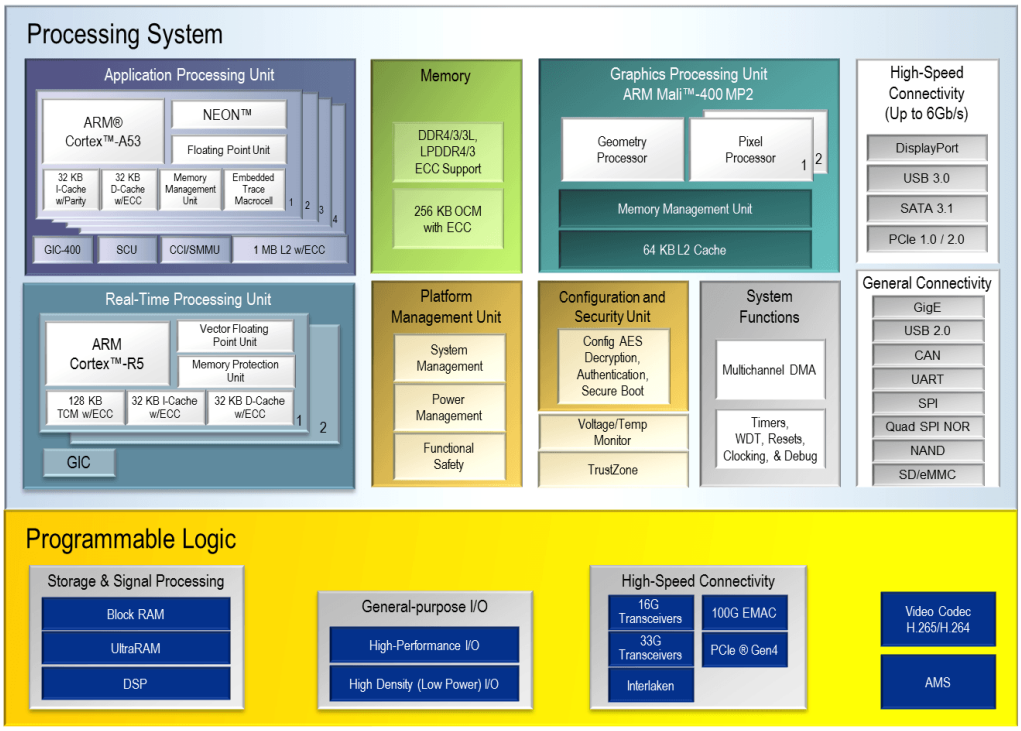


Figure 2.2.: Zynq UltraScale+ structure [8]

- **PS**: Based on a 64-bit Quad-Core Arm®Cortex™-A53 and a dual-Core Arm Cortex-R5. The most important components by which it is composed are:

  - APU based on Arm Cortex-A53 (figure 2.3). Built on application grade "hard" processors, capable of running complete Operating Systems such as Linux. Each of these processors is associated with the following computational units: (1) A NEON *Media Processing Engine* (MPE) and *Floating*

Point Unit (FPU); (2) a *Memory Management Unit* (MMU); (3) level 1 cache memories. The APU has also a level 2 cache memory, and numerous *On-Chip Memory* (OCM). Finally, there is the *Snoop Control Unit* (SCU) that forms the bridge between the Arm cores, with the cache and OCM memories. This last unit is responsible for the connection interface with the PL.

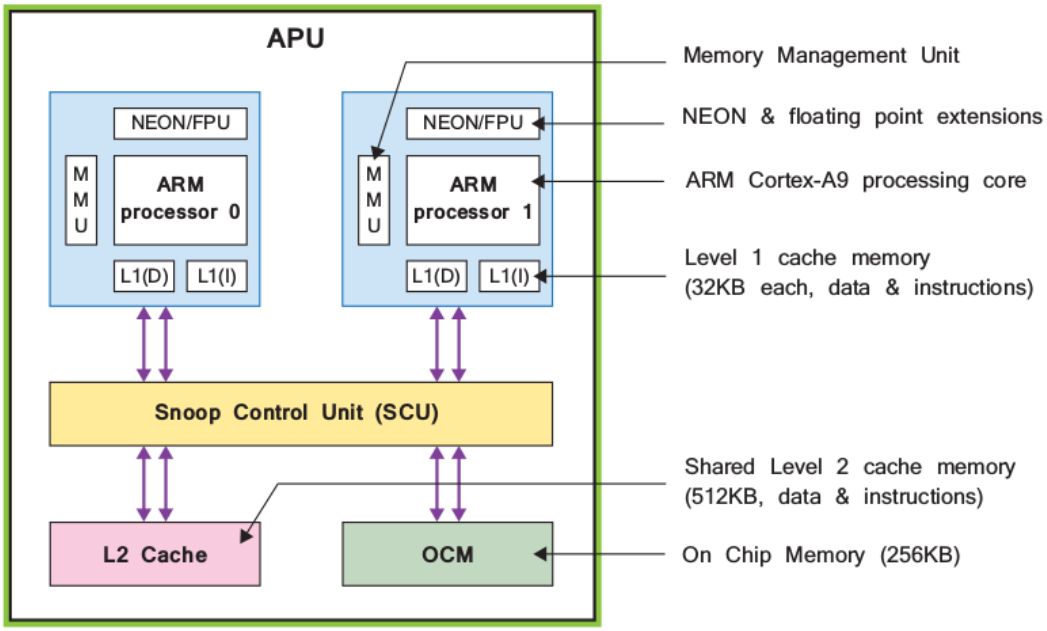


Figure 2.3.: Simplified diagram of the APU inside the Zynq UltraScale+ [2]

- *Real-Time Processing Unit* (RPU) based on a dual-core Arm Cortex-R5. Processors capable of running Real-Time Operating Systems, since they are designed for applications with strict real-time and reliability requirements. The possibility of execution in "lock-step" must be highlighted: it consists in the execution of the same instruction by both processors in different periods of time, thus being able to compare and verify the results.
- Arm Mali-400 *Graphics Processing Unit* (GPU). To accelerate the execution of graphics tasks and image processing functions.
- *Power Management Unit* (PMU). It is a dedicated user-programmable processor for initialisation of the system prior to boot, power management and system error handling of the platform.

- **PL**: Area based on the UltraScale architecture [11]. In this region, we can place both official *Integrated Property* (IP) blocks provided by Xilinx and own modules designed for a specific application. These last can be hardware described in VHDL or Verilog, or can be generated using HLS tools from a description in high-level languages.

Due to the availability of these elements, the platform becomes an ideal prototyping device for practically any application. But also the design process becomes complex and involves a lot of software components to configure and manage the platform. Thus, the availability of an Operating Systems to control this hardware becomes something important in a lot of applications.

The following section describes the parts that make up a system based on an Operating System on the board. Specifically, the case of using a Linux-based system on the platform is explained.

2.2 Embedded Software

The use of an embedded operating system in this type of platform adds a series of advantages associated with the increase of the level of abstraction over the hardware since everything happens to be controlled and managed by the OS kernel. The main advantages of this fact are listed below, and are due fundamentally to the possibility of reusing applications already developed:

- Reduction of **time to market**.
- Make Use of **Existing Features**.
- Reduce **Maintenance and Development Costs**.
- Easy possibility of **updating** and **adapting** to new features and specifications. Since the embedded system makes it easier to reprogram its elements after the deployment.

This section aims to describe the elements that make up the software integrated with the platform. First, the boot process and its stages are explained. Next, the OS is described briefly and the main reasons why Linux has been used are presented.

2.2.1 Booting Process

The booting process of the platform consists in the execution of a series of stages that, in the case of a Linux-based system, have the order defined in figure 2.4. This section is dedicated to the definition of each of these steps and to the description of the most important characteristics related to the work.

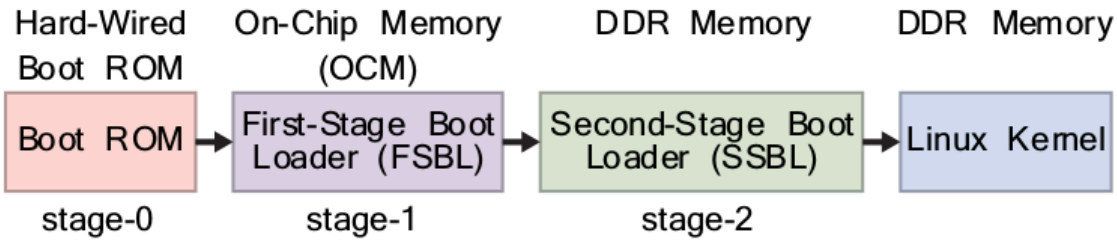


Figure 2.4.: Steps in the booting process on Zynq devices [2]

In order to complete the process, it is necessary to have a series of files in the boot medium, which in the project has been defined as an SD Card divided into two partitions, one for the boot files and the other for the root file system (figure 2.5). The files corresponding to the first partition are:

- `BOOT.BIN`. Created from the *Boot Information File* (BIF), it includes the binary files (`.elf` format) used during the boot stages, and the information on how they are executed, such as where they are placed and which are their exception levels. For the ZCU102 platform, the following files are included:
  - `FSBL.elf` and `SSBL.elf`. *First State Boot Loader* (FSBL) and *Second State Boot Loader* (SSBL) binary files.
  - `bl31.elf`. This file is mandatory only for the Zynq UltraScale+ MPSoC. It includes the *Arm Trusted Firmware* (ATF) which provides a reference implementation of secure world software for the Arm architecture [12].
  - `pmufw.elf`. Low-level firmware executed in the PMU.
- `image.ub`. It corresponds to the compressed Linux kernel image.
- `devicetree.dtb`. It contains information about the hardware of the board where the kernel is going to be booted.

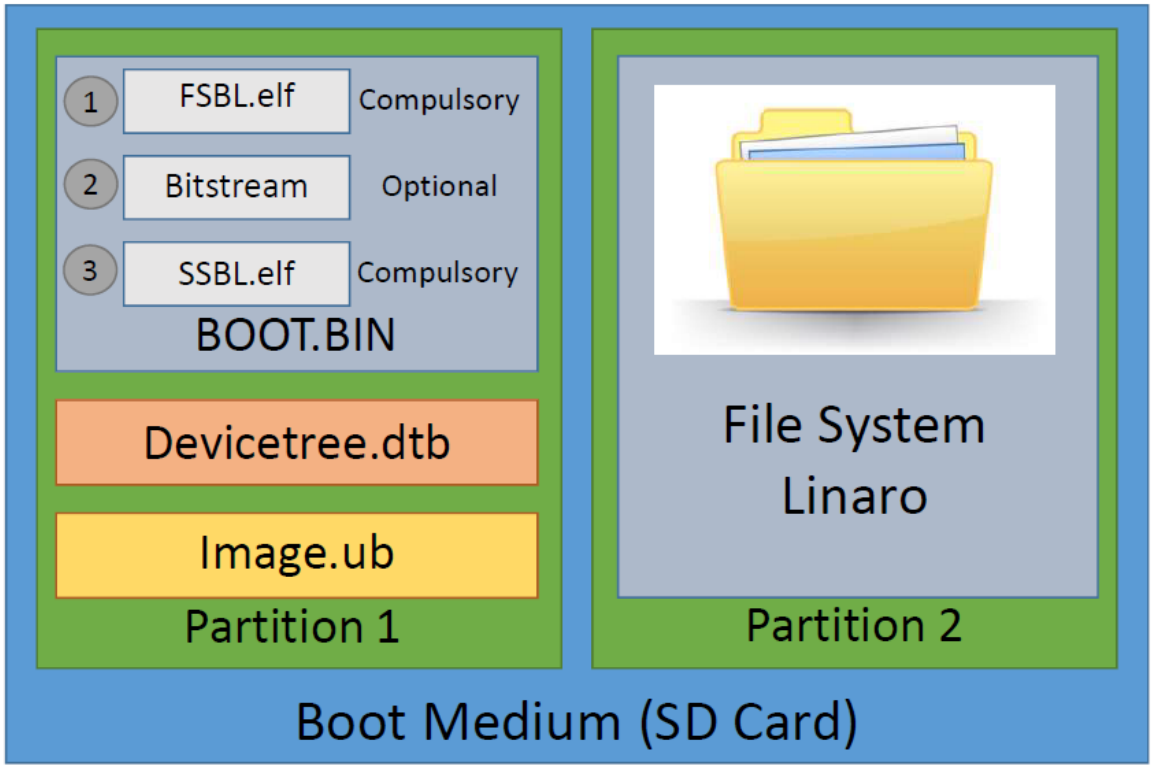


Figure 2.5.: Structure of an SD Card defined as Boot Medium

The steps corresponding to the booting process of the platform are described in the next subsections.

Stage 0 - Boot ROM

The function of the Boot ROOM is the loading and start of the FSBL, whose location is determined by some pull-up or pull-down values fixed by specific pins of the board. In the case of the ZCU102, these signals are defined by the `SW6` switches, whose options are included in the table 2.1.

The boot medium has been chosen as an SD Card, so the resulting configuration of the `SW6` pins are shown in figure 2.6.

Stage 1 - First State BootLoader

The FSBL is loaded by the Boot ROM in the OCM and is responsible for the execution of a large number of initialisation functions which include the configuration

Table 2.1.: Switch `SW6` possible configurations

Boot Mode	**Mode SW6 [4:1]**
JTAG	on, on, on, on
QSPI32	on, on, off, on
SD Card	off, off, off, on

Figure 2.6.: Switches that select the booting medium in the ZCU102 as an SD Card

of the CPU with the PS configuration data, programming the PL (if a bitstream is included in the boot medium) and the loading and execution of the SSBL.

Stage 2 - Second State BootLoader

This stage depends on the type of OS that will be used. In the case of a standalone application, the code is loaded and executed directly. But in the case of this project, as Linux is used, this stage will correspond to the U-Boot which is an SSBL.

The U-Boot [13] is a popular universal bootloader used by the Linux community, which has been customised by Xilinx for use in the Zynq platforms [14]. It is responsible for loading the OS, which is stored on the disk, into the memory. Once the Linux kernel and the device tree are loaded, it starts the kernel execution.

2.2.2 Operating System

The Quad-Core Arm Cortex-A53 allows having an Operating System. Although this is not necessary for all embedded systems, it provides numerous advantages, such as the increase of the levels of abstraction in the design of applications and the management of the platform.

The choice of the OS depends fundamentally on the specifications of the application, and in the case of the present project, a Linux-based OS has been chosen due to the following reasons:

- The non-existence of real-time requirements in our application.
- The experience in its use in similar applications.
- The availability of drivers for the hardware.
- The support that is offered on the internet, as official guides and documentation.

The elements that are required in the use of Linux on the platform are described below, and they are also included in the boot medium already showed in figure 2.5.

Device Tree

The device tree is a data structure that describes the hardware where the kernel is going to be executed since the same kernel can be used in different machines.

The Xilinx Design Tools [15] allows to automatically generate the Device Tree Blob (`.dtb` file) corresponding to the ZCU102 platform, and this has been used as a starting point in the realization of the project. From this file, the following modifications are made:

- The possibility of interruption from the PL.
- The corresponding SDSoC [16] node is added, which will add support to the applications that have been developed with this set of tools.
- Patches are applied to the corresponding node of the display port, which is a standard interface for video transmission that will be used to connect with the monitor. This corrects an error related to the limitation of resolution in the desktop environment.

- The definition of the bootargs, which are the arguments that are passed to the kernel at the beginning of its execution. For example the definition of the location of the root filesystem as the second partition of the SD card.

Linux Kernel

The most important function of the OS kernel is to abstract what concerns the hardware and to arbitrate access to shared physical resources, such as:

- CPUs: Concurrent execution of multiple programs and threads.
- Memories and hard drives: Communication between the programs and the file system.
- Input/output devices (mouse, keyboards ...).

This makes the software on top of the OS more independent of the hardware, which results in the development of applications much more easily, more portable to other platforms and more secure.

In this work the Linux kernel that provides Xilinx [17] customised for its use with the Zynq devices is used. In addition to the configuration defined by default for the ZCU102 platform, the following characteristics need to be modified in order to achieve the main goals of the project:

- Enable drivers for mouse and keyboard devices. Since these peripherals are needed by the user to control the desktop environment.
- `Xilinx APF Accelerator Driver`. Enable these drivers will allow the use of *Direct Memory Access* (DMA) hardware modules. This is also related with additional support to execute the binaries generated by the SDSoC tools.
- Enable the `Kernel function tracer` feature, which is a powerful resource to the debug the kernel.
- Disable the support for `Initial RAM filesystem`, because a partition of the SD card is used as a file system.

- Disable support for the `PCI BUS`. The board does not have a *Peripheral Component Interconnect* (PCI) bus, and it is a recommended action to avoid problems. This action appears in a Xilinx tutorial [18] that helps to build the kernel and the boot files to deploy an Ubuntu-based system.

File System

The file system refers to all other components of the user space of the Operating System. It contains, at least, the shared libraries used by the software, the scripts that are executed at the end of the boot process and a terminal emulator which allows the communication the user with the system.

In the project, a Linaro file system has been used, which is based on the Debian distribution. This choice is taken mainly for the following reasons:

- The fact that Linaro's goal is to provide an open-source ecosystem to ARM architectures to eliminate redundancy in the industry and reduce development efforts [19].
- It has a package manager, which improves the reuse of the system since it can be updated with new libraries and programs.
- It is based on Debian distribution, which is one of the most used Linux distributions. This means that the libraries are constantly updated and it has great support from the Linux community.

Although this system does not have a desktop environment, the package manager is used to install it once the system is running on the platform. Thus, another goal of the project is fulfilled. In addition, the controllers associated with the MALI 400 GPU [20] [21] need to be added in order to make it works.

2.3 Crispy-DOOM

The DOOM is a game released in 1993 that consolidated the first-person shooter genre. It is coded in C language, and it was mainly developed for computers. But, following the release of the code in 1997 [22], which is usually known as the Vanilla DOOM version, it has been ported to numerous platforms by users [23].

The Crispy-DOOM [24] is one of these source ports adapted by the users and has been chosen mainly due to its similarity with the original release of the game. It is a friendly fork of another important source port called Chocolate-DOOM [25], which is included in Debian official repositories so it can be installed through the `APT` package manager. The project started using it, but it keeps the characteristic of the Vanilla DOOM of limiting the *Frames Per Second* (FPS) ratio to 30, and this restricts the possibilities of acceleration. Crispy-DOOM, on the other hand, corrects this limitation and at the same time maintains the similarity with the original DOOM.

Although the source code has been released, the graphics contents of the game, such as the different episodes and the sound content, are not free. Despite this, there is a shareware version which consists of a small enough content to carry out research projects and demos [26].

3. DEVELOPMENT AND METHODOLOGY

This chapter describes the development and methodology followed to achieve the main goals of the project, which are already defined in the section 1.2 of the introduction chapter. First, the construction of the system that is deployed in the platform is described. Next, the execution of the game and its profiling in the device is described. Finally, the steps followed in order to implement in hardware the identified function are explained.

3.1 System construction

The first main goal to achieve is the construction of the system that will be deployed in the platform. During this process, different tools were used to generate the required files, because some of them did not fulfil the requirements that we needed.

At the beginning of the project, PetaLinux, which is a set of tools provided by Xilinx [27], is used. Although, due to the absence of some needed features, the construction of the file system is changed to use the Yocto Project. Using this last tool, a lot if incompatibilities are found that forced to abandon its use. However, thanks to the experience gained using this both tools, it is proposed to build the components separately, first imitating the steps executed by PetaLinux, and next adding the required features.

In this section each of the used tools is described, and also the steps followed in order to generate the components of the system, whose functionality is already defined in the previous chapter.

3.1.1 PetaLinux

PetaLinux is a Linux tool provided by Xilinx that offers everything necessary to build and deploy Embedded Linux solutions on Xilinx processing systems. Through a command-line interface, it allows developers to customise the boot loader, Linux kernel, file system, libraries and system parameters, making all the required steps in

order to achieve the desired configuration. Due to this, it is a very good choice if you want to quickly prototype an application. We can resume the advantages of its use with the following ideas:

- It makes automatically the entire process of generating the necessary files to deploy the system. So you can quickly prototype an application for a Xilinx platform.
- It enables synchronising the software system with the hardware design while adding new features. When something is changed, it is not necessary to rebuild all the project, only the affected component is updated.

The main consequence of this advantages is that the tool makes a lot of steps hidden for the user. This causes a loss of control of the process and the difficulty of debugging the system in case it does not work correctly. For example, it deals with the addition of the features required to accomplish the user's specifications, which could not be desirable at all. The additional problems that appeared when we have used it are:

- The desktop environment that can be included in the filesystem is very simple.
- The created filesystem is very limited, and it is not based in any official distribution, so if it is wanted to use a package manager, like *Advanced Packaging Tool* (APT) from Debian, it is necessary to create a local repository with the libraries, and be aware of the dependencies.

PetaLinux uses the official repositories to create the different components of the system. For example, it downloads the Linux kernel repository customised by Xilinx and applies the modifications before building it. This process is repeated for all components of the system and is based on the Yocto Project. In the next section, the actions that are carried out in order use this build system to add the missing features to the filesystem are explained.

3.1.2 Yocto and PetaLinux

The Yocto Project [28] is an open source collaboration project supported by the Linux Foundation. It helps to create custom Linux-based systems, regardless of the

hardware architecture. The project provides a flexible set of tools and space where embedded developers can share technologies, software stacks, configurations and the best practices when deploying Linux-based embedded systems.

What distinguishes Yocto Project from any other build system, is the development model called the Layer Model. This model is based on splitting the configuration instructions into a series of layers. The fact that a layer can contain instructions to overwrite a previous configuration make possible to use layers provided by the community and adapt them to make suitable for your application. This powerful capability is the reason why it is said that the Layer Model is designed to support both collaboration and customisation at the same time [29].

There is a layer called `meta-debian` [30] which contains a set of recipes in order to cross-building a filesystem with Debian distribution source packages. This would allow adding a great and solid base to the PetaLinux project filesystem, and the disposition of a package manager to install a desktop environment.

The problem that forces us to abandon this attempt is that this new configuration layer, provides a lot of configuration instructions that are incompatible with the `meta-petalinux` layer, which is a "distro" layer [31]. This layer is mandatory in the PetaLinux Tools, so it is impossible to replace it with the `meta-debian`.

The generation of an independent filesystem is also tried, but problems appear because PetaLinux really introduces files of necessary drivers in the file system, such as the Mali GPU drivers, which are mandatory to launch a desktop environment.

These tools have the characteristic of being open source. So the recipes and the configuration files that are applied in the process of compiling the MALI Drivers can be read. Thanks to this, the changes that must be made in the source files to compile them for the platform are known and can be applied in order to generate both the userspace libraries and the kernel module needed to handle the GPU.

At this point, the possibility of automating the entire process of creating the required files into a single script is taken into account. Moreover, the production of an image of the SD Card as the output is evaluated, because it is a great contribution to use in future lines of work. The next section is dedicated to this development.

3.1.3 Custom Script

The time spent using the previous tools is considerable. Each time that something is changed produces the rebuild of the system. All these steps are only performed to prepare the platform to execute and profile the doom. Due to these reasons, the benefits of making a script to create the SD Card image automatically begins to be important, because the future developers that want to create a similar system will save a lot of time using it. This fact makes this script an important contribution to the project. This section is dedicated to the description of this script, whose code is in the appendix A. At the time of writing this document, the code has not been published in a repository due to the use of files that are not yet public, so the steps that include these files are not explained.

The main objective of the script is to create an SD Card image ready to be mounted and placed in the platform. To achieve this, the script downloads and compiles the official repositories from Xilinx in order to obtain the executable files that will be placed in the SD Card partitions. In the case of the filesystem, it downloads and modifies a Debian-based Linaro from its website, building and adding the drivers to use the Mali GPU based on the recipes applied by PetaLinux. The place of all this elements are represented in figure 2.5 in the section 2.2.1 of the previous chapter.

The structure of the script is based on the `install.sh bash` script from the ReconOS "getting started" website [32]. But in this case, instead of the change line by line the source files, patches are used to apply all the modifications with one command. In addition, it is split into shorter scrips that contains the commands for each component of the system, which increases the modularity. As a conclusion of this, a group of files is needed to generate the output image, instead of one single file used in ReconOS.

The workflow of the script is shown in figure 3.1, although it shows some steps to be independent of each other, the script is executed sequentially in the specified order.

The following subsections describe briefly the functionality of each of these parts in which the script is divided.

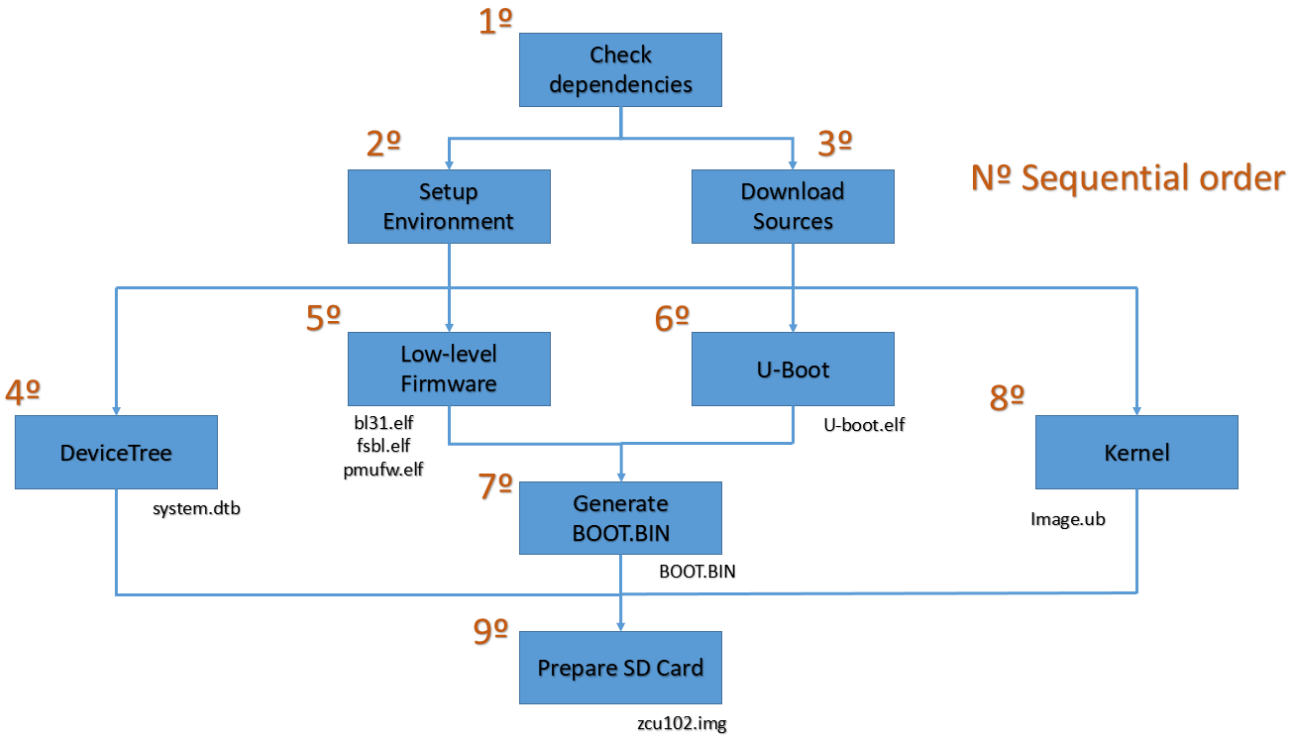


Figure 3.1.: Work flow of the custom `bash` script used to generate the SD card image

Step 1 - Check dependencies

In order to correctly generate the output image of the SD Card, the script executes some commands that might not be installed in the system. This step is responsible for ensuring this, checking the existence of the libraries files in the file system.

Step 2 - Setup environment

The script works in a cross-compilation environment because it builds executables for another machine with a different architecture. So the compilers need to know the target architecture and this step defines these environmental variables. In addition, is responsible for ask for the directory where put the downloaded sources and the generated files. The version of the Vivado Design Tools is also asked since is used in some future steps like the device tree generation. An important thing is that, although it asks for it, the script only works with the 2018.1 version of the tool. This is because the support for other versions is not discarded as a future line of work.

Step 3 - Download sources

As the name suggests, it only downloads the official repositories to work with. This step was kept from the ReconOS script, but it could be mixed with the other steps in future versions of the script.

Step 4 - Build devicetree

It creates the `.tcl` files to generate the Vivado project and build the *Device Tree Blob* (`.dtb` file) of the platform, which is placed in the boot partition of the SD Card. It also applies the modifications and bug fixes that are explained in the section 2.2.2 of the previous chapter related with the device tree.

Step 5 - Build low-level firmware

This step generates the following files related to the low-level firmware that is executed in the platform:

- `pmufw.elf`. The binary file that is placed in the PMU. It is responsible for numerous functionalities related to the management of the platform, such as power management and error handling at early stages.
- `bl31.elf`. Executable file for the Executable Level 3 Run time Trusted Firmware. Related with the Arm Trusted Firmware mandatory only in the Zynq UltraScale+ device.
- `fsbl.elf`. FSBL executable file.

Step 6 - Build U-Boot

It generates the executable file of the U-Boot, the universal SSBL that Xilinx recommends to use with its platforms. The elimination of all the bootargs definition must be highlighted since these arguments are defined by the device tree.

Step 7 - Generate BOOT.BIN

The BOOT.BIN is another of the required files placed in the Boot partition of the SD Card. This file contains the executables to be placed at the startup of the platform. It is generated with the `bootgen` command through a *Boot Image Format* `boot.bif` file, which is a human-readable file that contains the information about the destination of each executable.

Step 8 - Build kernel image

The objective of this step is to build the file `image.ub`, which is the compressed image of the Linux kernel. To do that, first it applies a default configuration defined for the ZCU102 platform, and next applies a patch to modify the required features before building it.

Step 9 - Prepare the SD Card

This is the most custom part of the script, and its output is the `ZCU102.img` file ready to be mounted in an SD Card and deployed in the platform.

First, it creates the image file and connects it with a "loop device", which is a pseudo-device that makes the file accessible as a block device. This allows its modification as if it was a disk device. Next, it configures the two partitions and its types. Once this configuration is finished, it copies the required files in the boot partition. Once this first partition is complete, it downloads the Linaro filesystem and adds some files related with the MALI Video drivers, which are downloaded and prepared to be compiled with some patches taken from the PetaLinux sources.

Finally, it generates a `bash` script that must be executed once the system is booted on the platform. It makes the steps in order to finally install the video drivers and the desktop environment.

3.2 DOOM execution and profiling

Once the system is completely functional and fulfils the project main goals of having a package manager and a desktop environment, the execution and profiling of

the DOOM game is done. This section explains the steps followed in order to achieve this, first in a general purpose computer, and then in the ZCU102 platform.

As it is said in the section 2.3 of the previous chapter, the Crispy-DOOM source port is used. The version that is selected is not the most recent one, because it is preferred to use the most similar version to the original Vanilla DOOM. Thus, the first version that breaks the FPS limit of 30 is used, because it restricts the acceleration of the game. In fact, due to this similarity, the final function that is accelerated also exists in the source code of the original game.

3.2.1 Steps to execute the game

The steps that must be done in order to execute the game are the same in both systems, but in the case of make the profiling in the platform, it is necessary to add an extra `CFLAG` to the makefiles due to the default configuration of the `GCC` compiler in the platform. These steps are:

1. Download the source code of the Crispy-DOOM from its official repository, and the shareware version of the game content.

2. Install the dependencies. They are the same as de `chocolate-doom` source port, which is in the Debian official repositories, so we can ask the `apt-get` command to install them.

3. Generate the configuration files using the `autoreconf` command.

4. Generate the makefiles executing the configuration files generated above. If the profiling is going to be executed, first the `-pg` flag have to be added to to the `CFLAGS` environmental variable used by the `gcc` compiler. This will instrument the code so that `gprof`, which is a performance analysis tool for Unix applications, will report detailed information about the execution of the game. In addition, if it is going to be executed in the ZCU102 platform, the `-no-pie` flag must be added too. This is due to the default configuration of the platform `gcc` compiler, which generates `pie` shared object binaries, although they run as a normal executable would.

5. Compile the source code with the `make` command. This will generate the executable file of the game inside the `src/` directory.

6. Execute the game giving the `WAD` file path as an argument, which is the file that contains the shareware version of the game. It is recommended to run first the setup executable to select the resolution of the window and other optional features.

The following `bash` script performs the steps described above:

```
#!/bin/sh

# Install dependencies
sudo -H apt-get install build-essential automake
sudo -H apt-get build-dep chocolate-doom

# Clone repo
git clone https://github.com/fabiangreffrath/crispy-doom.git
cd crispy-doom

# Checkout version
git checkout -b wb crispy-doom-3.0

# Generate configuration files
autoreconf -fiv

# Export environmental variables
export CFLAGS='-pg -no-pie'

# Generate makefiles
./configure

# Compile with maximum cores
make -j$(nproc)

# Downloads the .WAD and executes the game (ID Doom archive)
wget http://50.38.134.5/be_wads/doom1.wad
./src/crispy-doom-setup -iwad src/doom1.wad
```

3.2.2 Profiling results

Once the game is executed and closed correctly, a new file called `gmon.out` is generated in the directory of the executable. This file is created during the execution of the game and includes information about the functions that of the game has been

executed. This file could be read by the `gprof` tool which will display a flat profile with the following columns:

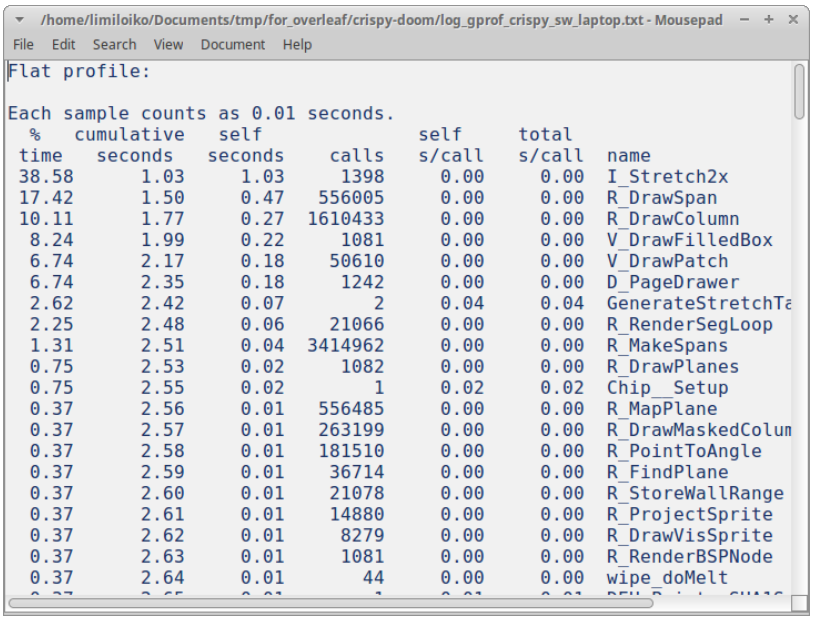
- `% time`. The percentage of the total running time of the program used by this function.
- `cumulative seconds`. A running sum of the number of seconds accounted for by this function and those listed above it.
- `calls`. The number of times this function was invoked.
- `self seconds`. The number of seconds accounted for by this function alone. This the major sort of the graph.
- `self ms/call`. The average number of milliseconds spent in this function per call.
- `total ms/call`. The average number of milliseconds spent in this function and its descendants per call.
- `name`. The name of the function.

The first lines of the resulting flat profiles of the Crispy-DOOM executed in a general purpose computer and in the ZCU102 platform are shown in figure 3.2.

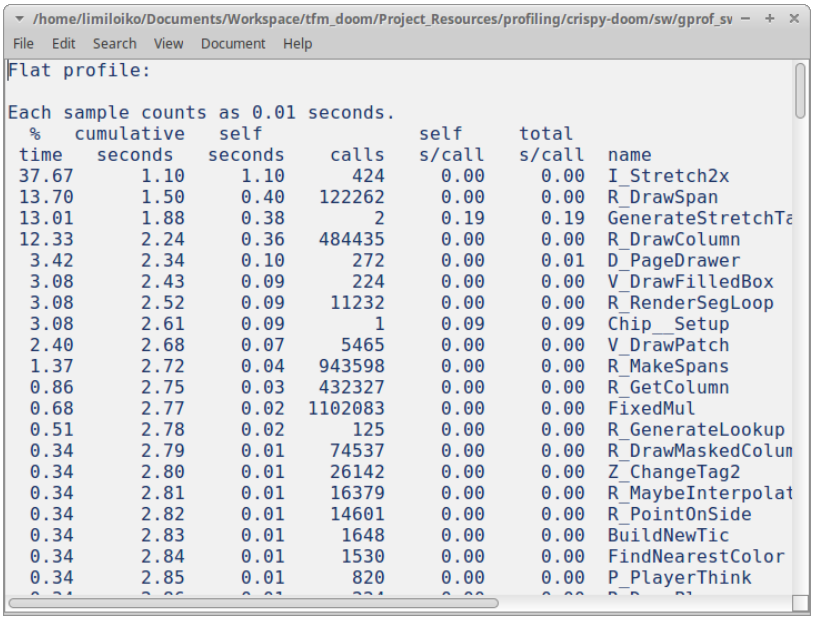
The result of the profiling is clear, the `I_Stretch2x` function is the one that keeps the CPU occupied the longest. Due to this, the work is focused on the creation of a hardware accelerator to reduce its impact in the CPU usage. The next section explains the methodology followed in order to implement this function in the PL of the platform.

3.3 Optimisation methodology

The original resolution of the Vanilla DOOM is $320x200$. Nowadays, the screens can reach bigger window sizes, as for example the $1280x960$ resolution that can be used in the ZCU102 platform. Due to this, the CPU needs to scale the resolution to the used one. Although modern CPUs can execute this very fast, it is what consumes the most time. One of the developers of Chocolate-DOOM corroborates this in an issue post in GiHhub [33].



(a) General purpose computer



(b) ZCU102 platform

Figure 3.2.: Profiling results

This section describes the methodology followed in order to accelerate this scale function, whose name is `I_Stretch2x`. It starts with the understanding of the function and continues with the explanation of the different steps that have been followed in

order to implement it in hardware. The different versions of the C function code can be found in a public BitBucket repository [34].

Although it is known that Vanilla DOOM has the resolution of $320x200$ [35], Crispy-DOOM uses the basic resolution of $640x400$. The maximum resolution that can be achieved in the platform is $1280x960$, so that is the reason of the function to be named "stretch2x", because it doubles the width and multiplies by 2,4 the height (stretch) of the image.

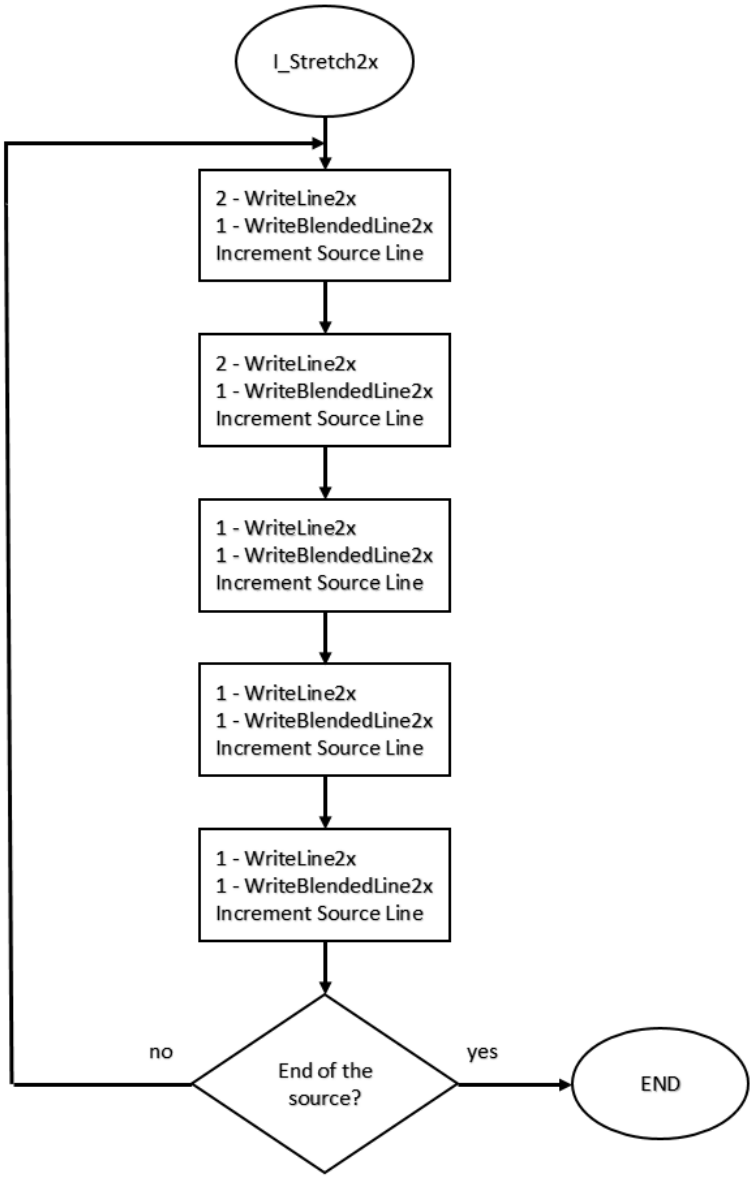


Figure 3.3.: Flowchart of the `Stretch2x` function

3.3.1 The `I_Stretch2x` function

The C code of the original function uses two buffers to read and write the pixel values, and its flowchart is shown in figure 3.3. If it is analysed, two main sub-functions that are called repeatedly are found:

- `writeLine2x`. Each time this function is called, a line of double width is written in the destination buffer. To do so, it writes each read pixel value two times in the destination buffer, until the line is finished.
- `writeBlendedLine2x`. It uses an external palette of colours to make a mixture in between the two lines. This is used to write a transition line.

Counting the number of lines written (sub-function calls), and the number of lines read (source line increments), it is said the function "writes 12 lines in the destination buffer for every 5 lines in the source buffer" and this is also included in one comment in the code. So the function is based on the execution of the main loop, which is executed until all the input frame is read. Although, if the number single write and read pixel operations are counted, only the third part of the input frame is read, thus only the third part of the destination is written. This fact is also noticed when the function is isolated and the input and output frames are saved as images, which are explained in the next subsections.

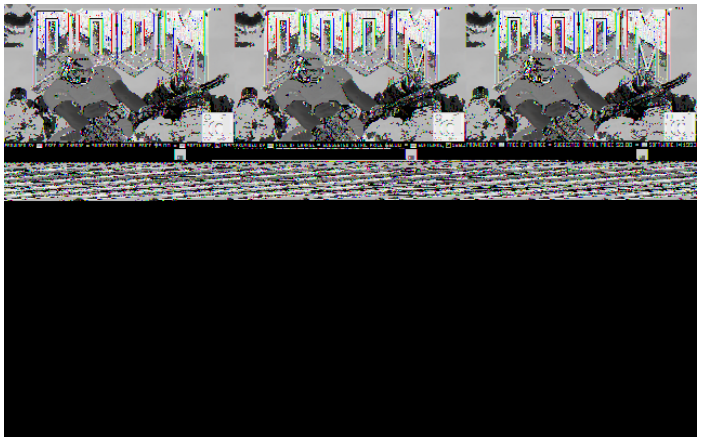


Figure 3.4.: Input frame $640x400$

Figure 3.5.: Output frame $1280x960$

3.3.2 Software isolation

To check the functionality of the `I_Stretch2x` function, a simple `cmake` project is created to apply the function to an image. The code of this project is available in a public repository in BitBucket [36] and it has the main goal of providing more information about the way of working of the software function.

First, the source code of the Crispy-DOOM is modified to save the input and output frames as images (figures 3.4 and 3.5). The analysis of the images is clear, the function changes the size of the image, the width is transformed to be 1280 (duplicated), and the height changes from 400 to 960 (multiplied by 2,4, stretched). It is also appreciable that only the third part of the image is scaled since the input frame shows noise that is not included in the output frame.

When a function is implemented in hardware, it must be aware of the global variables that it uses. Due to this, now that the aspect of the output is known, the

Figure 3.6.: Isolated output frame $1280x960$ of simpler implementation

consequences of substitute the `WriteBlendedLine2x` function by the `writeLine2x` are studied. It is known that this function is responsible of making the transition between lines smoothly, but it uses global variables that need to be handled by the communication interface, so the complexity of the hardware integration increases with its use.

The output of this substitution seems to have almost the same aspect (figure 3.6). To make sure of it, the same substitutions are made in the Crispy-DOOM source code, the results are the same, although the output frame has different values.

3.3.3 High Level Synthesis

Now that the function is isolated, the hardware implementation is started. To do this, Vivado®HLS [37] is used, which is a High-Level Synthesis tool capable to generate the hardware implementation of an algorithm described in C/C++. This tool

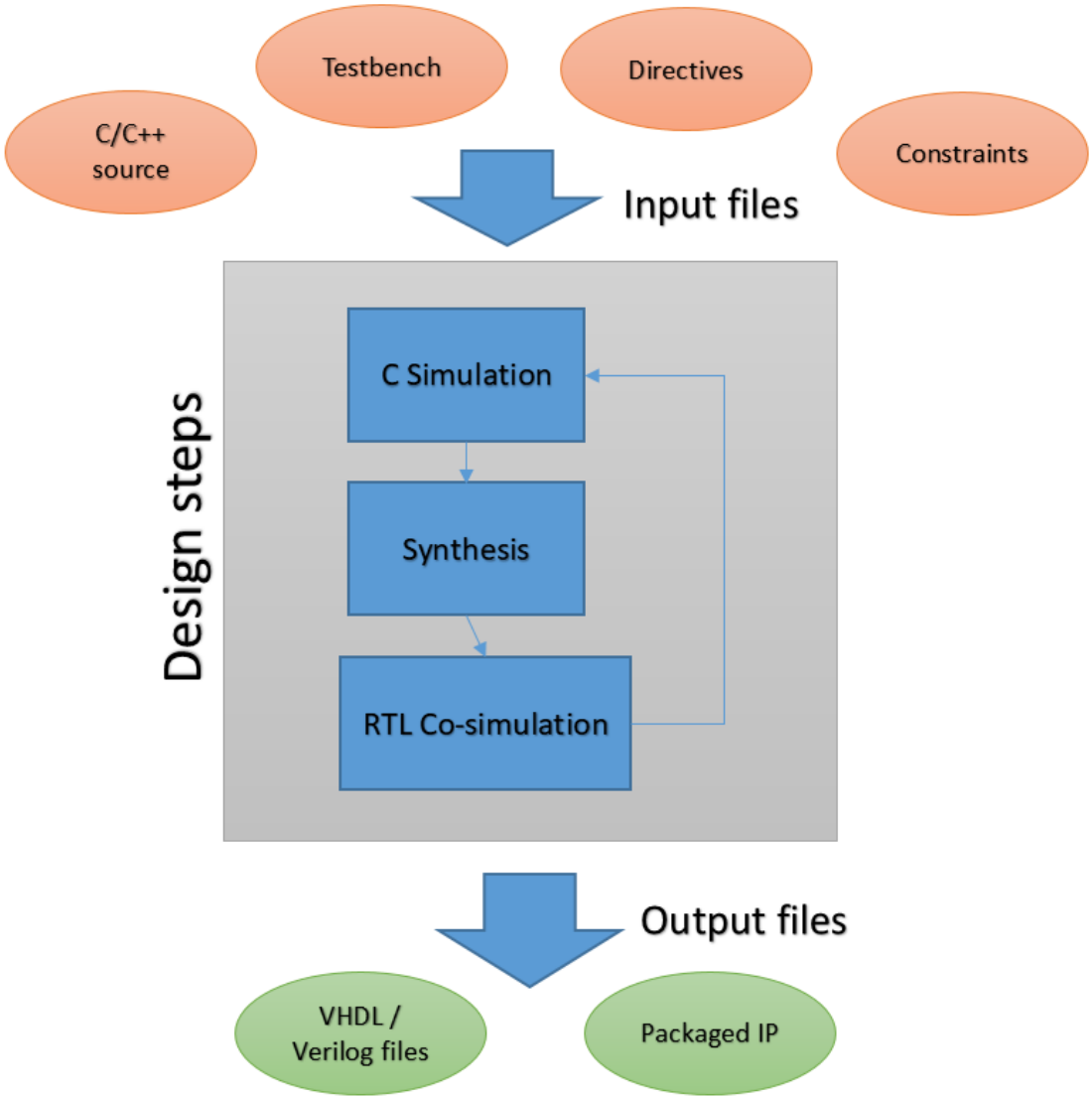


Figure 3.7.: Simplified HLS design process

also includes facilities of packaging the hardware into an IP block so it can be used with Vivado Design Suite.

The simplified design process used in order to implement a function in hardware is represented in figure 3.7. These steps take as input the following files:

- **C/C++ files**. They contain the function to be synthesised.
- **C testbench files**. To make the verification both the C code and the *Register-Transfer Level* (RTL) code generated by the synthesizer.

- **Constraints files**. The designer can supply timing constraints along with the clock uncertainty figure and details of the target device.
- **Directives**. The designer can apply some directives to influence the generated implementation from the high-level description, such as the pipelining level and the parallelism of the code.

In the case of the output files, the designer is able to choose which type is generated, and these are:

- **VHDL or Verilog files**. The tool generates the RTL-Level output depending on the language targeted by the user. This code is used to generate a bitstream for programming the FPGA.
- **Packaged IP for Vivado**. These outputs can be included in an IP Integrator project like Vivado. This is the output that it is used in order to create a bare-metal application to check the implementation of the hardware, which is the next step in the methodology.

In the design process showed in the figure, it can be identified the following design steps, which are usually executed cyclically until getting the desired output requirements.

- **C Simulation**. Using the testbench files included in the project, the objective is to check the correct functionality of the algorithm. In our case, the two versions of the implemented function are executed, and their results are compared.
- **Synthesis**. Once the functionality is checked, the directives must be included in order to optimise the hardware implementation. When this step is finished, a report is opened with the resources used and the first timing expectations, such as the expected latency of the module. This step is usually repeated to study the effect of applying different directives.
- **RTL co-simulation**. This is the final step and checks the correct functionality of the generated RTL code. It uses the same testbench as in the first step and gives a more accurate report of the timing and resources expectations.

These three design steps are usually executed in a cycle until getting the desirable implementation. Especially, in the case of changing the directives in the synthesis. This process of hardware optimisation could be hard, because it increases its difficulty with the module complexity, and it depends a lot of the experience of the designer. Thus, there is an order that it is advised to follow, which is:

1. **Datapath optimisations**. This affects the concurrency of the code in terms of the execution of instructions. An example is the "loop unrolling", which splits the loop iterations in order to execute them in parallel.

2. **Memory access optimisations**. The data access is usually a bottleneck in hardware implementations. When the parallelism of the code is increased, the concurrency of the data has also to be increased in order to be accessible in parallel. An example of these directives is the "array partitioning" of an array type argument, which splits the data into different memory blocks.

3. **Source code reshaping**. This step implies the change of the high-level description of the implemented algorithm. When the use of the last optimisation techniques is not enough to achieve the desired implementation, this step should be considered as a re-start process to change the code in order to make it more optimizable.

While these optimisations are applied, the interface definitions of the hardware module must be taken into account too. This part is really important because it will define the communication method between the PS and the PL hardware module. So for example, it is useless to have divided the input arguments into different memories, if we are using a streaming based interface since the data is arriving one by one. The Ultrascale+ platform is fully optimised to use AXI4 interfaces [38], so the main choices are:

- AXI4-Full protocol. For high-performance, memory-mapped requirements.
- AXI4-Lite protocol. For simple, low-throughput memory-mapped communication. For example, to control the status registers of the hardware module.
- AXI4-Stream protocol. For high-speed streaming data.

In our case, to make the optimisations in the hardware module, the reshaping of the code is done first. The next subsections describe these changes and the directives implementations that are made to the software function in order to obtain better optimisations results.

Hardware function reshape

The nature of the application gave some ideas on how to optimise it. It is known that function will be used continuously because is part of a video streaming, so it is intended to create a function able to give an output all the time.

When it is decided to implement it in hardware, the variables that it is using to perform its functionality must be taken into account. So, first, the use of each argument is studied. As a result, it is noticed that the function arguments are only related with the check of " full-screen update" function of the game, which the profiling shows that it is on most of the time. In addition, one important characteristic of the hardware is that it cannot use global variables. So the input and output buffers, which are global pointers, need to be implemented as arguments of the function.

The next list resumes the code changes that are made from the isolated function:

- The elimination of the `writeLine2x` calls. Now its code is included in the main function.
- The old arguments are deleted and also the instructions where the "'full screen update" function is checked.
- A new local variable is created to store the values of each line since each value is written two times before change the source line.
- Three new arguments are added, two of them to give the input and take the output frame. The third one is related to the parallelism strategy and is explained in the next subsection.

Hardware optimisations

At the moment at the code reshape is finished, the application of directives is started. The conclusion of this process is the application of the following directives:

- `HLS INTERFACE axis`. Both the interfaces of the input and output frames are defined as AXI Streaming.
- `HLS UNROLL`. The loops that perform the read operations of the source data are unrolled by a 2 factor in order to read two values at the same time.
- `HLS ARRAY_PARTITION`. The local variable that stores the input data is split into two blocks in order to allow the parallel read instructions above.

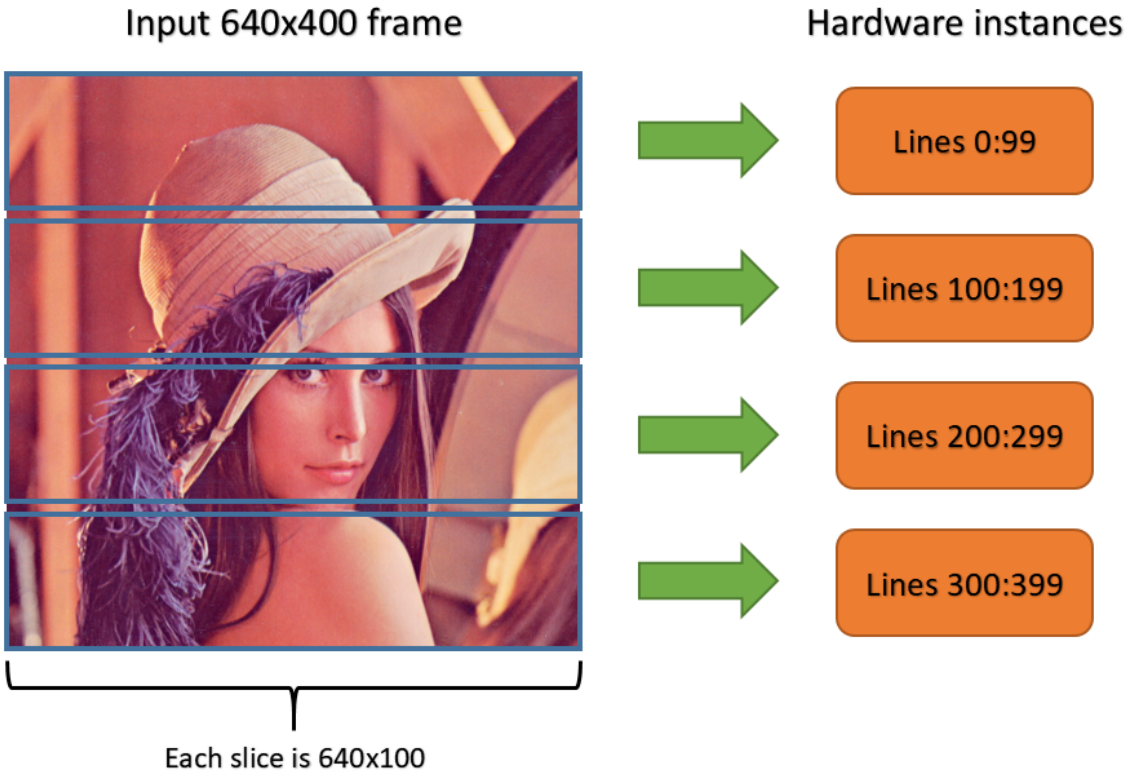


Figure 3.8.: Example of using 4 modules to split the input frame

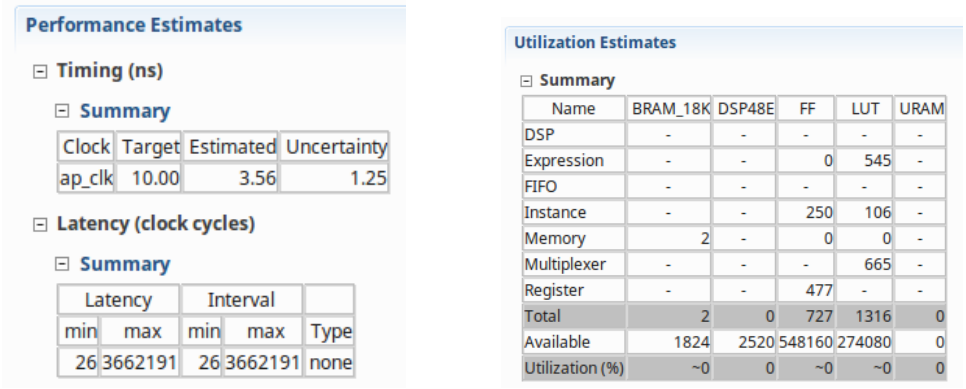
In addition to this hardware optimisations, the possibility of split the image into slides and perform the scale operation for each slide in parallel is taken into account. This strategy exploits The intrinsic level of parallelism of the image, as it is shown in figure 3.8, which is an example of using four modules. To implement this, a new function argument needs to be added in order to control the amount of data used by each module. This third argument is implemented through an AXI4-Lite interface, which also controls the status signals of the hardware.

The value of this third argument defines the number of iterations that the main loop performs. If the sentence "'for every 5 lines, 12 are written" is remembered, the conclusion that the input frame needs to have a height multiple by 5 is reached. Table 3.1 shows the effects of using a different number of modules in terms of main loop iterations and slices height.

Table 3.1.: Specifications with the number of instances

Nº instances	**Main loop iterations**	**Slice height (ppx)**
1	80	400
2	40	200
4	20	100
5	16	80
8	10	50

A direct limitation of this strategy is the use of the resources in the FPGA. But, as a limitation of the streaming based interface to use, input data cannot be accessed in parallel, since it brings data in a streaming connection with small packages. As a result, a module small enough is got that allows to have various instances of itself programmed at the same time. Figure 3.9 shows the resources used by one module. As is seen, the implementation of all modules can be done since fewer resources than the total are used.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	3.56	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
26	3662191	26	3662191	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	545	-
FIFO	-	-	-	-	-
Instance	-	-	250	106	-
Memory	2	-	0	0	-
Multiplexer	-	-	-	665	-
Register	-	-	477	-	-
Total	2	0	727	1316	0
Available	1824	2520	548160	274080	0
Utilization (%)	~0	0	~0	~0	0

(a) Performance estimation

(b) Resources estimation

Figure 3.9.: Hardware `stretch2x` function synthesis results

3.3.4 Bare metal application

Now that the hardware accelerator is designed, its work implemented in the ZCU102 platform must be checked. To do so, a Vivado Design Suite project is created using the module as a packaged IP. Figure 3.10 shows a screen-shot of its block design, in which are identified:

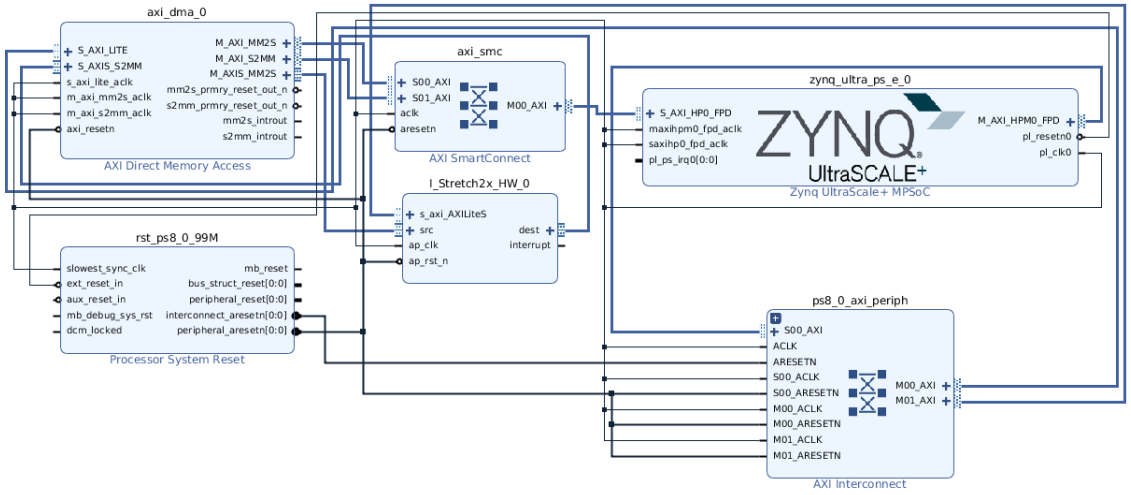


Figure 3.10.: Block design of Vivado bare-metal project

- The required Zynq-UltraScale PS block, which includes the platform configuration.
- The HLS packaged IP with the name of `I_Stretch2x_HW_0` imported from the Vivado HLS tool, which includes the designed hardware module. Noticed that there is only one HLS block because in this bare metal application only one module is used.
- The AXI DMA block that will be used to give and take the data from the hardware module.
- The additional AXI4 interface blocks, which control the configuration registers of the module, such as the signals related to its state, and the function argument that defines the number of instances.

Once the block design of the bare-metal application is defined, the code that will be executed in the PS must be designed. To do this, the Vivado SDK is used to

create an application that will do almost the same as the C Simulation of the Vivado HLS project, which executes both software and hardware functions and checks their results. The difference here is that the code uses the pre-defined functions to control the AXI DMA interface.

With this project, the correct functionality of the hardware module is checked on the device with a standalone OS, and not in a Linux-based system. This is due to the fact that the SDK is in charge of using the JTAG connection to program the PL and the PS.

3.3.5 Linux application

Giving the fact that the module is correct and can be placed in the PL of the device, we need to check how it is executed in a Linux-based system. To accomplish this, the SDSoC tools provided by Xilinx are used [39].

The SDSoC environment provides a set of tools to create a Linux-based application. It provides the already pre-built files that must be included in the SD Card to deploy the embedded Linux system on the platform. Due to this, the system is really limited to execute the application that is designed. In spite of this, the required features are included in our custom script in order to be able to execute the applications generated with this tool. So in fact, the bitstream and executable files generated by the SDSoC are only used, because the already generated boot files are used.

The great advantage of using this tool is that it is aware of the hardware interface with the module. This is, using directives defined by the user, it creates the additionally required blocks to handle the hardware. In this way, it takes care of the DMA drivers that are used to communicate with the module. This functionality is based on the substitution of the user-defined function that is wanted to be implemented in hardware, by a *stub* function that handles it. So the user only has to design the application and selects which function is liked to be implemented in hardware. The only restriction is that the source of this code should not have directives to define the interfaces. Instead of this, it has its own directives that define the interface method. In our case, the next ones have been used to define an AXI4-Streaming interface [40]:

- `#pragma SDS data access_pattern`. With the value of *sequential*, it defines the hardware interface as streaming.

- `#pragma SDS data copy`. Implies that data is explicitly copied between the host processor memory and the hardware function, instead of being accessed via shared memory.

This environment is used to create an executable which makes the same as the C Simulation step in the Vivado HLS project and in the bare-metal application, which is the call of both the isolated software function and the hardware implementation and compare their results. After that, a set of executables which use different hardware clock frequencies and a different number of instances are generated. With this, the different speedups that can achieve the hardware are taken to create a design space for our application. The results of these tests are presented in the next chapter of this document.

One important thing that must be known is that the code used to generate the different instances of the module is the same. This is, directives are used to tell the SDSoC environment that we want to use a specific number of modules. In addition, the tool uses a template to create the Vivado project where it places the HLS IP modules. And that is the reason why we have used 8 modules as the maximum because this template can not use more than 8 peripherals connected through DMA Blocks.

The source code of this SDSoC project is available as a public repository on BitBucket [41].

3.3.6 Crispy-DOOM integration

Now that the hardware function is checked in a Linux environment, the next step is to implement it in the Crispy-DOOM source code. To do this, another SDSoC utility is used, which is the creation of a C shared library that includes the software stub functions. Thanks to this, other applications can be compiled using these hardware modules. Of course, while the bitstream is loaded in the FPGA.

In order to compile the Crispy-DOOM with the hardware functions, the following files must be added to the game source repository:

- The shared library file, which includes the stub function and its dependencies, such as the information of the hardware addresses.
- A header file (`.h` file), which includes function prototypes. An `#include` statement must be added in the file where the function is called.

- The bitstream with the hardware, in order to be able to load it before executing the game.

With these files placed in the repository, we can create a similar `bash` script as the previous showed in the section 3.2.1 to prepare the environment to build the game:

```
#!/bin/sh

# To select PL frequency
# Possible values: 100, 150, 200, 300
MHZ=100

# To copy and load bitstream
cp bitstreams/stretch2x_$MHZ.bin /lib/firmware
echo stretch2x_$MHZ.bin > /sys/class/fpgma_manager/fpga0/firmware

# To set environmental variables
# Shared library
export LD_LIBRARY_PATH=LD_LIBRARY_PATH:$PWD/src/hadrware_library
export LDFLAGS='-L/home/documents/crispy/src/hardware_library/'
export LIBS="-lpthread -l${MHZ}_stretch2x_library"
# Include file
export CPPFLAGS='-I/home/documents/crispy/src/hardware_include/'
# gprof tool requirements
export CFLAGS='-pg -no-pie'

# To generate makefiles
./configure

# To compile with maximum cores
make -j$(nproc)

# To download the .WAD and execute the game
wget http://50.38.134.5/be_wads/doom1.wad
./src/crispy-doom-setup -iwad src/doom1.wad
```

When the previous script loads the bitstream, it includes the eight hardware modules. But the use of them is defined in the source code of the game. This is, all are loaded, in spite of the game not use all. This allows changing the number of used modules at run time, although it is not implemented anything that changes this number without recompiling the source.

To end this integration, a new feature has been added to the game source code, which is the measure of the FPS rate. The source code works like a big loop that

performs a lot of instructions, included the optimised function. What it is done, is the time measurement between the calls of the hardware function. As a result, the game prints the FPS rate for every 100 frames through the serial connection to the host PC.

In the next chapter, the results of measuring the FPS rate for different hardware configurations are included and described.

4. PROJECT RESULTS

The objective of this chapter is to describe and present the results of the different measurements that have been done. With these tests, a design space is explored to compare the different hardware setups and their characteristics in terms of cost-efficiency.

The results of implementing the hardware function in the Crispy-DOOM game are also presented. This measure is in terms of FPS rate, which is related to the speed of the game to perform its tasks.

In the first section, the design space and its components are described. Next, the tests and measurements are presented with their results. Finally, the implementation in the Crispy-DOOM game is shown.

4.1 Design Space

Design Space Exploration (DSE) refers to the activity of exploring the design alternatives prior to implementation. This task is very useful in a lot of engineering environments to study the different trade-offs that may exist and find the best compromise between them. In fact, in the world of the Embedded Systems, this task is getting more important with the increase of the technology complexity and the different choices that it has [42].

In the case of the implementation of our hardware function, the design space is limited to take into account as inputs:

- Number of hardware modules. Increasing it results in the acceleration of the hardware function since the image intrinsic level of parallelism is exploited. So each module has to process less data, and they execute in parallel. The available options are 1, 2, 4, 5 or 8 modules, since the SDSoC limits at 8 the number of instances.

- Hardware frequency. It refers to the hardware resources used in the implementation since the FPGA can use different clock frequencies. All the possible options that are able to use are checked, which are 100, 150, 200 and 300 MHz.

The combination of the previous variables conforms different hardware setups, whose cost-efficiency characteristics have been measured. In particular, the following characteristics have been considered:

- Speed-up. It is defined as the relation between the execution time of the software and hardware functions.
- Energy consumption. It has to do with the usage of platform resources, such as the CPU and FPGA.

The following section presents the tests performed to create this design space. The results obtained are also presented and described.

4.2 Measurements

The main goal of this section is to describe the two main tests that have been done in order to explore the design space. To do so, previously presented tools are used, such as the SDSoC (section 3.3.5) and Linux drivers that are included in the operating system.

The tests are focused on the different configurations that can be made and how they affect the acceleration of the execution. In addition, the energy consumed by the different parts of the platform is measured.

4.2.1 Hardware Speed-ups

The basic functionality of these tests lies in the comparison of the time it takes to execute both the software and hardware functions. In addition, since the executable provides the same input frame, the results are verified.

To generate the different executables and bitstream, the same project as in section 3.3.5 is used. This tool allows selecting the hardware frequency that will implement the modules. In addition, it can easily define the number that is wanted to use.

The main goal of this tests is to measure the time each of the software and hardware function takes to execute. Moreover, each function is executed a fixed number of times to increase the accuracy of the measure. This is because the CPU could be executing other minor tasks at the same time.

Table 4.1 shows the speed-up achieved in each setup, which use a different number of modules and clock frequencies. The calculations have been made 10, 100 and 1000 times.

Table 4.1.: Speed-ups using different number of modules, hardware frequencies and calculations

	100 MHz			**150 MHz**			**200 MHz**			**300 MHz**		
Nº	10	100	1000	10	100	1000	10	100	1000	10	100	1000
1	0.61	0.67	0.67	0.99	0.98	0.99	1.30	1.30	1.38	1.37	1.38	1.37
2	1.05	1.13	1.35	1.87	1.87	1.87	2.42	2.42	2.43	2.52	2.56	2.56
4	1.66	2.29	2.40	3.15	3.18	3.20	3.91	3.94	3.96	4.06	4.13	4.11
5	2.64	2.67	2.59	3.57	3.61	3.60	4.30	4.36	4.37	4.62	4.54	4.53
8	3.24	3.28	3.30	4.07	4.13	4.13	4.65	4.65	4.74	4.73	4.81	4.83

The following conclusions can be extracted from these results:

- Increasing the number of modules causes a reduction in the execution time. As expected, since the level of parallelism is increased and the amount of data that each instance has to process is reduced. This behaviour is represented better in figure 4.1, where only the results repeating the calculations 1000 times are used.
- The moment when the calculation is executed is also very important, as shown by the different results obtained by repeating the calculation different number of times. This is due to the other tasks which are executed by the operating system at the same time.
- The relation that the speed-up has with the frequency is also appreciable since the hardware circuit performs its calculations faster.

The final conclusion of these tests is that the use of the maximum number of modules with the highest frequency gives the best speed-up. But this not means that it will be optimum configuration. In the next subsection, the results of measuring the energy consumption of each hardware setup are presented.

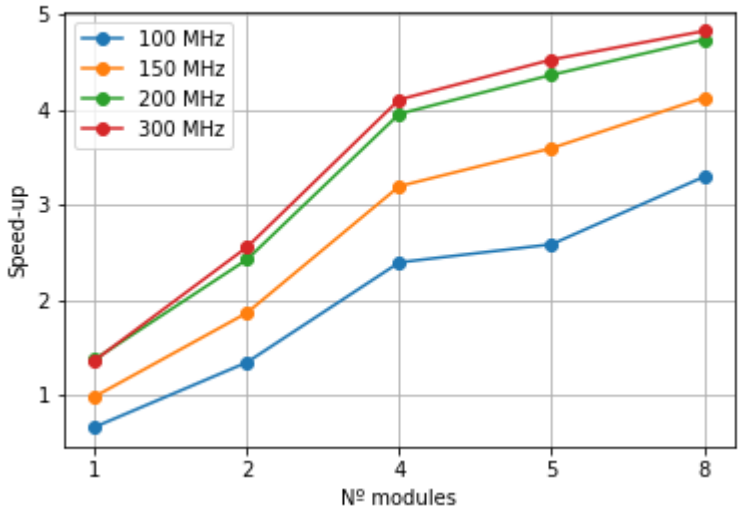


Figure 4.1.: Speed-ups using different hardware frequencies and number of modules

4.2.2 Energy consumption

In the area of embedded systems, it is very important to maintain a cost-efficiency balance. This is because there are numerous applications where optimal energy consumption is required. An example is the *Internet of Things* (IoT) field, where devices can be powered by batteries whose duration depends directly on consumption. Another very important area is aerospace, where there may be intermittent energy sources, such as solar energy. All this means that the applications must have an implementation with an optimal cost-efficiency balance.

To measure the power consumption, the same executables and bitstreams that check the sped-up are used. The values are read from an INA226 power monitor which is included in the ZCU102 platform. This device is connected to the MPSoC through an i2c interface, so the Linux i2c drivers can be used in order to read its values.

The important signals considered are those that represent the consumption of the Processing System (PS) and the Programmable Logic (PL). Their addresses are extracted from the ZCU102 user guide [43] and are represented in table 4.2 which also includes the `bash` command used to print the read value.

Table 4.2.: INA226 signal addresses

Signal	Address	Linux command
VCCINT (PL)	PL_PMBUS:0x40	`sensors ina226-i2c-4-40`
VCCPSINTFP (PS)	PS_PMBUS:0x40	`sensors ina226-i2c-3-40`

Figure 4.2 shows the power consumption of the PS according to the number of modules used. The case of executing the software function is represented with 0 modules.

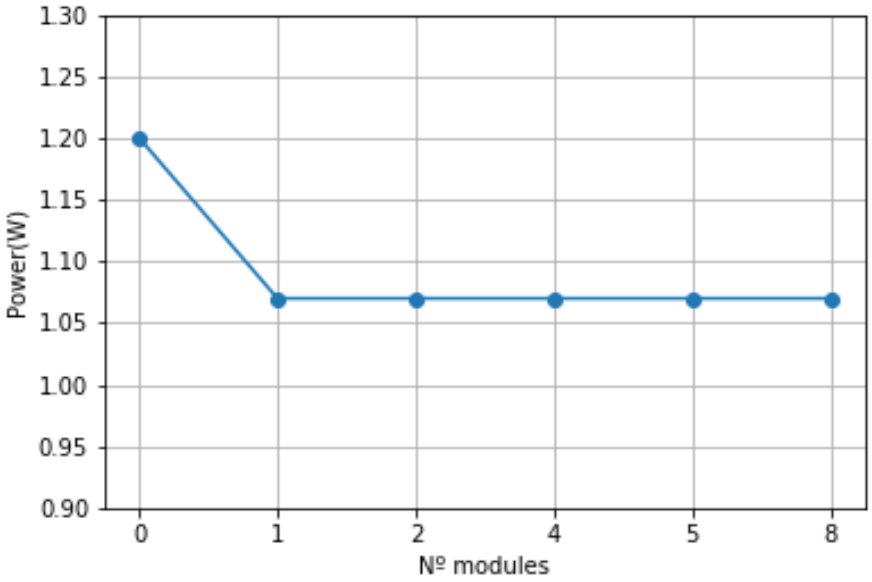


Figure 4.2.: Power consumption of the Processing System according to the number of modules used.

The main conclusion that can be extracted is that the PS power consumption is independent of the number of used modules. In addition, the static consumption of the PS is the same as when the hardware is used. Therefore, executing the hardware function does not produce any increase in the power consumption of the PS.

In the case of PL power consumption, there are more measurements, because the PL is directly affected by the loaded bitstream. In order to calculate the measure-

Table 4.3.: Programmable Logic power consumption

Nº	100 MHz	150 MHz	200 MHz	300 MHz
1	375 mW	400 mW	475 mW	500 mW
2	400 mW	500 mW	650 mW	700 mW
4	500 mW	575 mW	725 mW	975 mW
5	525 mW	750 mW	900 mW	1.15 W
8	650 mW	850 mW	1.02 W	1.43 W

ments, the values are read 1000 times. In table 4.3 these averaged results are shown as a function of the number of used modules and frequency.

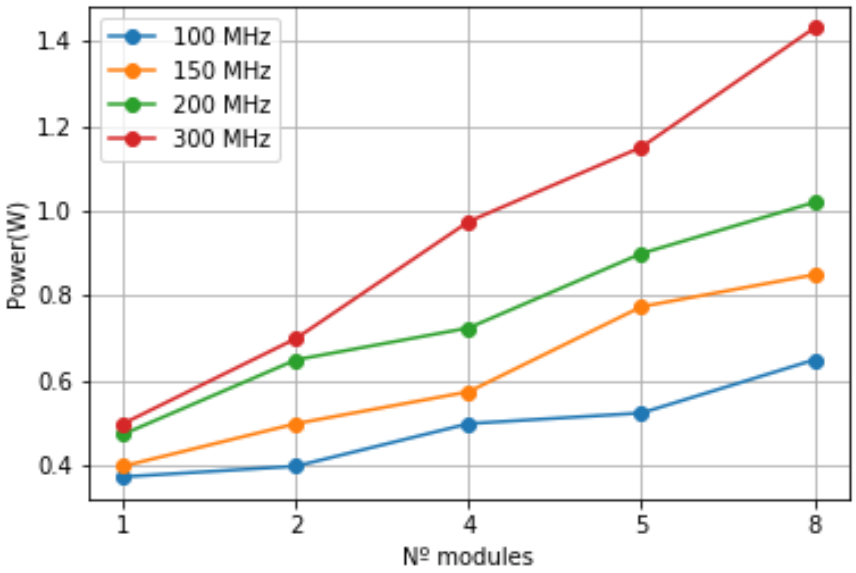


Figure 4.3.: Power consumption of the Programmable Logic according to the number of modules and frequency used

From these results, the following conclusions can be extracted:

- Increasing the number of modules produces an increment in the power consumption. The hardware circuit grows in area, thus more resources need to be supplied.

- Using higher frequency also increment the consumption. More calculations are performed increasing the frequency, so the signals are propagated faster.

Table 4.4.: Efficiency results of different hardware setups

Nº	**100 MHz**	**150 MHz**	**200 MHz**	**300 MHz**
1	1.787	2.475	2.905	2.740
2	3.375	3.740	3.738	3.657
4	4.800	**5.565**	5.46	4.215
5	4.933	4.645	4.855	3.939
8	5.077	4.859	4.647	3.377

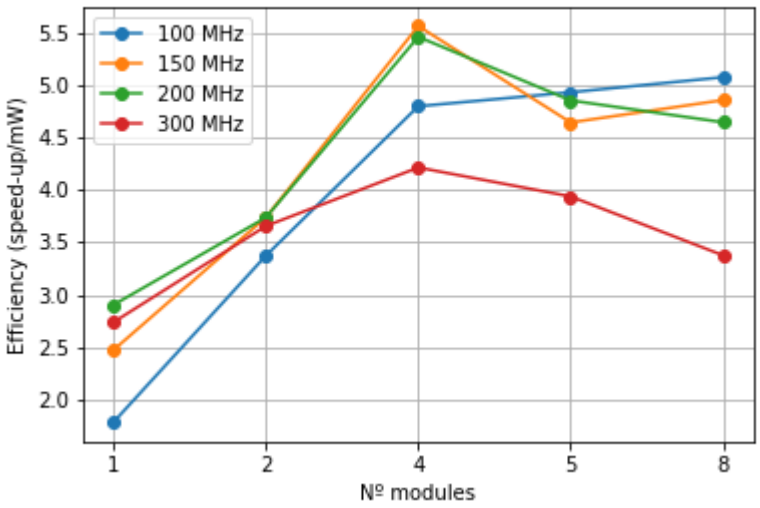


Figure 4.4.: Efficiency curves of different hardware setups

To finish the design space exploration, the efficiency values have been calculated with the division of each speed-up by the related PL power consumption. There results are included in table 4.4 and showed in figure 4.4. Thanks to these results, the following ideas are extracted:

- The cost of accelerating using 300 MHz is higher compared to the other frequencies in all cases.

- There is an efficiency peak in all cases where 4 modules are used.
- The most efficient setup in terms of acceleration per mW is the 150 MHz implementation. In particular, the case of 4 modules is the most efficient of all the design space.

The next section presents the results obtained with the implementation of different hardware configurations or setups in the Crispy-DOOM game.

4.3 In-game FPS

The same functions that are used in the speed-up tests have been implemented in the Crispy-DOOM game. To do so, the already generated bitstreams can be used. But in the case of the stub functions, the SDSoC environment is used to generate the shared libraries that contain the function prototypes.

To make the measure of the FPS rate, two new functions are added to the Crispy-DOOM source code. One to get the date time of the CPU in nanoseconds, and the other to send the information through the serial connection to the host PC. The first function is called each time a frame is calculated, and the second when 100 frames have been calculated. In this moment, the average FPS of those 100 measurements is packaged and sent as a kernel warning message. Due to the fact that the host PC is connected through a serial connection with the platform, it can read all the kernel messages, so the warning message sent from the game is successfully read.

Table 4.5.: FPS measured in the Crispy-DOOM game using different number of modules and hardware frequencies

Nº	**100 MHz**	**150 MHz**	**200 MHz**	**300 MHz**
0	55	55	55	55
1	28	33	38	39
2	39	44	47	49
4	46	51	53	53
5	49	53	55	56
8	54	55	56	56

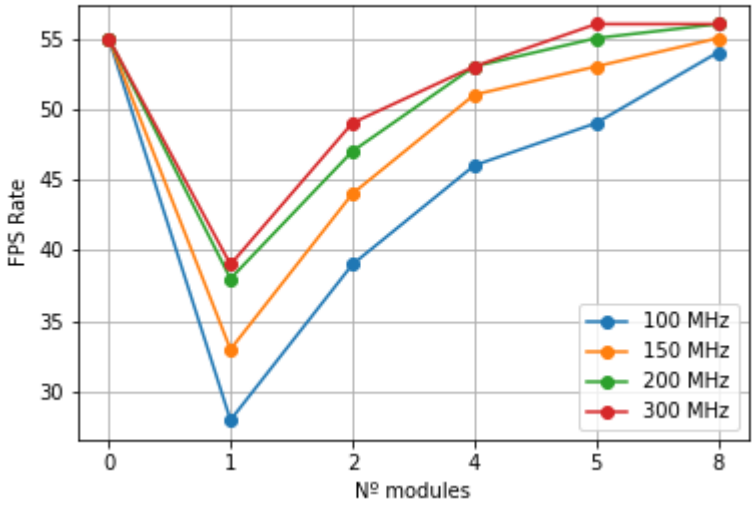


Figure 4.5.: Measured FPS rate using different hardware setups

The data that is taken as valid is the one referred to the time it takes to execute the second 100 frames. This is because is known that before the player starts a new game, the loop performed by the code is smaller.

Table 4.5 and figure 4.5 show the results of the implementation using different setups. The case of the software function is referred to as the case with 0 modules. The main conclusions that can be extracted from them are:

- All setups start bad and never improve the software implementation.
- The increments of the number of modules increases the FPS rate. The same happens with the frequency. This is the same behaviour as the speed-up results show.

The final conclusion of these results is that the game must have a new limit related to the maximum speed at which the game can be executed. In spite of this, the result of generating the new profile of the game (figure 4.6) shows that it has been possible to replace the software function with the hardware implemented in the PL.

One thing that can be thought after extracting the results, is that the function that has been accelerated may have a dependency relationship with the other two functions that now appear in the first positions of profiling. But this was ruled out with the replacement of empty functions, resulting in a measurement of 60 FPS without any

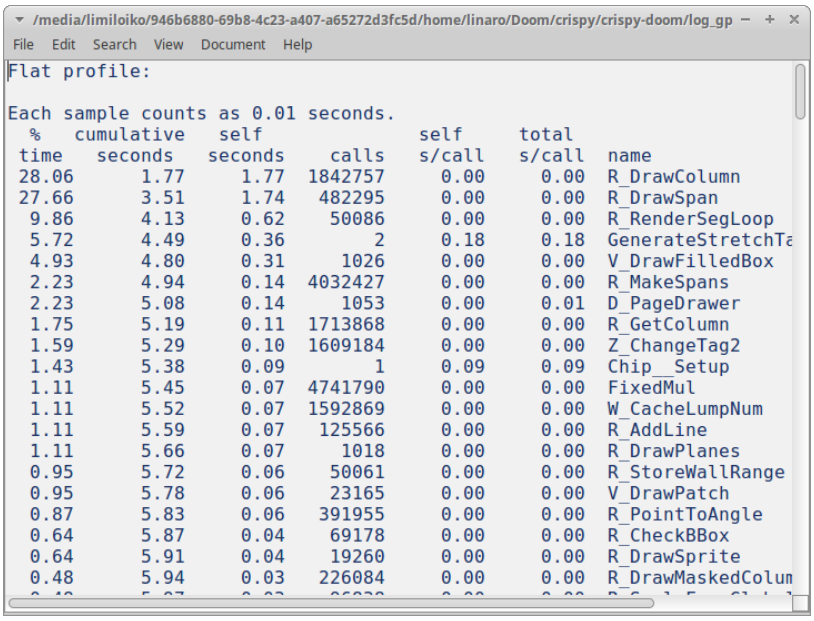


Figure 4.6.: `gprof` profiling results using the hardware modules

image output. Despite these bad results, the realization of other projects with the objective of making a deeper analysis of the game source code is encouraged.

5. CONCLUSIONS AND FUTURE WORK

The evolution of technology in the last decade has led to the appearance of new computational systems. These new devices are characterised by the integration of both software and hardware processing elements inside the same embedded system. Due to its intrinsic complexity, the design methodology of an application becomes difficult. The use of an embedded operating system is common in this new generation of electronics devices. This is due to the advantages derived from its presence, such as the abstraction level respect to the hardware.

The work done in this Master Thesis represents the methodology followed to design and implement a heterogeneous application in one of these new platforms. First, each of the elements that make up a Linux-based system has been generated and deployed. Next, the execution and profiling of the application have been made in order to identify the software function to implement in hardware. Finally, the exploration of the design space has been made in order to compare the different setups that can be used.

The process of exploring the design space has great importance in this type of heterogeneous devices since there are many options when designing the application. In our case, it could be assumed that the best setup to implement the hardware function would be the one that accelerated the most. But with the realization of the efficiency curves, it was not the most optimal choice in terms of power consumption.

The complexity of the system where the hardware function has been introduced has also been a key factor. Since a general acceleration of the game has not been achieved, despite the fact that the function that has more use of the CPU has been accelerated.

Future research projects

An important contribution of this project is the creation of a script that automates the process of creating a Linux-based system for the platform ZCU102. This script can serve as a starting point for future projects of heterogeneous acceleration, and can

also be improved to support new features. For example, the support for Artico^3 [44] has been added to use in reconfigurable research areas.

Future application projects

The process of making a deeper analysis and profiling of Crispy-DOOM game can be done in order to correct the saturation of the FPS rate. Furthermore, there are some other resources of the platform that can be used in order to increase the acceleration of the game, such as Mali GPU or the Dual-Core ARM R45.

APPENDICES

A. CUSTOM SCRIPT CODE

This appendix includes the source code of the custom script that is used to generate the SD Card image ready to be mounted. It is divided into sections for each of its parts, which are represented in figure 3.1 of chapter 3.

Although this is an important contribution to the project, it does not have a public repository yet, because some steps use non-public sources codes. An example of this is the part related with the `Artico`^{3} [44] that is developed in the university.

The following code is for the main script, which is responsible for sequentially calling the other steps.

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "desktop_image_zcu102.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Script to automatically build an SD Card image to boot a Debian 9
#        Stretch based system with a desktop-like environment in the Xilinx
#        zcu102 ultrascale+ platform.
#
#   It is recommended to run this script with the following command:
#
#   £ source doom.sh | tee log.txt
#
#   This will save all the output in a log file for future debug, if
#        needed.
#
# TODO:
#   - Complete the step 10: artico3 files.
```

```
#        - Add compatibility with other vivado versions.
#
# CHANGELOG:
#         [22-11-2018]: Init separated repository.
#
#-----------------------------------------------------------------------

# Git tags to get known stable versions
#
# NOTE: users can decide to comment the respective lines in the code to
#       get the latest versions, but these are provided as failsafe
#       alternatives.
GITTAG_LINUX="xilinx-v2018.1"
GITTAG_DTC="v1.4.7"

# Scripts directory
SCRIPTS="$PWD/scripts"

#
# [0] Parse command line arguments
#

if [ "$#" -ne 0 ]; then
    printf "Illegal number of arguments\n"
    printf "Usage: $0\n"
    exit 1
fi

#
# [1] Check dependencies
#
echo  "----------------------------------"
printf "Checking dependencies ...\n"
source $SCRIPTS/01_check_dependencies.sh
printf "\n"

#
# [2] Setup environment variables and working directory
#
echo "----------------------------------"
```

```
printf "Setting environment variables and working directory ...\n"
source $SCRIPTS/02_setup_environment.sh
printf "\n"

#
# [3] Download sources from Git repositories
#
echo "-----------------------------------"
printf "Downloading u-boot and linux-xlnx repositories ...\n"
source $SCRIPTS/03_download_sources.sh
printf "\n"

#
# [4] Build Device Tree
#
echo "-----------------------------------"
printf "Building Device Tree ...\n"
source $SCRIPTS/04_build_devicetree.sh
printf "\n"

#
# [5] Build low-level firmware
#
echo "-----------------------------------"
printf "Building low-level Firmware ...\n"
source $SCRIPTS/05_build_lowlevel_firmware.sh
printf "\n"

#
# [6] Build U-Boot
#
echo "-----------------------------------"
printf "Building U-Boot ...\n"
source $SCRIPTS/06_build_u_boot.sh
printf "\n"

#
# [7] Generate boot image
#
echo "-----------------------------------"
```

```
printf "Creating boot.bin ...\n"
source $SCRIPTS/07_generate_boot_image.sh
printf "\n"

#
# [8] Build Linux kernel
#
echo "-----------------------------------"
printf "Compiling the kernel ...\n"
source $SCRIPTS/08_build_kernel.sh
printf "\n"

#
# [9] Prepare SD card
#
echo "-----------------------------------"
printf "Preparing the SD Card image ...\n"
source $SCRIPTS/11_prepare_sd_card.sh
printf "\n"

#
# [10] Final steps
#
echo "-----------------------------------"
printf "ARTICo3/desktop SD card image has been generated successfully.\n\n"
printf "Please execute the following post-installation steps:\n"
printf "* SD card image : $WD/rootfs/desktop_image_zcu102.img\n\n"
printf "  > sudo -H dd if=$WD/rootfs/desktop_image_zcu102.img of=/dev/sdX \
        bs=4M status=progress\n\n"
printf "* Set boot mode to SD card.\n"
printf "* Turn on the board, connect with UART.\n"
printf "* If 'ZynqMP> ' prompt appears type 'boot' followed by 'Enter'.\n"
printf "The Linux prompt '#/ ' will appear.\n"
printf "Log in as Linaro\n\n"
printf "* >login linaro\n\n"
printf "Execute ./desktop_install.sh to make final steps.\n\n"
printf "* source desktop_install.sh\n\n"
printf "After reboot you should see the login screen.\n"
```

A.1 check_dependencies.sh

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/1_check_dependencies.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Check that all required libraries are available.
#
#
# TODO:
#
# CHANGELOG:
#
#-------------------------------------------------------------------------


# Functions to check availability of required applications and libraries
function pre_check {
    var=$3
    if command -v "$2" > /dev/null 2>&1; then
        printf "[OK]    $1\n"
    else
        printf "[ERROR] $1 (ubuntu pkg: $4)\n"
        eval $var=false
    fi
}

function pre_check_lib {
    var=$3
    ldconfig -p | grep "$2" > /dev/null 2>&1
    if [ $? -eq 0 ]; then
        printf "[OK]    $1\n"
    else
```

```
        printf "[ERROR] $1 (ubuntu pkg: $4)\n"
        eval $var=false
    fi
}

# General dependencies
pre_check "Git" "git" ALLCHECK "git"
pre_check "Sed" "sed" ALLCHECK ""
pre_check "Make" "make" ALLCHECK "build-essential"
pre_check "Diff" "diff" ALLCHECK "diff"
pre_check "Tar" "tar" ALLCHECK "tar"

# Device Tree Compiler (dtc) dependencies
pre_check "Flex" "flex" ALLCHECK "flex"
pre_check "Bison" "bison" ALLCHECK "bison"

if [ "$ALLCHECK" = false ]; then
    printf "Dependencies not met. Install missing packages and \
        try again.\n"
    exit 1
fi

# Exit if [ £? -ne 0 ] from now on (i.e. if any error occurs while
# executing one of the commands in the script.)
set -e
```

A.2 setup_environment.sh

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/2_setup_environment.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Set working paths and all required environmental variables, such as
#   ARCH and CROSS_COMPILE.
#
# TODO:
#
# CHANGELOG:
#   [22-10-2018]: PATCHES_DIR environment variable added.
#
#-------------------------------------------------------------------------

# Save patches directory
PATCHES_DIR="$PWD/patches"

# Xilinx tools installation directory
printf "Type yout Xilinx installation directory:\n"
printf "/opt is the default:"
while true; do
    read -e -p "> " -i "/opt" XILINX_ROOT
    if [[ $XILINX_ROOT = "" ]]; then
        continue
    fi
    if [[ ! -e "$XILINX_ROOT" ]]; then
        printf "$XILINX_ROOT was not found, please select an existing \
                directory.\n"
        continue
    else
        break
```

```
    fi
done

# Create working directory
printf "Type path for the working directory:\n"
while true; do
    read -e -p "> " WD
    if [[ $WD = "" ]]; then
        continue
   fi
    WD=${WD/#\~/$HOME} # make tilde work
    if [ -d "$WD" ]; then # true if dir exists
        if find "$WD" -mindepth 1 -print -quit | grep -q .; then
            printf "The directory $(pwd)/$WD is not empty, please choose \
                different name:\n"
        else # true if dir exists and emtpy
            break
        fi;
    else # true if dir does not exist
        mkdir -p $WD
        break
    fi
done
WD=$( cd $WD ; pwd -P ) # change to absolute path
cd $WD

# Get cross-compiler toolchain
printf "Type in your Xilinx Vivado tool version.\n"
printf "2018.1 is the only compatible\n"
while true; do
    read -e -p "> " -i "2018.1" XILINX_VERSION
    if [[ $XILINX_VERSION = "" ]]; then
        continue
    fi
    XILINX_VERSION=${XILINX_VERSION/#\~/$HOME} # make tilde work
    if [[ ! -e "$XILINX_ROOT/Xilinx/Vivado/$XILINX_VERSION" ]]; then
        printf "$XILINX_ROOT/Xilinx/Vivado/$XILINX_VERSION was not \
                found. Change XILINX_ROOT value to match your installation \
                directory and run script again.\n"
        exit 1
```

```
    else
        break
    fi
done

# Export environment variables
export ARCH="arm64"
export CROSS_COMPILE="$XILINX_ROOT/Xilinx/SDK/$XILINX_VERSION/gnu\
/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-"
export KDIR="$WD/linux-xlnx"
export PATH="$WD/devicetree/dtc:$PATH"
export PATH="$WD/u-boot-digilent/tools:$PATH"


# Source Xilinx tools script (add Xilinx tools to the path)
source "$XILINX_ROOT/Xilinx/Vivado/$XILINX_VERSION/settings64.sh"
```

A.3 download_sources.sh

```
#/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/3_download_sources.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Download u-boot and linux kernel repos
#
# TODO:
#
# CHANGELOG:
#
#-------------------------------------------------------------------------

# Download repos in the working directory
git clone https://github.com/Xilinx/u-boot-xlnx.git
git clone https://github.com/xilinx/linux-xlnx
```

A.4 build_devicetree.sh

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/4_build_devicetree.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Creates the vivado project and create the first system.dtb file,
#   compatible with device-tree overlays.
#
# CHANGELOG:
#  [03-10-2018] Update to Vivado v2018.1:
#    - Board file version changed to 3.2 (xilinx.com:zcu102:part0:3.2)
#    - Device 'xczu9eg-ffvb1156-2-i' changed to 'xczu9eg-ffvb1156-2-e'
#
#  [14-11-2018] SDSoC driver added.
#
#------------------------------------------------------------------------
# Create working directory for device tree development
mkdir -p $WD/devicetree

# Download latest Device Tree Compiler (dtc), required to work with
# device tree overlays, and build it.
cd $WD/devicetree
git clone https://git.kernel.org/pub/scm/utils/dtc/dtc.git
cd $WD/devicetree/dtc
git checkout -b wb "$GITTAG_DTC"
make -j"$(nproc)"

# Download Xilinx device tree repository
cd $WD/devicetree
git clone https://github.com/Xilinx/device-tree-xlnx
cd $WD/devicetree/device-tree-xlnx
```

```
git checkout -b wb $GITTAG_LINUX

# Create directory to build device tree for zcu102 board
mkdir $WD/devicetree/zcu102
cd $WD/devicetree/zcu102

# Create required files
cat > create_hdf.tcl << EOF
set cur_dir [pwd]

# Create Vivado project
create_project -force zcu102_dt \$cur_dir/zcu102_dt \
        -part xczu9eg-ffvb1156-2-e
set proj_name [current_project]
set proj_dir [get_property directory [current_project]]
set_property "target_language" "VHDL" \$proj_name
set_property board_part xilinx.com:zcu102:part0:3.2 [current_project]

# Create block diagram
create_bd_design "design_1"
update_compile_order -fileset sources_1

# Add processing system for Zynq US+ Board
create_bd_cell -type ip -vlnv xilinx.com:ip:zynq_ultra_ps_e:3.2 \
        zynq_ultra_ps_e_0
apply_bd_automation -rule xilinx.com:bd_rule:zynq_ultra_ps_e -config \
        {apply_board_preset "1"}  [get_bd_cells zynq_ultra_ps_e_0]

# Make sure required AXI ports are active
set_property -dict [list CONFIG.PSU__USE__M_AXI_GP0 {1}  \
                         CONFIG.PSU__USE__M_AXI_GP1 {1}  \
                         CONFIG.PSU__USE__M_AXI_GP2 {1}] \
                         [get_bd_cells zynq_ultra_ps_e_0]

# Add interrupt port
set_property -dict [list CONFIG.PSU__USE__IRQ0 {1}] \
        [get_bd_cells zynq_ultra_ps_e_0]

# Set Frequencies
set_property -dict [list CONFIG.PSU__FPGA_PL0_ENABLE {1} \
```

```
        CONFIG.PSU__CRL_APB__PL0_REF_CTRL__FREQMHZ {100}] \
        [get_bd_cells zynq_ultra_ps_e_0]

# Connect clocks
connect_bd_net [get_bd_pins zynq_ultra_ps_e_0/pl_clk0] \
               [get_bd_pins zynq_ultra_ps_e_0/maxihpm0_fpd_aclk] \
               [get_bd_pins zynq_ultra_ps_e_0/maxihpm1_fpd_aclk] \
               [get_bd_pins zynq_ultra_ps_e_0/maxihpm0_lpd_aclk]

# Update layout of block design
regenerate_bd_layout

# Create wrapper file
make_wrapper -files \
        [get_files \$proj_dir/\$proj_name.srcs/sources_1/bd/design_1/\
        design_1.bd] -top
add_files -norecurse \$proj_dir/\$proj_name.srcs/sources_1/bd/design_1/\
        hdl/design_1_wrapper.vhd
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
set_property top design_1_wrapper [current_fileset]
save_bd_design

# Generate output products
generate_target all [get_files  *.bd]

# Write hardware definition and close project
write_hwdef -force -file "\$cur_dir/zcu102.hdf"
close_project
EOF

cat > create_devicetree.tcl << EOF
open_hw_design zcu102.hdf
set_repo_path ../device-tree-xlnx
create_sw_design device-tree -os device_tree -proc psu_cortexa53_0
set_property CONFIG.periph_type_overrides "{BOARD zcu102-rev1.0}" [get_os]
generate_target -dir dts
EOF

# Generate device tree from a simple vivado project
```

```
vivado -mode batch -source create_hdf.tcl
hsi -mode batch -source create_devicetree.tcl

# Edit Device Tree
cd $WD/devicetree/zcu102/dts/

# Add interrupt controller to PL node
sed -i '/compatible = "simple-bus";/a \\t\tinterrupt-parent = <&gic>;'\
        pl.dtsi

# Enable DMA usage via DMA Proxy
sed -i '/fpd_dma_chan1/a \\t\t\t#dma-cells = <1>;'\
 zynqmp.dtsi

# Modify bootargs
BOOTARGS="bootargs = \"console=ttyPS0,115200 rootdelay=3 \
root=/dev/mmcblk0p2 rw earlyprintk uio_pdrv_genirq.of_id=generic-uio \
earlycon clk_ignore_unused\";"
sed -i "s|.*bootargs.*|\t\t$BOOTARGS|" system-top.dts

# Patch to correct dual lane DP
patch $WD/devicetree/device-tree-xlnx/device_tree/data/kernel_dtsi/\
2017.1/BOARD/zcu102-revb.dtsi $PATCHES_DIR/devicetree/\
0001-Fix-for-gtr_sel0-polarity-correct-for-dual-lane-DP.patch

# Add SDSoC driver
# Create the .dtsi file
cat > sdsoc-driver.dtsi << EOF
/{
 xlnk {
 compatible = "xlnx,xlnk-1.0";
 };
};
EOF

# Modify system-top.dts to include the .dtsi
sed -i '/"pl.dtsi"/a /include/ "sdsoc-driver.dtsi"' $WD/devicetree/\
zcu102/dts/system-top.dts

# Patch the pcw.dtsi file to correct the display-port
```

```
patch $WD/devicetree/zcu102/dts/pcw.dtsi $PATCHES_DIR/devicetree/\
0002-fix-display-port.patch

# Generate device tree blob (adding symbols, that is the reason for
# using the last version of dtc)
cd $WD/devicetree/zcu102/dts
dtc -I dts -O dtb -@ -o system.dtb system-top.dts
```

A.5 build_lowlevel_firmware.sh

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/5_build_lowlevel_firmware.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Generates the fsbl.elf, bl31.elf and the pmufw.elf files.
#
# TODO:
#
# CHANGELOG:
#   [22-10-2018]: xilfpga_pcap drivers changed:
#           - Added two patch files to bugfix a problem in the pmu, drivers
#             changed to v2.0.
#
#-------------------------------------------------------------------------

# Create directory to low-firmware development
mkdir -p $WD/firmware
cd $WD/firmware

#
# Build ARM Trusted Firmware
#
# NOTE: repo version xilinx-v2018.1 is used (it is compatible) because
#       it is the first release in which PL configuration from Linux is
#       done properly [1] (in previous releases, it was a non-blocking
#       process that led to runtime errors due to wrong PL programming
#            and race conditions).
#
# PL programming from Linux steps:
#
#   1. echo £flags > /sys/class/fpga_manager/fpga0/flags
```

```
#       echo $firmware > /sys/class/fpga_manager/fpga0/firmware
#
#       This procedure does not follow conventional FPGA Manager
#             framework, since it uses custom files named _flags_ and
#             _firmware_ instead of relying on Device Tree Overlays.
#       See [2] for more info.
#
#   2. Linux (xilinx-v2017.2) @ ARM Cortex A53
#       | fpga_mgr_firmware_load()  drivers/fpga/fpga-mgr.c
#       | fpga_mgr_buf_load()       drivers/fpga/fpga-mgr.c
#       | zynqmp_fpga_ops_write()   drivers/fpga/zynqmp-fpga.c
#       | zynqmp_pm_fpga_load()     drivers/soc/xilinx/zynqmp/pm.c
#       | invoke_pm_fn()            drivers/soc/xilinx/zynqmp/pm.c
#       | do_fw_call()              drivers/soc/xilinx/zynqmp/pm.c
#       | do_fw_call_smc()          drivers/soc/xilinx/zynqmp/pm.c
#       | arm_smccc_smc()           arch/arm64/kernel/smccc-call.S
#
#   3. ARM Trusted Firmware (xilinx-v2018.1) @ ARM Cortex A53
#       | pm_smc_handler()          plat/xilinx/zynqmp/pm_service/pm_svc_main.c
#       | pm_fpga_load()            plat/xilinx/zynqmp/pm_service/pm_api_sys.c
#       | pm_ipi_send_sync()        plat/xilinx/zynqmp/pm_service/pm_ipi.c
#       | pm_ipi_send_common()      plat/xilinx/zynqmp/pm_service/pm_ipi.c
#       | ipi_mb_notify()           plat/xilinx/zynqmp/pm_service/pm_ipi.c
#
#   4. Zynq MP PMU Firmware (xilinx-v2017.1) @ PMU MicroBlaze
#       | PmProcessApiCall ()
#                                   <embeddedsw>/lib/sw_apps/zynqmp_pmufw/src/pm_core.c
#
#       | PmFpgaLoad ()
#                                   <embeddedsw>/lib/sw_apps/zynqmp_pmufw/src/pm_core.c
#
#       | XFpga_PL_BitSream_Load()
#                                   <embeddedsw>/lib/sw_services/xilfpga/src/xilfpga_pcap.c
#
# [1] https://github.com/Xilinx/arm-trusted-firmware/commit/\
c055151bfd7641f9a748de2ecd50ff968ff07176#diff-84707f287ea5d2f11d613b22d81b5534
# [2] Documentation/devicetree/bindings/fpga/fpga-region.txt
#
git clone https://github.com/Xilinx/arm-trusted-firmware.git
cd $WD/firmware/arm-trusted-firmware
```

```
git checkout -b wb "$GITTAG_LINUX"

make -j $(nproc) PLAT=zynqmp RESET_TO_BL31=1

# Create ZynqMP FSBL + PMU Firmware projects
cd $WD/firmware
cp $WD/devicetree/zcu102/zcu102.hdf $WD/firmware
cat > create_firmware.tcl << EOF
set hwdsgn [open_hw_design zcu102.hdf]
generate_app -hw \$hwdsgn -os standalone -proc psu_cortexa53_0 \
        -app zynqmp_fsbl -sw fsbl -dir fsbl
generate_app -hw \$hwdsgn -os standalone -proc psu_pmu_0 -app \
zynqmp_pmufw -sw pmufw -dir pmufw
EOF
hsi -mode batch -source create_firmware.tcl

# Modify BSPs to support DPR
#
# Patch xilxfpga_pcap v4.0 to be compatible with v2.0.
patch $WD/firmware/pmufw/zynqmp_pmufw_bsp/psu_pmu_0/libsrc/xilfpga_*/\
        src/xilfpga_pcap.c \
        $PATCHES_DIR/pmu/0001-xilfpga-pcap-c_v4_0-to-be-compatible-with-v2_0.patch

patch $WD/firmware/pmufw/zynqmp_pmufw_bsp/psu_pmu_0/libsrc/xilfpga_*/\
src/xilfpga_pcap.h
        $PATCHES_DIR/pmu/0002-xilfpga-pcap-h_v4_0-to-be-compatible-with-v2_0.patch

# Build ZynqMP FSBL + PMU Firmware
cd $WD/firmware/fsbl
make
cd $WD/firmware/pmufw
make
```

A.6 build_u_boot.sh

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/6_build_u_boot.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Compiles and generates the u-boot.
#
# TODO:
#
# CHANGELOG:
#   [04-10-2018] Update to Vivado v2018.1:
#     - defconfig file changed to xilinx_zynqmp_zcu102_rev1_0_defconfig
#     - Default bootargs are deleted from defconfig file.
#
#------------------------------------------------------------------------

cd $WD/u-boot-xlnx
git checkout -b wb "$GITTAG_LINUX"

# Remove default bootargs (in some defconfigs, BOOTARGS is in
# the *_defconfig file)
sed -i '/.*BOOTARGS.*/d' $WD/u-boot-xlnx/configs/\
        xilinx_zynqmp_zcu102_rev1_0_defconfig

make -j $(nproc) xilinx_zynqmp_zcu102_rev1_0_defconfig ;
# xilinx_zynqmp_zcu102_rev1_0_defconfig for version >= xilinx-v2018.1

sed -i 's|"sdboot=.*|"sdboot=mmc dev \$sdbootdev \&\& mmcinfo \&\& \
run uenvboot;" \\|' $WD/u-boot-xlnx/include/configs/xilinx_zynqmp.h

cd $WD/u-boot-xlnx
make -j $(nproc)
```

A.7 generate_boot_image.sh

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/7_generate_boot_images.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Generates the boot.bin file.
#
#-------------------------------------------------------------------------

mkdir -p $WD/boot
cd $WD/boot

# Copy files
cp $WD/firmware/fsbl/executable.elf fsbl.elf
cp $WD/firmware/pmufw/executable.elf pmufw.elf
cp $WD/firmware/arm-trusted-firmware/build/zynqmp/release/bl31/bl31.elf\
 bl31.elf
cp $WD/u-boot-xlnx/u-boot.elf u-boot.elf

# Generate boot file
cat > boot.bif << EOF
image : {
    [bootloader, destination_cpu = a53-0]fsbl.elf
    [destination_cpu = pmu]pmufw.elf
    [destination_cpu = a53-0, exception_level = el-3, trustzone]bl31.elf
    [destination_cpu = a53-0, exception_level = el-2]u-boot.elf
}
EOF
bootgen -arch zynqmp -image boot.bif -o boot.bin
```

A.8 build_kernel.sh

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/8_build_kernel.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Apply a patch to enable the required features and compile the kernel.
#
#-------------------------------------------------------------------------

cd $WD/linux-xlnx
git checkout -b wb "$GITTAG_LINUX"

make -j $(nproc) xilinx_zynqmp_defconfig

# Apply the kernel patch to enable all required features, which are:
#
# *[DISABLE] General setup->Initial RAM filesystem and RAM disk (
#                 initramfs/initrd) support
#
# *[DISABLE] Bus cupport->PCI support
#
# *[ENABLE]  Device Drivers->Input device support->Mouse interface
# *[ENABLE]  Device Drivers->Staging drivers->Xilinx APF Acelerator
#                 Driver->Xilinx APF DMA engines support
#
# *[ENABLE]  Kernel hacking->Tracers->Kernel Function Tracer
#
patch $WD/linux-xlnx/.config $PATCHES_DIR/kernel/\
0001-enable-required-features.patch

# Compile the kernel
make -j $(nproc)
```

A.9 prepare_sd_card.sh

```
#!/bin/bash
# ------------------------------------------------------------------------
# ------------------------- Doom TFM (CEI 2017/2018)---------------------
# ------------------------------------------------------------------------
#         Desktop image zcu102 project
#                 File: - "scripts/11_prepare_sd_card.sh"
#
# ------------------------------------------------------------------------
# Authors: David Lima       (davidlimaastor@gmail.com)
#          Leonardo Suriano (leonardo.suriano@upm.es)
#
# Description:
#   Prepare the sd card image with the linaro debian 9 FS with required
#   mali userspace drivers and mali.ko module.
#
# TODO:
#        - Add variables to use instead of paths.
#
# CHANGELOG:
#   [11-9-2018] Mali drivers:
#     - Defined loop10 as loop device
#     - Mali userspace drivers steps
#     - Module mali.ko compilation
#     - Xorg drivers and configuration
#     - Script install_desktop.sh
#   [19-11-2018] Now It uses the first unused loop device
#
#-------------------------------------------------------------------------

cd $WD/rootfs

# Create SD image file
dd if=/dev/zero of=desktop_image_zcu102.img bs=1M count=4k status=progress

# Mount SD image file in loop device
sudo -H losetup -D

# Get first empty loop device
LOOP_DEVICE=$(losetup -f)
```

```
sudo -H losetup -P $LOOP_DEVICE desktop_image_zcu102.img

#
# Create partitions
#
# to create the partitions programatically (rather than manually)
# we're going to simulate the manual input to fdisk
# The sed script strips off all the comments so that we can
# document what we're doing in-line with the actual commands
# Note that a blank line (commented as "default" will send a empty
# line terminated with a newline to take the fdisk default.
#
sed -e 's/\s*\([\+0-9a-zA-Z]*\).*/\1/' << EOF | sudo -H fdisk $LOOP_DEVICE
  o # clear the in memory partition table
  n # new partition
  p # primary partition
  1 # partition number 1
    # default - start at beginning of disk
  +100M # 100 MB boot parttion
  n # new partition
  p # primary partition
  2 # partion number 2
    # default, start immediately after preceding partition
    # default, extend partition to end of disk
  a # make a partition bootable
  1 # bootable partition is partition 1 -- /dev/sda1
  t # change a partition type
  1 # target partition is partition 1
  c # partition type is W95 FAT32 (LBA)
  p # print the in-memory partition table
  w # write the partition table
  q # and we're done
EOF

# Format partitions
LOOP_DEVICE_P1="${LOOP_DEVICE}p1"
LOOP_DEVICE_P2="${LOOP_DEVICE}p2"

sudo -H mkfs.vfat $LOOP_DEVICE_P1
sudo -H mkfs.ext4 $LOOP_DEVICE_P2
```

```
# Mount partitions
mkdir $WD/rootfs/boot
mkdir $WD/rootfs/root
sudo -H mount -o defaults,uid=1000,gid=1000 $LOOP_DEVICE_P1 $WD/rootfs/boot
sudo -H mount -t ext4 $LOOP_DEVICE_P2 $WD/rootfs/root

# Set up boot partition
cp $WD/boot/boot.bin $WD/rootfs/boot
cp $WD/linux-xlnx/arch/arm64/boot/Image $WD/rootfs/boot
cp $WD/devicetree/zcu102/dts/system.dtb $WD/rootfs/boot
sync

# Download Linaro (Debian) and extract rootfs
wget http://releases.linaro.org/debian/images/developer-arm64/18.04/\
linaro-stretch-developer-20180416-89.tar.gz
sudo -H tar --strip-components=1 \
        -xzphf linaro-stretch-developer-20180416-89.tar.gz -C root
sync

# Create script to initialize ARTICo3
cat > $WD/rootfs/setup.sh << EOF
# Load ARTICo3 device tree overlay
echo "Loading ARTICo3 device tree overlay..."
mkdir /sys/kernel/config/device-tree/overlays/artico3
echo overlays/artico3.dtbo > /sys/kernel/config/device-tree/\
overlays/artico3/path

# Load DMA proxy driver
echo "Loading DMA proxy driver..."
modprobe mdmaproxy

# Remove kernel messages from serial port
echo 1 > /proc/sys/kernel/printk
EOF
chmod +x $WD/rootfs/setup.sh

# Create script to move kernel modules to /lib/modules/...
cat > $WD/rootfs/artico3_init.sh << EOF
mkdir -p /lib/modules/\$(uname -r)
```

```
mv /root/mdmaproxy.ko /lib/modules/\$(uname -r)
touch /lib/modules/\$(uname -r)/modules.order
touch /lib/modules/\$(uname -r)/modules.builtin
depmod
rm -f /root/artico3_init.sh
EOF
chmod +x $WD/rootfs/artico3_init.sh

# Copy additional files
sudo -H mkdir -p $WD/rootfs/root/lib/firmware/overlays
#
# TODO: Discomment this ?
#
#sudo -H mv £WD/rootfs/artico3.dts £WD/rootfs/root/lib/firmware/overlays
#sudo -H mv £WD/rootfs/artico3.dtbo £WD/rootfs/root/lib/firmware/overlays
#sudo -H cp £WD/artico3/linux/drivers/dmaproxy/mdmaproxy.ko \
        $WD/rootfs/root/root
#sudo -H mv £WD/rootfs/setup.sh £WD/rootfs/root/root
#sudo -H mv £WD/rootfs/artico3_init.sh £WD/rootfs/root/root
sync


# Create temp dir to store temp files
mkdir $WD/temp
cd $WD/temp

# Mali userspace drivers:
# Xilinx release
REL="rel-v2018.1"
# Driver version
RP="r8p0-01rel0"

# Download mali userspace drivers
wget https://www.xilinx.com/publications/products/tools/\
        mali-400-userspace.tar

# Extract until have the libraries
tar -xf mali-400-userspace.tar -C $WD/temp/
tar -xf $WD/temp/mali/$REL/downloads-mali.tar -C $WD/temp/
mkdir $WD/temp/mali-userspace.git
```

```
tar -xf $WD/temp/downloads-mali/git2_gitenterprise.xilinx.com.Graphics.\
        mali400-xlnx-userspace.git.tar.gz -C $WD/temp/mali-userspace.git
git clone $WD/temp/mali-userspace.git $WD/temp/mali-userspace

# Copy to rootfs
sudo -H cp -r $WD/temp/mali-userspace/$RP/aarch64-linux-gnu/x11/usr/ \
        $WD/rootfs/root/


# mali.ko compilation
# Extracted source code directory
drv_dir="$WD/temp/DX910-SW-99002-$RP/driver/src/devicedrv/mali"

# Patches directory
#patches_mali="£WD/temp/patches/kernel-module-mali"
#mkdir -p £patches_mali

# Download abd extract mali utgard drivers source code from ARM website
wget https://armkeil.blob.core.windows.net/developer/Files/downloads/\
        mali-drivers/kernel/mali-utgard-gpu/DX910-SW-99002-r8p0-01rel0.tgz
tar -xf DX910-SW-99002-$RP.tgz -C $WD/temp/

# Download and apply patches
# cd £patches_mali

patch $drv_dir/Makefile                         $PATCHES_DIR/mali/\
        kernel_module/0001-Change-Makefile-to-be-compatible-with-Yocto.patch
patch $drv_dir/platform/arm/arm.c               $PATCHES_DIR/mali/kernel_\
        module/0002-staging-mali-r8p0-01rel0-Add-the-ZYNQ-ZYNQMP-platfor.patch
patch $drv_dir/linux/mali_linux_trace.h         $PATCHES_DIR/mali/kernel_\
        module/0003-staging-mali-r8p0-01rel0-Remove-unused-trace-macros.patch
patch $drv_dir/platform/arm/arm.c               $PATCHES_DIR/mali/kernel_\
        module/0004-staging-mali-r8p0-01rel0-Don-t-include-mali_read_phy.patch
patch $drv_dir/linux/mali_kernel_linux.c        $PATCHES_DIR/mali/kernel_\
        module/0005-linux-mali_kernel_linux.c-Handle-clock-when-probed-a.patch
patch $drv_dir/platform/arm/arm.c               $PATCHES_DIR/mali/kernel_\
module/0006-arm.c-global-variable-dma_ops-is-removed-from-the-ke.patch

# Patch 0007 and 0008 modify two files each
cd $drv_dir/linux/
```

```
patch -i                                          $PATCHES_DIR/mali/kernel_\
module/0007-Replace-__GFP_REPEAT-by-__GFP_RETRY_MAYFAIL.patch
patch -i                                          $PATCHES_DIR/mali/kernel_\
module/0008-mali_internal_sync-Rename-wait_queue_t-with-wait_que.patch

patch $drv_dir/linux/mali_memory_swap_alloc.c   $PATCHES_DIR/mali/kernel_\
module/0009-mali_memory_swap_alloc.c-Rename-global_page_state-wi.patch
patch $drv_dir/common/mali_pm.c                 $PATCHES_DIR/mali/kernel_\
module/0010-common-mali_pm.c-Add-PM-runtime-barrier-after-removi.patch
patch $drv_dir/linux/mali_kernel_linux.c        $PATCHES_DIR/mali/kernel_\
module/0011-linux-mali_kernel_linux.c-Enable-disable-clock-for-r.patch

# Change to source directory and compile the module
cd $drv_dir
KDIR=$WD/linux-xlnx ARCH=arm64 BUILD=release MALI_PLATFORM="arm" \
        USING_DT=1 MALI_SHARED_INTERRUPTS=1 MALI_QUIET=1 make
# NOTE: If you try to compile with -j$(nproc) it fails.

# Copy to rootfs
sudo -H cp $drv_dir/mali.ko $WD/rootfs/root/root/
sync

# Modify the rc.local file to load the module automatically each boot
sudo -H sed -i '21i insmod /root/mali.ko' $WD/rootfs/root/etc/rc.local

# xorg-driver and xorg-xserver
# This is the version that should be used with 2017.1:
# GITTAG_XF86="1.3.0"
# NOTE: Compile errors with this version

# For 2018.1:
GITTAG_XF86="1.4.1"

# Extracted source code directory
xorg_dir="$WD/temp/xf86-video-armsoc"
# Patches directory
#patches_xorg="$WD/temp/patches/xf86-video-armsoc"
#mkdir -p $patches_xorg

# Download repo
```

```
cd $WD/temp
git clone https://gitlab.freedesktop.org/xorg/driver/xf86-video-armsoc
cd $xorg_dir
echo "------------------------------------------------------------------"
#git checkout -b "wb" £GITTAG_XF86

# Download and apply patches
#cd £patches_xorg


# The patch modifies multiple files
cd $xorg_dir/src
patch -i $PATCHES_DIR/mali/xf86_video_armsoc/\
        0001-src-drmmode_xilinx-Add-the-dumb-gem-support-for-Xili.patch
# One of the patches files is wrong located
mkdir -p $xorg_dir/src/drmmode_xilinx
mv $xorg_dir/src/drmmode_xilinx.c $xorg_dir/src/drmmode_xilinx/\
        drmmode_xilinx.c

# Copy the complete folder, it will be compiled after xfce4 installation
sudo -H cp -r $WD/temp/xf86-video-armsoc $WD/rootfs/root/home/linaro

cd $WD/temp

# Xorg configuration file
wget https://raw.githubusercontent.com/Xilinx/meta-xilinx/rel-v2018.1/\
        meta-xilinx-bsp/recipes-graphics/xorg-xserver/xserver-xf86-config/\
        zynqmp/xorg.conf

# Copy to rootfs, it will be moved after xfce installation
sudo -H cp $WD/temp/xorg.conf $WD/rootfs/root/home/linaro/
sync

# Script to set up the desktop-like environment in the zcu102
cat > $WD/rootfs/desktop_install.sh << EOF
#!/bin/sh

# Update and upgrade from repos
sudo -H apt-get update
printf 'Y' | sudo -H apt-get upgrade
```

```
# Install desktop-like environment packages
printf 'Y' | sudo -H apt-get install xfce4 xorg-dev libudev-dev \
        xutils-dev autoconf libtool autogen

# Configure xorg
sudo -H chmod 777 xorg.conf
sudo -H cp xorg.conf /usr/share/X11/xorg.conf.d/

# Compile and install armsoc.so driver
sudo chmod 777 /home/linaro/xf86-video-armsoc/*
cd /home/linaro/xf86-video-armsoc
sudo -H autoreconf -fi
sudo -H ./configure --with-drmmode=xilinx --enable-maintainer-mode
sudo -H make
sudo -H make install

# Change te location of armsoc.so file
sudo -H cp /usr/local/lib/xorg/modules/drivers/armsoc_drv.so \
        /usr/lib/xorg/modules/drivers/
sudo chmod 777 /usr/lib/xorg/modules/drivers/armsoc_drv.so

# Reboot
sudo -H reboot
EOF
chmod +x $WD/rootfs/desktop_install.sh

sudo -H mv $WD/rootfs/desktop_install.sh $WD/rootfs/root/home/linaro/
sync

# Unmount partitions
sudo -H umount $LOOP_DEVICE_P1
sudo -H umount $LOOP_DEVICE_P2
rm -rf $WD/rootfs/boot
rm -rf $WD/rootfs/root

# Remove SD image file from loop device
sudo -H losetup -D

cd $WD
```

REFERENCES

REFERENCES

[1] K. Rupp, "42 Years of Microprocessor Trend Data," https://github.com/karlrupp/microprocessor-trend-data, [Online; Acceso 15-01-2018].

[2] M. A. E. R. W. S. Louise H. Crockett, Ross A. Elliot, *Embedded Processing with the ARM®Cortex™-A9 on the Xilinx®Zynq®-7000 All Programmable SoC*, 1st ed. Strathclyde Academic Media, 2014.

[3] A. Otero, R. Salvador, J. Mora, E. de la Torre, T. Riesgo, and L. Sekanina, "A fast reconfigurable 2d hw core architecture on fpgas for evolvable self-adaptive systems," in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*. IEEE, 2011, pp. 336–343.

[4] R. Salvador, A. Otero, J. Mora, E. de la Torre, L. Sekanina, and T. Riesgo, "Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE, 2011, pp. 164–169.

[5] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[6] H. D. Foster, "Why the design productivity gap never happened," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 581–584.

[7] A. Takach, "High-level synthesis: Status, trends, and future directions," *IEEE Design & Test*, vol. 33, no. 3, pp. 116–124, 2016.

[8] I. Xilinx, "Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit," https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, [Online; Acceso 09-01-2018].

[9] ——, "ZCU102 Evaluation Board, User Guide 1182," https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf, [Online (25-01-2019)].

[10] ——, "Zynq UltraScale+ MPSoC Data Sheet: Overview," https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, 2018, [Online; Acceso 09-01-2018].

[11] ——, "UltraScale Architecture," https://www.xilinx.com/products/technology/ultrascale.html, [Online; Acceso 15-01-2018].

[12] ——, "Arm trusted firmware xilinx repository," https://github.com/xilinx/arm-trusted-firmware, [Online; Acceso 09-01-2018].

[13] "Das U-Boot – the Universal Boot Loader," http://www.denx.de/wiki/U-Boot/, [Online; Acceso 15-01-2018].

[14] I. Xilinx, "The official Xilinx u-boot repository," https://github.com/Xilinx/u-boot-xlnx, [Online; Acceso 15-01-2018].

[15] ——, "Vivado Design Suite," https://www.xilinx.com/products/design-tools/vivado.html, [Online; Acceso 15-01-2018].

[16] ——, "SDSoC Development Environment," https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html, [Online; Acceso 15-01-2018].

[17] ——, "The official Linux kernel from Xilinx ," https://github.com/Xilinx/linux-xlnx/tree/xilinx-v2018.1, [Online; Acceso 15-01-2018].

[18] ——, "Building and Running the Ubuntu Desktop from sources," https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841937/Zynq+UltraScale+MPSoC+Ubuntu+part+2+-+Building+and+Running+the+Ubuntu+Desktop+From+Sources, [Online; Acceso 09-01-2018].

[19] Linaro, "Linaro Overview," https://www.linaro.org/about/, [Online; Acceso 15-01-2018].

[20] I. Xilinx, "Download: Arm mali 400 support binaries for zcu102," https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk/arm-mali-400-software-download.html, [Online; Acceso 09-01-2018].

[21] ——, "Open source mali utgard gpu kernel drivers," https://developer.arm.com/products/software/mali-drivers/utgard-kernel, [Online; Acceso 09-01-2018].

[22] IdGames, "DOOM source code," https://www.doomworld.com/idgames/idstuff/source/doomsrc, [Online; Acceso 15-01-2018].

[23] D. World, "DOOM source ports," https://doom.fandom.com/wiki/Source_port, [Online; Acceso 15-01-2018].

[24] "Crispy DOOM official repository," https://github.com/fabiangreffrath/crispy-doom, [Online; Acceso 15-01-2018].

[25] "Chocolate-doom Website," https://www.chocolate-doom.org/wiki/index.php/Chocolate_Doom, [Online; Acceso 15-01-2018].

[26] "Doom archive: Shareware version," http://www.doomarchive.com/ListFiles.asp?ContentsFolderId=216&FolderId=216, [Online; Acceso 09-01-2018].

[27] I. Xilinx, "PetaLinux Tools," https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html, [Online; Acceso 15-01-2018].

[28] "The Yocto Project," https://www.yoctoproject.org/, [Online; Acceso 15-01-2018].

[29] "The Yocto Project Software overview," https://www.yoctoproject.org/software-overview/, [Online; Acceso 15-01-2018].

[30] "Meta-layer for Poky to build embedded Linux exvironments by Debian's source codes ," https://github.com/meta-debian/meta-debian, [Online; Acceso 15-01-2018].

[31] I. Xilinx, "`meta-petalinux` "distro" layer GitHub repository," https://github.com/Xilinx/meta-petalinux, [Online; Acceso 09-01-2018].

[32] "install.sh The Easy Start (ReconOS)," http://www.reconos.de/gettingstarted/tutorial/, [Online; Acceso 15-01-2018].

[33] "Github issue comment of CPU Usage," https://github.com/chocolate-doom/chocolate-doom/issues/418#issuecomment-48192780, [Online; Acceso 15-01-2018].

[34] "`stretch2x` function repository," https://bitbucket.org/d_lima/stretch2x_tfm/src/master/, [Online; Acceso 09-01-2018].

[35] "Vanilla doom aspect ratio," https://doom.fandom.com/wiki/Aspect_ratio, [Online; Acceso 09-01-2018].

[36] "Cmake project official repository," https://bitbucket.org/d_lima/stretch2x_cmake/src/master/, [Online; Acceso 09-01-2018].

[37] I. Xilinx, "Vivado High-Level Synthesis tools website," https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, [Online; Acceso 15-01-2018].

[38] ——, "AXI Reference Guide, UG1037," https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, 2017, [Online; Acceso 09-01-2018].

[39] ——, "Sdsoc development environment web page," https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html, [Online; Acceso 09-01-2018].

[40] ——, "Directives sdsoc, sds pragmas," https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/nmc1504034362475.html, [Online; Acceso 09-01-2018].

[41] "Stretch2x sdsoc project repository," https://bitbucket.org/d_lima/stretch2x_sdsoc/src/master/, [Online; Acceso 09-01-2018].

[42] A. D. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Design & Test*, vol. 34, no. 1, pp. 77–90, 2017.

[43] I. Xilinx, "Zcu102 evaluation kit platform user guide," https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf, [Online; Acceso 09-01-2018].

[44] A. Rodríguez, J. Valverde, J. Portilla, A. Otero, T. Riesgo, and E. de la Torre, "Fpga-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The artico3 framework," *Sensors*, vol. 18, no. 6, p. 1877, 2018.

Acronyms

PCB: *Printed Circuit Boards.*

FPGA: *Field-Programmable Gate Array.*

SoC: *System-on-Chip.*

SoPC: *System-on-Programmable-Chip.*

MPSoC: *Multi-Processing System-on-Chip.*

CPU: *Central Processing Unit.*

AXI: *Advanced eXtensible Interface.*

PS: *Processing System.*

PL: *Programmable Logic.*

APU: *Application Processing Unit.*

MPE: *Media Processing Engine.*

FPU: *Floating Point Unit.*

MMU: *Memory Management Unit.*

OCM: *On-Chip Memory.*

SCU: *Snoop Control Unit.*

RPU: *Real-Time Processing Unit.*

GPU: *Graphics Processing Unit.*

PMU: *Power Management Unit.*

HLS: *High Level Synthesis.*

DMA: *Direct Memory Access.*

IP: *Integrated Property.*

APSoC: *All Programmable System On Chip.*

OS: *Operating System.*

FSBL: *First State Boot Loader.*

SSBL: *Second State Boot Loader.*

BIOS: *Basic Input/Output System.*

MBR: *Master Boot Record.*

PCI: *Peripheral Component Interconnect.*

FPS: *Frames Per Second.*

APT: *Advanced Packaging Tool.*

RTL: *Register-Transfer Level.*

DSE: *Design Space Exploration.*

IoT: *Internet of Things.*

BIF: *Boot Information File.*

ATF: *Arm Trusted Firmware.*