Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Chinmayi Hassan Shyamprasad Nadig

Analyzing and improving graphics processing performance in microcontrollers

Master's thesis in Electronic Systems Design

September 2020

NTNU
Norwegian University of Science and Technology

NORDIC SEMICONDUCTOR

ANALYZING AND IMPROVING GRAPHICS PROCESSING PERFORMANCE IN MICROCONTROLLERS

MASTER'S THESIS IN ELECTRONIC SYSTEMS DESIGN

CHINMAYI HASSAN SHYAMPRASAD NADIG

SEPTEMBER 2020

Academic Supervisor: Trond Ytterdal
Professor, Department of Electronic Systems

Industrial Supervisor: Martin Olsson
Senior R&D engineer, Nordic Semiconductor ASA

Abstract

A typical microcontroller unit (MCU) has limited capabilities for processing and displaying graphics, due to power and size constraints. An increasing demand for rich graphical user interface (GUI) applications in battery powered systems motivates microcontroller vendors to include additional hardware to accelerate graphics processing. The goal of this master thesis was to analyze the graphics processing capability of a typical microcontroller and to explore different architectures for improving performance. A RISC-V ISA based simulator of a generic, heterogeneous, and multi-core system on chip (SoC) with shared memory and I/O written using SystemC + TLM provided the hardware environment needed for the analysis and exploration. LVGL which is a embedded graphics library was used for writing the application code for this hardware simulator.

The main phases of the thesis were - setting up the hardware simulator environment, setting up a benchmarking framework on the application code, doing baseline performance analysis and arriving at possible areas for improvement, designing architectural improvements and exploring various scenarios. The two improvements which were analyzed and performed were adding direct memory access (DMA) capability to the basic display controller, and designing a hardware accelerator for offloading fill and blend operations from the CPU, also with DMA. When these two were used together for drawing different scenes, an average 68% reduction in the cycles was obtained compared to the cycles taken to render it in the baseline scenario, thus increasing the processing speed of the application. In addition to this reduction, an average of 18% of the cycles taken were saved, thus freeing up the CPU to do something else during these cycles.

Keywords: *MCU, RISC-V, Hardware Accelerator, TLM, Graphics Processing*

Preface

This report has been written in Spring 2020 to fulfill the requirements of the Master's thesis and has been submitted to the department of Electronic Systems Design at Norwegian University of Science and Technology (NTNU). This work is done in collaboration with Nordic Semiconductor and is a continuation of the specialization project done during Fall 2019.

I would like to first thank Nordic Semiconductor, especially the System Architecture Group, for providing the necessary environment to work comfortably for a year. Biggest thanks to my supervisor, Martin Olsson, who has been the backbone of this thesis by providing invaluable guidance and support since day 1.

Special thanks to my supervisor at NTNU, Trond Ytterdal, for his support and cooperation throughout. His understanding nature helped me complete this work in the midst of the various challenges I faced. My sincere gratitude to my late professor, Kjetil Svarstad, for motivating me to take up the project during Fall 2019 which formed the foundation for this thesis. I wish he could have been there till the end to see me submit this work, but I am fortunate that I got an opportunity to work with him.

The acknowledgements are incomplete without thanking my family for giving me constant comfort and happiness to push through. Big hug and thank you to my partner, Abhilash, who has been my rock for the past decade. Thank you to my mother and my pet dog, Shibu, for checking up on me everyday from 7500kms away, talking to them is a highlight of my day. Finally, thank you to all my friends, colleagues, and everyone else who helped and supported me in finishing this work.

September 2020
Chinmayi Nadig

Table of Contents

List of Tables

List of Figures

Listings

Abbreviations

GUI	=	Graphical User Interface
MCU	=	Microcontroller Unit
SoC	=	System on Chip
RAM	=	Random Access Memory
DMA	=	Direct Memory Access
ISA	=	Instruction Set Architecture
TLM	=	Transaction Level Modelling
RGB	=	Red Green Blue
RGB	=	Red Green Blue Alpha
API	=	Application Programming Interface
SDL	=	Simple DirectMedia Layer
CSR	=	Control and Status Register

Chapter 1

Introduction

This chapter first talks about the motivation for selecting this topic, then the objective of the thesis, the methodology followed, contribution made by the author and ends with an overview of the chapters in the report.

1.1 Motivation and Objective

GUIs are omnipresent in today's world. Computers are conventional devices which can drive excellent quality graphics, but they are general-purpose and power-hungry. The demand for rich GUIs in battery powered systems like watches, smartphones, medical devices, handheld gaming devices to name a few is on the rise [16]. Many of these specialized devices employ MCUs because they cost less, are not power intensive, and are relatively less complex compared to general-purpose computers. Typically, MCUs have limited capabilities for driving graphics owing to the power and size constraints. Therefore, MCUs with graphics capabilities is a niche and developing market [9].

A theoretical study [19] was done as a part of the specialization project during Fall 2019 which discussed and compared the various hardware architectures which can be used for accelerating graphics processing performance in MCUs along with the various kinds of software libraries which can be used to write the GUI code. Of the various graphics libraries compared in the project, LVGL was concluded to be best suited for use in research because it is free, open-source, lightweight, and has support for hardware acceleration. LVGL is therefore used for writing graphics application code in this thesis.

The aim of this thesis is to keep the specialization project as a theoretical foundation and develop upon it. The thesis focuses on analysis of the graphics processing capability of a typical MCU and seeks to explore various architecture topologies for improving the same. A simulator of a SoC is needed for performance analysis and architectural exploration. A RISC-V ISA based simulator written in SystemC + TLM is used and the reason for this choice is elaborated in the background chapter. To explain briefly, Nordic Semicondutor currently uses the ARM ISA which is licensed. RISC-V on the other hand is free, open source, and royalty-free [4]. It has huge potential for use in research and education due to its simple, modular and extensible nature. SystemC + TLM is used to write the simulator because it is highly suited for use in SoC modelling and architectural exploration owing to its high level of abstraction [7]. Having a RISC-V based simulator model in SystemC + TLM which can run the same software code as ARM makes it the perfect choice for use in this thesis.

The thesis primarily seeks to be a study backed by simulation results which aims to understand graphics subsystem in microcontrollers first, set up the system on a simulator, add a benchmarking framework to it, and then analyze its performance along with exploring various architectures to improve the graphics processing performance.

1.2 Methodology

The first part of the project was literature review. The search engines used for this purpose were NTNU's Oria, Scopus, ACM digital library, Google Scholar, and IEEE Xplore. The thesis can be broadly divided into three phases - setting up phase, benchmarking phase, and architectural exploration phase.

The setting up phase consists of setting up the simulator and running the LVGL graphics library for writing GUI code on it. The benchmarking phase consists of setting up timing information for the hardware components of the simulator and benchmarking the application code running on the simulator. The benchmarking phase encompasses the baseline analysis phase where the graphics processing performance is analyzed on a baseline architecture and the areas for improving the performance are identified. This phase also provides us a quantitative way to analyze the improvement in performance. The architecture exploration phase consists of designing architectural improvements and exploring various architectural topologies for improving the graphics processing performance.

1.3 Contributions

The contributions made by the author are:

- Explaining how to set up a RISC-V based simulator of a SoC which is configurable and represents a modern day, generic, heterogeneous, and multi-core SoC systematically. The code listings are also added in the appendix and they can be referred to understand the process better. Here, the simulator is used for graphics processing analysis, but it can also be used for other types of analyses.
- Setting up timing in the TLM models of hardware components in the simulator. This helps to mirror a real-life situation where there are different kinds of components with different latencies connected together in a SoC and quantify the time spent in doing different operations. This general methodology can be referred to set up timing in other TLM models.
- Here, the graphics application code is benchmarked to understand which operations act as a bottleneck on the simulation speed. The methodology can also be followed to benchmark other kinds of application code.
- The graphics processing performance is analyzed in a baseline architecture configuration and then improvements are identified, designed, and explored. The ease of adding new components, changing the topologies, and adding latencies to perform architectural exploration on TLM models is demonstrated here.

1.4 Report Structure

Chapter 1 - Introduction
Chapter 1 introduces the thesis along with its motivation and objective. The methodology adopted in this thesis has then been described along with the contribution made by the author. The chapter concludes with the report structure.

Chapter 2 - Background
Chapter 2 provides the background theory needed to read this report. It first explains the graphics subsystem in a MCU environment and also introduces the graphics software libary LVGL which will be used for writing GUI code. The chapter also explains why a RISC-V ISA based simulator of a SoC written using SystemC + TLM is used in the thesis by providing relevant background.

Chapter 3 - System Setup
Chapter 3 provides a detailed account of how the SoC simulator was set up, the graphics subsystem of a typical MCU emulated on it, and a simple graphics test written using LVGL was run on it. The chapter is divided into sub-chapters as needed to break up the process and for easy understanding.

Chapter 4 - Benchmarking
Chapter 4 is an integral one. The first part explains the concept of timing in TLM models and the next part explains how timing is set up in the TLM models of the hardware components which make up the simulator. The middle part explains how the benchmarking framework is set up to analyze the breakdown of the cycles taken to draw a scene to the display. The last two parts of the chapter are results and discussion where the baseline performance is analyzed by drawing different scenes to the display and the bottleneck operations which hog the most cycles are identified for improvement.

Chapter 5 - Architectural Exploration
Chapter 5 begins by presenting the design of two architectural improvements - Adding DMA to the display controller model, and a hardware accelerator model for accelerating blend and fill drawing operations, also with DMA. These improvements are then explored under various scenarios in the results section and the observations are summarized in the discussion section.

Chapter 6 - Conclusion
Chapter 6 summarizes the entire work done and concludes the thesis by presenting the future scope of the work.

Chapter 2

Background

This chapter helps to establish the theoretical knowledge needed to understand the report. The first part provides an overview of graphics subsystem in a MCU by explaining its high-level architecture. It also introduces the GUI software library LVGL which is used for application code development in this thesis. The next part explains modelling systems using SystemC + TLM and its advantages. The final part talks about the RISC-V ISA and why it is used in processors. The first parts provide relevant background knowledge, but the last two parts along with providing knowledge answer why a RISC-V ISA based simulator written using SystemC + TLM is used in this thesis.

2.1 Graphics Subsystem

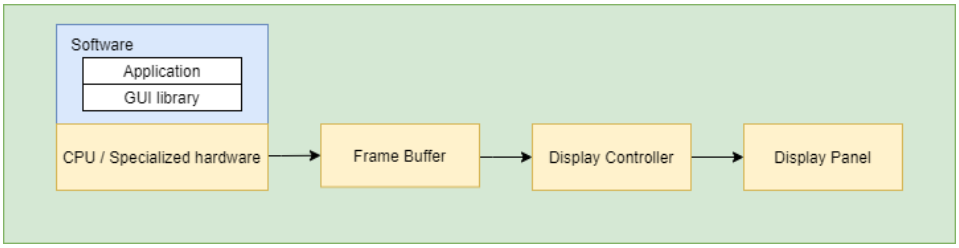

Figure 2.1: The data flow to rendering graphics in a graphics subsystem

The basic flow to displaying graphics is as shown in the figure 2.1. The application code can be run on the CPU or a specialized hardware unit. The code is usually written using a GUI software library. When the code runs, the graphical instructions build the image in the frame buffer which is a memory unit. The display controller is responsible for picking up the image built from the frame buffer and driving it to a display panel which displays it [15].

2.1.1 Components

The primary components which make up a graphics subsystem are the microcontroller, software, frame buffer, display controller, and the display panel which are described briefly in the following subsections.

Microcontroller

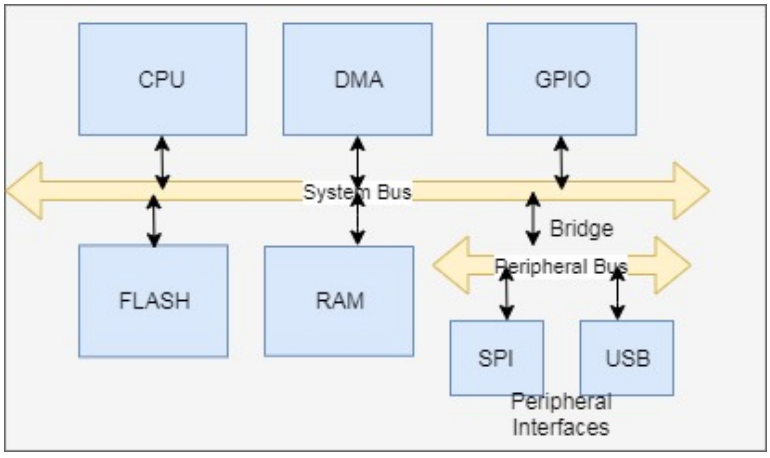

Figure 2.2: General architecture of a MCU

Figure 2.2 shows the general architecture of an MCU. The system bus is connected to the Flash Memory, RAM, DMA (optional) and General Purpose Input Output (GPIO) units. Units for interfacing with the outside world, called the peripheral units are connected through a peripheral bridge which is connected to the system bus via a bridge. The application code is run on the CPU, but it can also be made to run on a dedicated hardware unit. The graphical instructions build the image in the frame buffer. The frame buffer is a memory unit which can be present internally in the MCU or be external to the MCU [3].

Software

The application code which runs on the microcontroller is called the software. It is usually written using a GUI software library. A GUI library has callback functions to the driver of the hardware units if present. The library helps to set up the GUI by implementation of APIs for drawing fundamental shapes, 2D image processing and providing support for hardware acceleration of graphics functions [3].

Frame Buffer

It is also known as the Graphic Random Access Memory (GRAM). The frame buffer is a volatile memory space that is used for storing the final image that is shown on the screen. Its size depends on the resolution of the display and the color depth.

Frame buffer size (Bytes) = Number of Pixels x Color Depth (Bits)/8

Example: For a display at 24 bpp color depth and resolution of 480x272, the frame buffer memory required is 480x272x24/8 = 391,680 Bytes (392 kB)

It can be stored in the microcontroller RAM, in an external RAM or integrated in the display controller. Double buffering (having two frame buffers) is commonly used to avoid a glitch called tearing which occurs when two frames are displayed simultaneously. In double buffering, one buffer is used for drawing to compose the next image while the other stores the current image and is driven to display [20].

Display Controller

The purpose of the display controller is to transfer the contents of the frame buffer to the display panel. In this way, it continuously refreshes the display panel and the frequency with which this is done is called the refresh rate. If the screen is refreshed 60 times in a second, then the refresh rate is 60Hz. The display controller can be present either in the MCU or external to the MCU [20].

Display Panel

The display panel is driven by the display controller and it displays the final image. The data is driven to it by the controller from the frame buffer by formatting it. The data output to the panel has many signals for synchronization. Display panels come in many varieties and sizes, and they are chosen depending on the preference of the system [20].

2.1.2 Configurations

The components of a graphics subsystem can be connected together in different configurations. The most common configurations are:

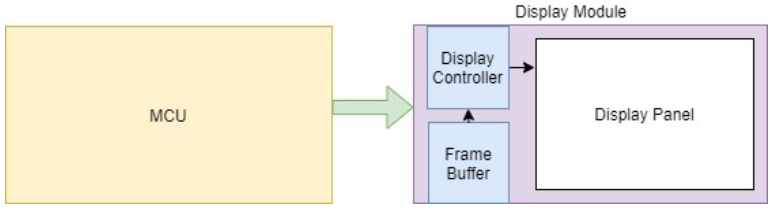

Figure 2.3: Configuration 1: Display module with frame buffer and display controller

The configuration in figure 2.3 is used in MCUs which do not have built-in graphics support. The frame buffer and the display controller are both located on a display module and connected to the MCU through it. A serial interface like SPI is used for the transfer of data from the MCU to the display module.

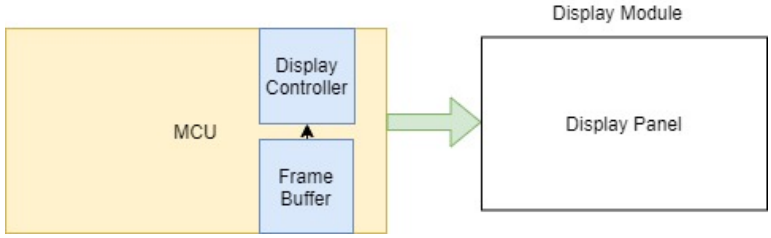

Figure 2.4: Configuration 2: MCU with frame buffer and display controller present on-chip

As compared to the previous configuration, the configuration in figure 2.4 has both the frame buffer and display controller present on the MCU and connected to an external display panel. This configuration can lead to significant savings in terms of memory accesses as having an internal frame buffer maximizes performance and minimizes bandwidth limitations for the display controller. A parallel interface like RGB is used for the transfer of data from the MCU to the display module.

Another advantage is that on-the-fly custom transformations of pixel data is possible when the controller is on the chip. It is also a configuration preferred by some customers as they can buy a simple display panel cheaply and there is no need to buy one with both the frame buffer and controller.

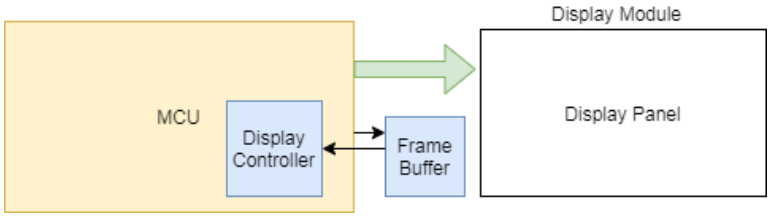

Figure 2.5: Configuration 3: MCU with display controller on-chip and external frame buffer

The configuration in figure 2.5 is similar to the one above and offers the same advantages. The only difference is the frame buffer can be external in case of storage constraints on the chip.

2.1.3 GUI library - LVGL

The GUI library which is used for application code development in this thesis is LVGL [12]. It is a graphical library which helps to create embedded GUI. Some of its key features are:

- Open-source and free under the MIT license.
- Hardware-independent and can be used without any MCU or display.
- Written in C but also compatible with C++.
- Made of building blocks like lists, blocks, charts, images and also supports advanced graphics like animations, opacity, anti-aliasing, etc.
- All the graphic elements are fully customizable.
- Supports multi input devices and multi displays.
- Has a very less memory footprint and is scalable.
- Can also support OS, external memory and GPU.

The reason it is used in this thesis is because it is open-source, completely free, easy to use, hardware-independent, and supports external memory and GPU. It can start an embedded GUI design by running it on its PC simulator environment. This offers a major advantage of writing and testing real LVGL applications without the need for embedded hardware. There are plenty of tutorials, examples, themes, and documentation which facilitates its ease of use and GUI designing. The structure of LVGL is as shown in figure 2.6 [27].

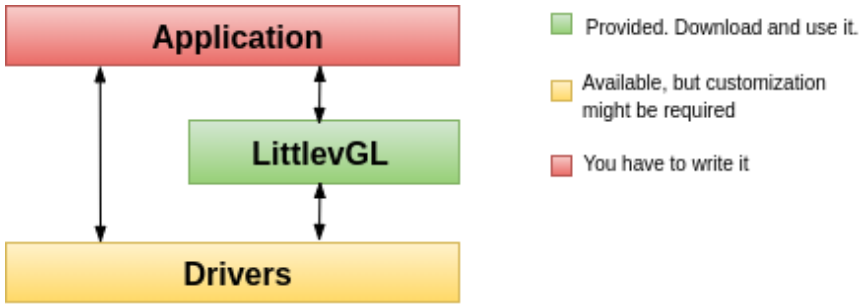

Figure 2.6: Structure of LVGL [27]

Application creates the GUI and handles the tasks. It is written using the LVGL API which makes calls to the functions of the LVGL library or directly to the drivers.

LVGL is the layer with which the application communicates to create the GUI. It registers the input and display device drivers using a Hardware Abstraction Layer (HAL).

Drivers contains functions which make calls to functions that drive the display, to hardware accelerator/GPU, or to the display controller.

2.2 SystemC + TLM modelling

The SoC designs today are complex with multiple heterogeneous processors, on-chip buses and caches, peripheral control devices, and hardware accelerators for dedicated functions. There are a growing number of Intellectual Property (IP) blocks that interact through bus technologies or networks on chip (NoC). Using the traditional RTL modelling for design and verification takes too much effort for development, provides slow simulation, and it is not ready early in the design flow for architectural exploration and early hardware/software integration. Until the final chip is ready, software cannot be written for the system which slows down the development cycle and increases the time-to-market.

This makes RTL costly and has limited debugging capabilities. One solution to these limitations is raising the abstraction level and creating models which have less details compared to the RTL models. To address these limitations of RTL, SystemC transaction level models are used widely for SoC design and verification. The system functionality is represented using the concept of transaction that is, operations and interactions between the components by hiding the low level implementation details [26] [25].

SystemC [2] is a system level design and verification language built on top of C++. It allows modelling and execution of both hardware and software at different levels of abstraction. The high level of abstraction enables faster and more productive analysis, design, and redesign of architectural trade-offs compared to the RTL level. TLM [28] is a transaction based methodology approach and is based on C++ and SystemC. TLM provides an abstraction level in which the behavior of the functional blocks in the system is separated from the communication. The focus is on communication and it is performed by passing a high level data structure called transaction between the blocks through abstract channels or interfaces. Transaction level models use software function calls to model the communication between the blocks in a system in contrast to the RTL models where signals are used [11].

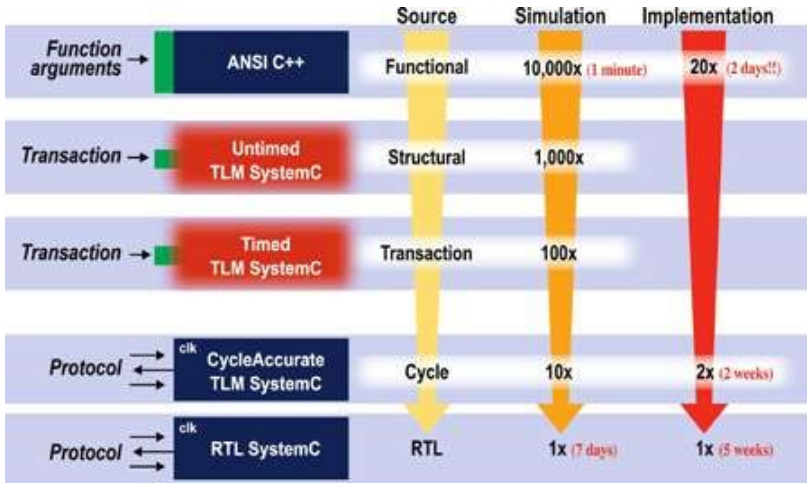

Figure 2.7: Implementation and simulation speeds at different levels of abstraction

TLMs have multiple abstraction levels from cycle accurate to un-timed models as shown in figure 2.7 . Initially, designers use higher level models with minimum details and these models can then be refined over time to include more information as the design cycle progresses.

Figure 2.7 also shows how the implementation and simulation speed increases with higher level of abstraction. The main concept in TLM is abstracting away the communication on the buses by using transactions. Instead of modelling all the bus writes and monitoring changes in their states, only logical operations like reading, writing, etc done by the buses are considered in the model. These abstractions increase the simulation speed by many orders of magnitude [1].

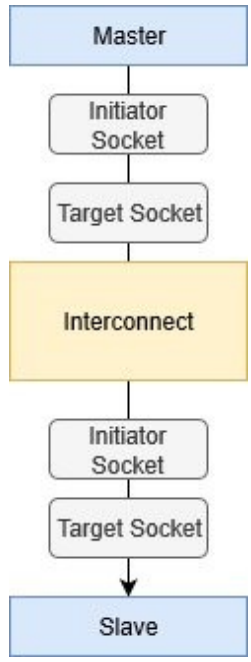

Figure 2.8: Representation of the flow of a TLM transaction from Initiator to Target

A transaction is an atomic exchange of data between the initiator or master and target or slave. The transactions are forwarded from the master to the slave. Example of a master is the CPU and a slave is Memory. The initiator initiates and issues the transaction through an initiator socket and a target is always ready to receive it through a target socket. The transactions are routed by the interconnects to their destination using the address. This corresponds to the classical concept in bus protocols [17]. The initiator communicates with the target using a transport interface and the target needs to implement the transport method. This is done by having the target register a callback method with the socket. This concept is represented in figure 2.8.

Some components have only initiator sockets, some have only target sockets, and some have both initiator and target sockets. The type of information which is exchanged via a transaction depends on the bus protocol being used. Generic payload is the class type used for the transaction objects which are passed through the interfaces and these objects have attributes that are typically found in memory-mapped bus protocols. Some are common to all protocols like:

- Type or command is the direction of data exchange, if it is read or write
- Address which determines the target and the register or component address
- Data that is sent and received
- Type of transfer like single word transfer or burst transfer
- Response status like success or failure

In RTL everything is synchronized using clocks and are synchronous in nature, whereas TLM models do not use clocks and are asynchronous in nature. In TLM, the synchronization happens when data is communicated between components. By modeling at this level early in the design cycle, designers can perform architectural exploration and find an optimal architecture before committing to the low-level details of a complete implementation. The TLMs can also be reused during functional verification to ensure that the design is equivalent to the RTL implementation [7] [10]. In conclusion, SystemC + TLM offers many advantages as mentioned above and is highly suitable for use in SoC modelling and architectural exploration thus making it the right choice to be used in this thesis.

2.3 RISC-V ISA

Custom SoCs are becoming ubiquitous and it is rare to find an electronics product which does not have an on-chip processor. The semiconductor industry today has been revolutionized by open source products from networking standards, to software to operating systems. Similarly, having an open ISA will enable greater innovation in processor architectures as a result of the free-market competition.

Many companies have patents on their ISAs which prevents others from using it without a license. The negotiations for obtaining a license can take a long time and it is also very expensive. This makes it very inaccessible to the academia and research sector which could have greatly contributed to the improvement of the ISA. On the other hand, a shared open core design translates to faster innovation, shorter time to market, low cost from reuse, transparency and processors becoming more affordable for smaller devices.

RISC-V was developed with the goal of creating a universal instruction set which is open and free to all users [4]. RISC-V [21] is a royalty-free and open-source general-purpose ISA used for designing processor architectures which builds and improves upon the original RISC architectures. It has a common base set ISA and a toolchain that can handle both the base ISA and customized instructions defined by a SoC architect. Some of its key advantages are [30]:

1. Enabling SoC architects in customizing processor architecture is one of its biggest advantages which differentiates it from the other ISAs in the market. Specific application issues can be solved by adding various customizations like hardware accelerators, custom instructions, different cache sizes all without breaking compatibility and causing fragmentation.

2. The base ISA is very simple and modular. The instruction coding is very regular and does not have complicated memory instructions. This simplifies the implementation and keeps it architecture clean as a result of which RISC-V cores are smaller than ARM and x86 cores [22].

3. As the ISA is open, the designs can be optimized for different scenarios like low power, performance, security, etc. It provides higher control over the hardware implementation and fewer compromises.

4. It is a frozen ISA which means that the base instructions are frozen and optional extensions when added are also frozen. This leads to stability of the ISA and provides solid foundation to preserve the software investments. Software development can be done more confidently because the software written for RISC-V will run on all similar cores of RISC-V forever.

Though the ISA is incomplete and its ecosystem is in its early stages of development, it is very promising. On one hand, its structure of a small base ISA makes it suitable for research and education while also making it capable of being a suitable ISA for inexpensive and low-power embedded devices. On the other hand, the option of adding a variety of extensions allows it to form a powerful ISA which could be used for general-purpose and high-performance computing.

Chapter 3

System Setup

This chapter describes in detail how the SoC simulator used in this thesis was set up and a simple graphics test was run on it. A RISCV-ISA based SoC simulator has been used in this thesis which has been structured as per our requirements. This simulator is written using SystemC + TLM. A base RISCV-ISA simulator written in SystemC + TLM provides important components needed for our simulator. The description of this base simulator is provided in one of the sections. How our simulator is customized and structured is also described in the next section followed by simple testing to check its sanity.

The next section of this chapter is emulating a graphics subsystem in our simulator. To emulate a simple graphics subsystem, the GUI code is written using a graphics library and run on the CPU model, a frame buffer is prepared in the RAM model and a display controller gets the data from the frame buffer and outputs it to a display unit. The graphics library used for writing the GUI code is LVGL. It is set up in our simulator by first running it in its own simulator environment, then compiling it using native GCC of the PC (x86) and then finally compiling it using GCC for RISC-V. The display is emulated by using a library which prints the data in the frame buffer to a bmp file and a simple display controller is modelled in SystemC + TLM.

3.1 Base Simulator

The RISCV-ISA simulator [18] which was used as the starting point for setting up the system was found on GitHub and is licensed under the GNU General Public License [8] giving permission for private use and modification. This simulator is coded in SystemC + TLM thus making it suitable for this thesis as discussed in the background chapter. The structure of the simulator is explained in the figure 3.1.

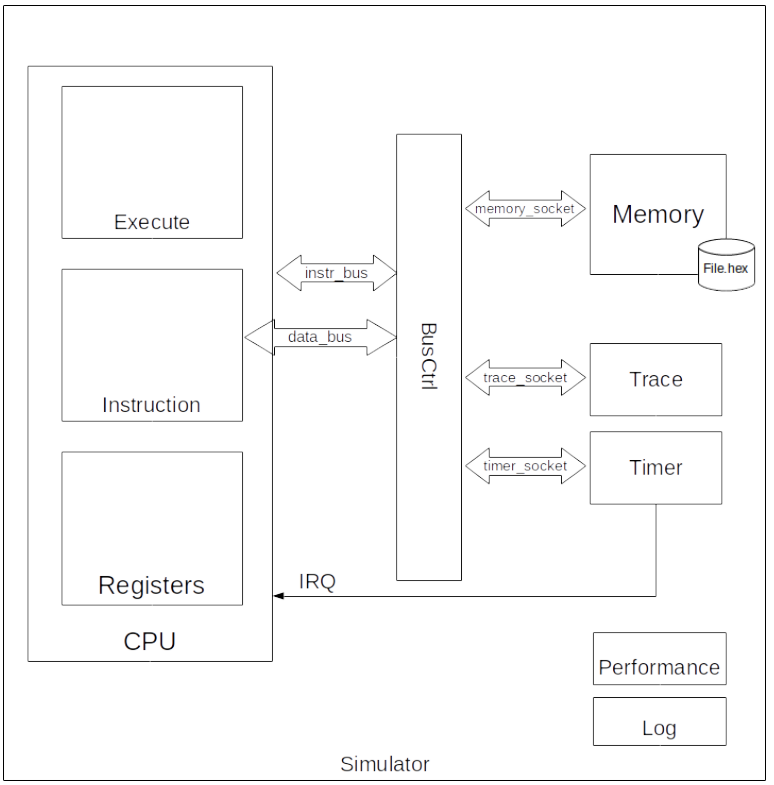

Figure 3.1: Structure of the base RISCV-ISA Simulator [18]

It is to be noted that this is how the components are connected in the simulator and in our model, this configuration of connection is not used as is. This configuration is used as an example and our simulator is connected along similar lines. Also not all the components in this simulator are used in our model and only the necessary ones are picked out, modified if needed and connected.

CPU is a top-level initiator model and encompasses the Registers, Instruction, and Execute models. The Registers model implements register files, PC register, and CSR registers. The software to be run on the CPU is stored in the Memory (loaded as a hexfile) and it has read and write capability. The Instruction model fetches the instruction to be decoded through the instruction bus (instr_bus) and decodes it. If the instruction requires any data to be read from or written to the Memory, it is fetched using the data_bus and the Execute model executes it.

The CPU model is an ISA based processor and it has capability to decode and execute three kinds of instructions:

- Compressed instructions having a C extension
- Multiplication and Division instructions having a M extension
- Atomic instructions having an A extension

BusCtrl is an interconnect model and a bus manager. It has target sockets for connecting initiator models like the CPU and initiator models for connecting target models like the Memory, Trace, and Timer. It only forwards transactions to the correct target without modifying the transactions. Trace is a simple trace peripheral which creates a xterm window for printing out the received data and timer is a simple real-time IRQ programmable counter peripheral. There are also some helper classes like the Performance model used for storing the performance indicators of simulation and Log class for logging.

All of these components are connected together as shown in the figure 3.1 in the Simulator Top file. This simulator was studied to get a basic understanding of modelling a SoC in SystemC + TLM. The simulator package also comes with many tests which can be run to test various criteria and it provided a good starting point to understand how the tests are written and run on the simulator.

3.2 Customized simulator

The simulator which is used throughout this thesis is connected as shown in figure 3.2 to start with. A few modifications are made to it as the thesis progresses which are described when they are done. It derives its components from the base simulator described in the previous section, in-house Nordic Semiconductor models and some models developed specially for this thesis. The simulator is structured to model a generic, heterogeneous, and multi-core system with shared memory and I/O.

Our simulator has 3 domains namely the Shared domain, App domain, and App1 domain. All the domains have their own Top file in which the component models are instantiated and connected. The App domain is the one which will be used for running the application code on. The shared domain acts like a global domain with shared memory and I/O, and all the other domains are connected to it. The App1 domain is not used in this thesis. It is connected to show that the simulator is easily configurable and extendable where new components and domains can be added with ease.

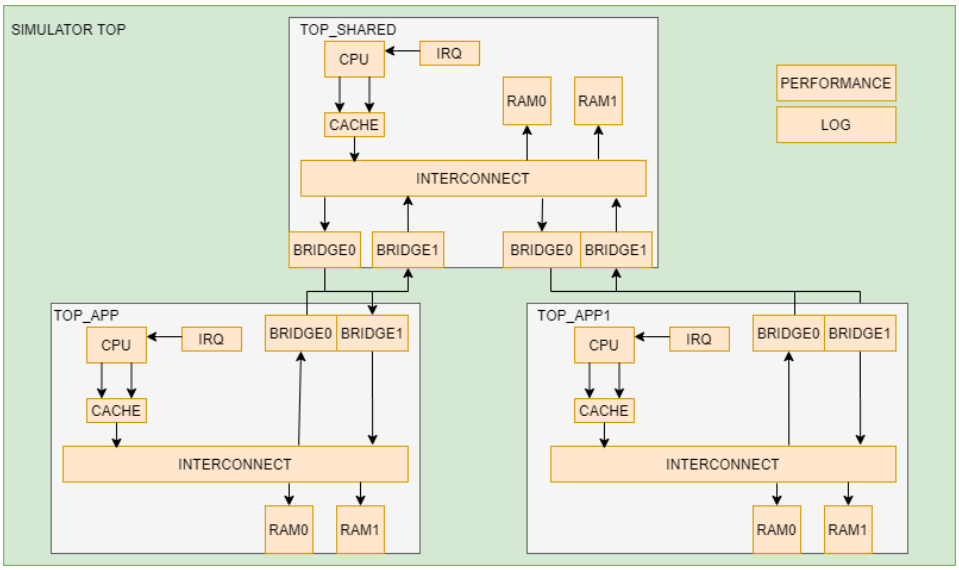

Figure 3.2: Structure of the Customized Simulator used in this thesis

In the Simulator Top file, the three domains are instantiated and connected. The CPU model and the Memory model (renamed here as Ram) are taken from the base simulator along with the helper models Performance and Log. The IRQ model, the Interconnect model, and the Bridge models have been developed in-house.

Each domain has a CPU with a Cache and IRQ. The CPU is a master or initiator model connected to the Interconnect through its Cache. The Memory model is renamed as the Ram model and used here. Two instances of Ram are connected in each domain where instance 0 is the ROM and instance 1 is the RAM. The Ram instances are connected as slaves or targets to the Interconnect. The Interconnect is a bus manager to which any number of master and slave components can be connected and it forwards the transaction without modifying it. It is similar to the BusCtrl model in the base simulator, but is designed differently.

The transactions are initiated in the CPU model and forwarded either to the Ram model or to the Bridge through the Interconnect depending on the target address. Bridge0 and Bridge1 are Bridge models having one initiator socket and one target socket. They are used for forwarding transactions out of the domain (Bridge0 connected as a target to the Interconnect) or for receiving transactions forwarded from other domains (Bridge1 connected as an initiator to the Interconnect). The App domain is the Application domain and used for running majority of the applications. The App1 is a replica of the Application domain. The Shared domain is connected to both App and App1 domains. Programs can be run on all the three domains. This functionality is tested by running a simple print test on all the CPUs.

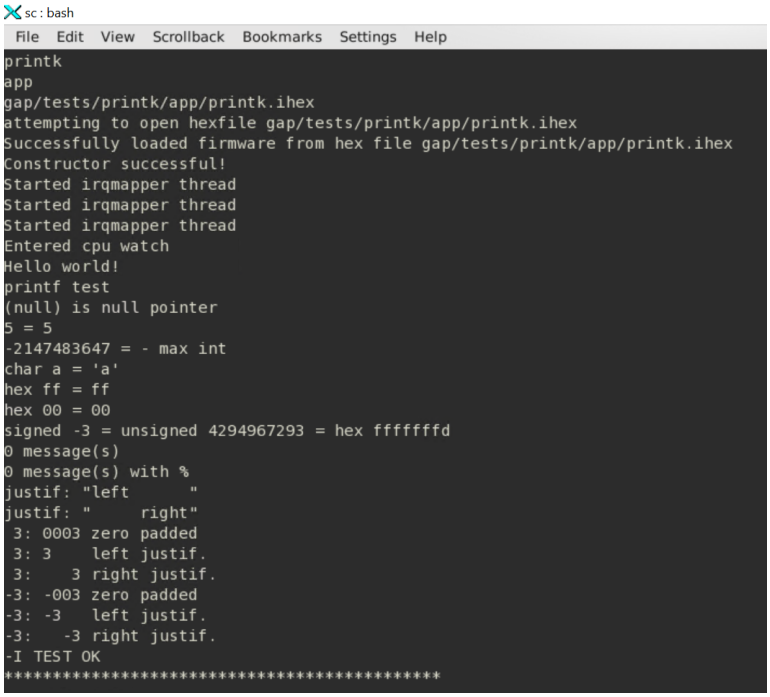

Figure 3.3: Result of the print test being run on the App domain CPU

The program ran successfully on all the three CPUs and the snapshot of the the test running on the App domain is shown in figure 3.3. The first line in the screenshot is the name of the test running and the second line is the name of the domain in which it is running.

3.3 Graphics subsystem on the simulator

The graphics subsystem in a MCU and its components were described in detail in the background chapter. This section describes how it will be set up in our SoC simulator. First, we need a software graphics library which is used for writing the GUI code by making calls to its API. The graphics library chosen to be used in this thesis is the LVGL library and it is described in the background chapter. Understanding the library is the first part in setting it up to run our simulator which is done by running LVGL in its own simulator environment. Next, LVGL was ported to run on the native GCC of the PC (x86) to understand the process of porting. The last part is emulating the entire graphics subsystem on the simulator and porting LVGL to run on our simulator. All these are explained in following sub-sections.

3.3.1 Running LVGL on its simulator environment

LVGL has the feature of running in its own simulator environment without the need for any development board [24]. This is a very useful feature as it allows one to write and experiment with real LVGL applications. Other advantages of having this feature are it makes the LVGL code hardware independent, cross-platform compatible, and portable.

The PC simulator was set up on Windows using Visual Studio. Everything was set up and only the main file had to be run by uncommenting the test to be run which can be chosen. There are a variety of tests to check the working of different LVGL objects, LVGL themes, LVGL fonts, etc. The simulator uses SDL [23] which is a cross platform libray for simulating the display and the input.

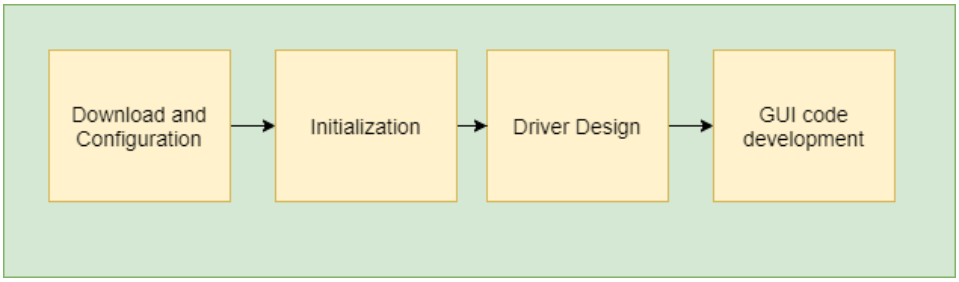

Figure 3.4: Steps to port LVGL for use in a project

The entire method for using LVGL was studied, the figure 3.4 illustrates the steps.

1. The first step is to download or clone its Github repository [13]. It should be copied to the project directory and then in the configuration file, only the modules and functions which will be used are enabled. Only enabling the parts of the library which will be used helps in keeping a small memory footprint of the library.
2. Next step is to initialize the library and its components.
3. Drivers make calls to functions that drive the display/input/file systems and also to hardware accelerators defined for specific functions. The drivers which are used must be designed, templates for which are provided in the library and they have to be modified according to the system being used. The three primary steps in the design of drivers is the initialization of the driver, definition of the driver, and finally registration of the driver. Drivers have two fields namely data fields and callback functions which must be defined.
4. Next part is the GUI code development. The LVGL repository has lot of examples for using different kinds of objects, themes, and applications. The examples can be used directly or new code can be written as required. The code is written by creating different kinds of objects like lists, widgets, images and defining their attributes like position, size, color, style, etc. The GUI functions are called and the final part is calling the task handler of the library periodically by using a timer interrupt to handle the tasks.

3.3.2 Porting LVGL to native GCC of the PC (x86)

Using the knowledge obtained by running the library in its simulator environment, LVGL was set up on native GCC of the PC in this part. In the simulator, SDL is used for simulating the display and input layers. The linux environment which was used for doing the simulations in this thesis did not have access to SDL. Assuming that the GUI we are making is static is nature, an input device would not be required. The display must still be present so that the GUI is output somewhere. Therefore, as an alternative to SDL for technical reasons, the display was emulated by writing the image which is in the frame buffer to a BMP (bitmap) image file. To do this a library called QDBMP [14] which stands for ”Quick N Dirty BMP library” was used. This is a minimalistic C library which is used for handling BMP image files. The QDBMP header and C files are added to the work folder.

The LVGL repository was cloned to the work folder and a simple makefile was written for compiling the various source code files in the library. A main file was written where initialization was done and a simple GUI function for creating some objects of different shapes and colors was defined and called. A display driver is written which has the data field as the frame buffer and a callback function to flush the contents of the frame buffer to the QDBMP file.

The frame buffer is screen sized meaning it is the size of the display screen and equal to resolution to the screen. The LVGL code is run on the CPU model which writes pixel data to the internal frame buffer and after it has finished preparing the frame, the display driver callback function writes it a BMP image file pixel by pixel by using functions from the QDBMP library.

3.3.3 Porting LVGL to run on the simulator

A graphics subsystem in a MCU as seen in the figure 2.1 is made up of - CPU, software, frame buffer, display controller, and display panel. The figure 3.5 illustrates how the entire graphics subsystem was emulated on our simulator. The GUI application code was written using the LVGL library. The same procedure like in porting LVGL to native GCC of the PC was followed. The application code is common for all the domains. We can choose which CPU the application code runs on by giving the test name and the domain name as command line arguments.

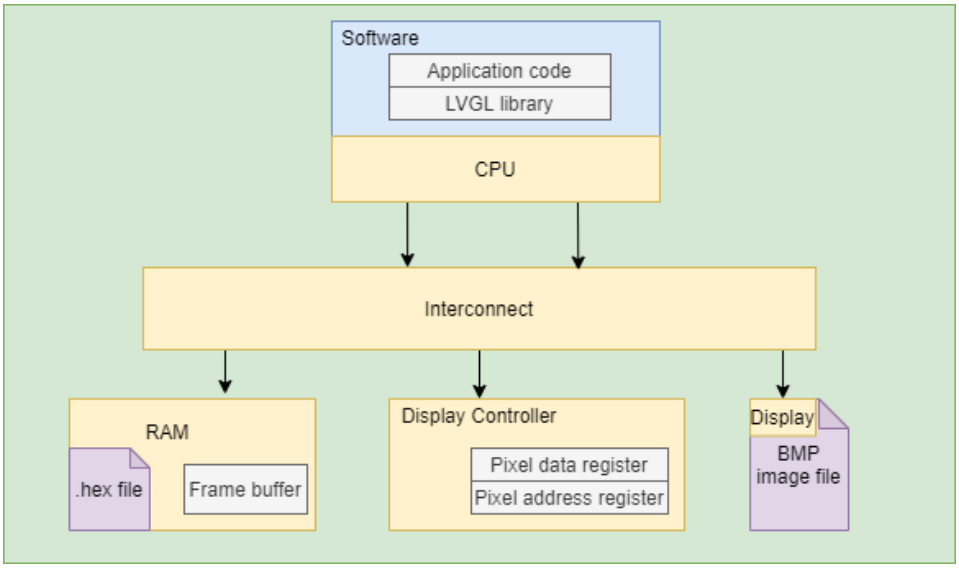

Figure 3.5: Graphics subsystem on the SystemC + TLM simulator

Instead of compiling it on native GCC of the PC(x86), to run it on our simulator it is compiled on GCC for RISC-V. It is compiled to the hex format, loaded to the RAM model and run on the CPU model. When the application is run on the CPU, it prepares the frame in the frame buffer which is stored in the RAM model . The CPU decodes the instructions and if the instruction requires read or write it has the capability to perform it.

Once the entire GUI has been prepared in the frame buffer, the callback function in the display driver to flush the contents of the buffer to the display is called. The CPU then transfers the data from the frame buffer to the display controller by writing the RGB data and 2D coordinates of the pixels one by one to the registers of the display controller. The baseline display controller is modelled using SystemC + TLM and connected to the simulator as shown in figure 3.5. It has one target socket which receives the pixel data in the form of transactions. It is connected as a target to the interconnect model which is a bus manager model for forwarding the transactions by address decoding. It has two registers one for the pixel address and one for the pixel data which can be read or written into. The pixel address register stores the 2D coordinates of the pixel and pixel data register stores the RGB color data of the pixel. The controller model implements a transport function. When the controller model sees that data has been written to both its registers, it uses the QDBMP functions to place the pixel in the BMP image file which emulates a display panel.

Chapter 4

Benchmarking

The simulator has both hardware and software parts. The simulator is made up of various hardware components modelled in SystemC + TLM like CPU, RAM, Bridge, etc. The software which runs on the simulator is the application code written using LVGL. Every operation takes some amount of cycles to complete be it a simple line drawing operation done by LVGL or a transaction moving from the CPU to the RAM through a interconnect.

The first section is explaining the timing concepts in TLM models followed by a section on setting up a suitable concept of timing in the hardware models of the simulator. The third section is setting up a benchmarking framework for the application code running on the simulator. The next section is the results section where the application code is run on the simulator for drawing different scenes on the display and the total cycles taken by the CPU is broken down to cycles taken for individual operations. The final section is the discussion section which analyzes and discusses the results presented in the previous section. Potential areas for improvement are identified by doing baseline performance analysis.

This chapter is integral as it helps us to understand which operations hog most cycles and gives a concept of time to the simulator as a whole. Otherwise, the simulation would complete in an infinitesimally small amount of time and would not mirror real-life situations well. It also helps to establish a baseline which can be compared with when architectural improvements are done on the simulator in the next chapter.

4.1 Timing concept in TLM models

Understanding the timing concepts and styles in TLM models is important in order to set up timing on our simulator. The types of timing styles in TLM models are briefly described first and the timing style which is used in our simulator is explained in detail.

4.1.1 Types of timing styles in TLM models

1. **Loosely-timed:** The timing is provided at the level of the individual transaction. It makes use of the blocking transport interface. This interface has only two timing points - the initiators call to b_transport which carries the transaction request (beginning of the request) and the return from the target which carries the response (beginning of the response). This style supports the temporal decoupling concept in which the individual SystemC processes are allowed to run ahead in their own local time wrap without advancing the actual simulation time until they reach a time when they should synchronize with the rest of the system. This style is ideal for use in our simulator since it uses (b_transport) and is explained in detail in the next subsection.
2. **Approximately-timed:** This is supported by the non-blocking transport interface which provides timing annotation for multiple phases and points during the life of a transaction. The transaction is broken down into many phases and this is used when working with particular hardware protocols. This style is detailed and slow to simulate and is not ideal for use in our simulator.
3. **Untimed style:** TLM has no explicit support for this as they have no value since all models need a concept of time. Loosely timed models with zero timing annotation can be used as untimed models. Here, b_transport is only used to send the data to the target and does not carry any information about response times [29].

4.1.2 Loosely-timed coding style and temporal decoupling

In loosely-timed style, the initiator communicates with the target using a blocking transport interface, the target implements the transport method and registers it as a callback with its target socket so that when the initiator calls the method it receives it. The initiator initiates transactions and forwards to the interconnect which routes the transaction to the correct target depending on the address embedded in the transaction.

Transaction argument is passed through reference using the b_transport function and has no return value. The target receives and responds to the transaction. It can perform some actions to modify the attributes of the transaction and finally it returns the transaction response status to the initiator. The b_transport of the target executes in the context of a thread process in initiator module and when it returns, the control is unwound through the call chain back to the initiator [29].

Another argument which is passed through reference using the b_transport function is the timing annotation which is the local time offset. The timing annotation is active on both the call to and the return from the transport method. Since it is sent as a reference, this means that the receiving function in target can directly modify it and that change is reflected on the sender side.

sc_time_stamp returns the current simulation time of the system. The recipient of a transaction is required to behave as if it had received the transaction at **effective_local_time = sc_time_stamp() + local_time_offset**. There are two ways in which synchronization of the local_time_offset with the system simulation time returned by sc_time_stamp() can be achieved.

Explicit synchronization

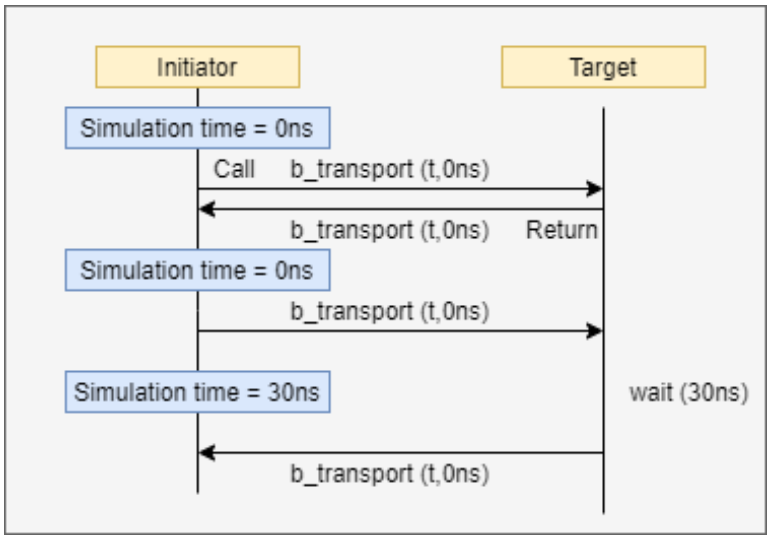

Figure 4.1: Blocking transport synchronized explicitly

In explicit synchronization, wait can be called explicitly on both the initiator and the target side. Untimed models can easily be implemented by setting the timing parameter in the transport calls to zero as shown in figure 4.1 where the transport method returns immediately. Wait can be called explicitly on the target side to represent the response time of the target. The wait models the time taken by the target to process the transaction and it waits for this time to return. We can observe that the simulation time is advanced when the wait is called and on return from the target, the time parameter is reset to 0.

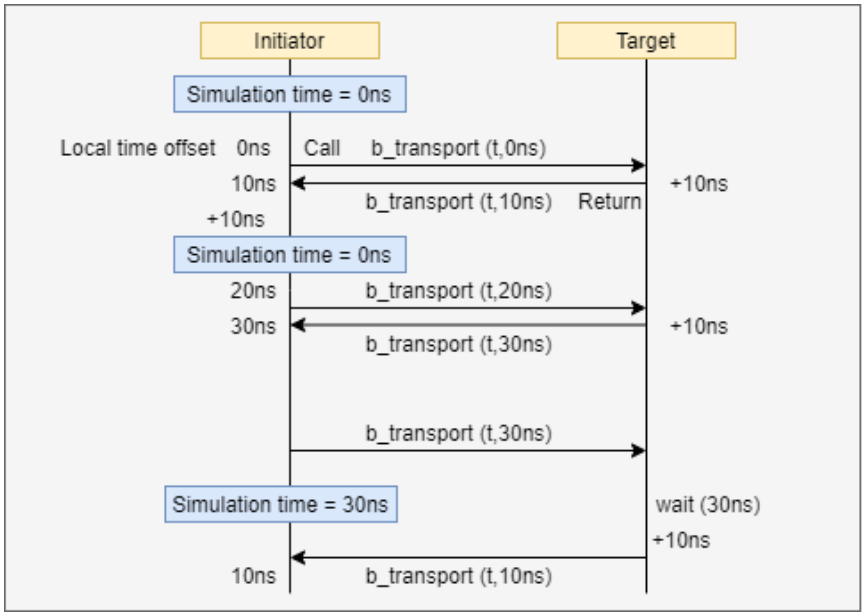

Figure 4.2: Blocking transport with temporal decoupling synchronized explicitly

Temporal decoupling is the running of the initiator thread ahead of the simulation time as shown in figure 4.2. The transport method passes a non-zero value for the time argument. The initiator and the target can each increase the value of the time argument to further advance the local time offset. The time argument is returned untouched to the initiator from the target if it is not incremented on the target side. Adding the time returned by the call to the simulation time can give the time at which the transaction completes, but the simulation time itself does not advance. For the time argument to be added to the simulation time, wait must be called either on the target or initiator side. After the wait is called, the local time offset must be reset to zero. A disadvantage of using temporal decoupling like this is that an initiator thread can hog the processing time indefinitely until wait is explicitly called [5].

Implicit synchronization

Loosely-timed models can also progress in the absence of explicit synchronization points. In SystemC a single synchronized view of time is present which is used by all the threads. When time quantum is used, the initiator can only run ahead as far as the end of the quantum before implicitly calling wait to synchronize with the SystemC simulation time. This is called temporal decoupling and it allows each initiator thread to keep its own local view of time and it can run ahead of the simulation time until it has to synchronize with the other threads. This concept is illustrated in figure 4.3 [6].

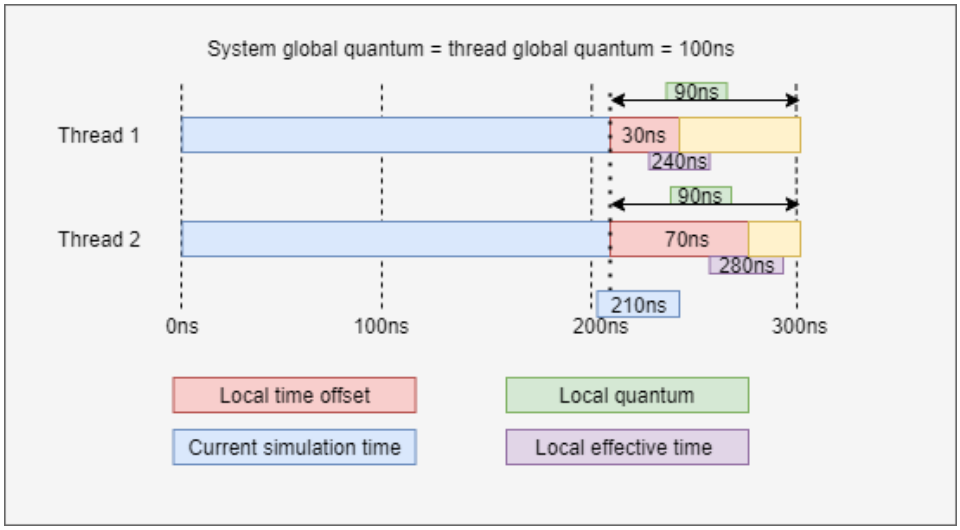

Figure 4.3: Illustration of temporal decoupling concept with time quantum

The system global quantum time is the time unit on which all the threads synchronize. Here, the system global quantum is taken as 100ns, so all the threads synchronize on 100ns, 200ns, 300ns and so on. The thread global quantum is the time unit on which a particular thread synchronizes. Both the system global quantum and the thread global quantum are taken to be 100ns. The current system time stamp is assumed to be 210ns. Both the threads can advance from the system time stamp and the time by which they advance is called the local time offset which is 30ns for thread 1 and 70ns for thread 2. The time remaining for both the threads until the end of the quantum as relative to the current system time is called local quantum and is equal to 90ns.The local effective time which is the sum of the current system time and the local time offset of the thread is 240ns for thread 1 and 280ns for thread 2.

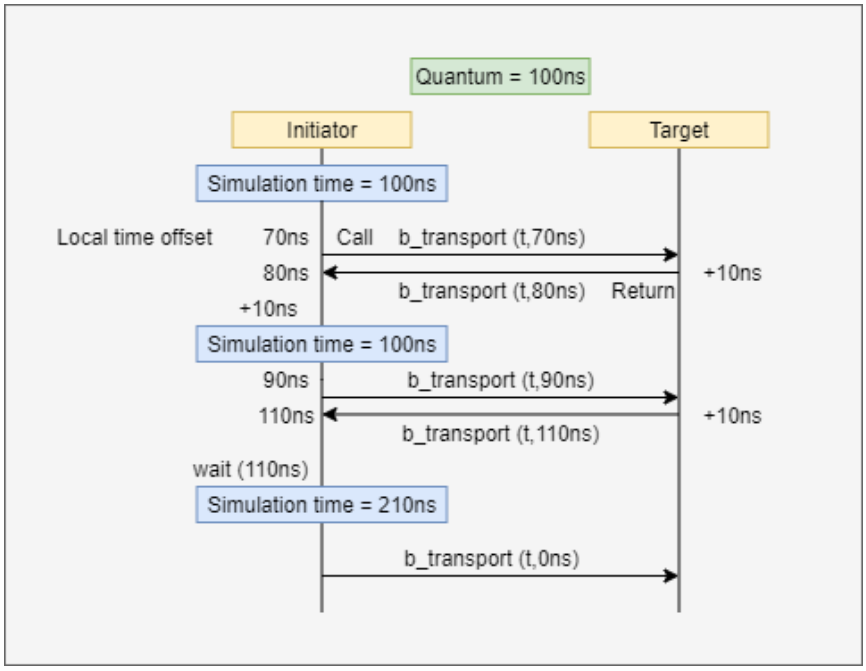

Figure 4.4: Blocking transport with temporal decoupling synchronized implicitly

Synchronization happens implicitly when temporal decoupling is used with time quantum as shown in the figure 4.4. A quantum is the greatest amount of time that a thread can differ from the system simulation time. Once, the local time of the thread exceeds the quantum, the wait is called implicitly in the thread to synchronize with the system time and the local time is reset to zero.

The temporally decoupled initiator advances its local time offset until the time quantum is exceeded. This happens when the transport method returns from the target with a local time offset of 110ns which exceeds the quantum 100ns. So, wait is called implicitly in the initiator thread for the time unit 110ns and the simulation time advances to 210ns. We can also observe that when wait is called, the local time offset is reset to 0ns.

4.2 Timing model implemented on the simulator

The loosely-timed style with temporal decoupling and time quantum is implemented in our simulator. The advantages of this is:

- It allows multiple system initiators to progress ahead of the system time and they synchronize due to the presence of the time quantum.
- The performance of loosely-timed models with blocking interfaces is improved and bottlenecks in processing are avoided. It ensures that a thread does not hog all the processing time and synchronizes with the simulation time regularly.
- Easy to implement. Latency is given as an argument in the constructor while initialising new objects of a target and when the transaction gets routed to the target, this latency is added to the delay sent from the initiator and gets reflected in the total simulation time when it is synchronized each quantum.

The methodology for setting up the timing model involves two parts - set up in the initiator and set up in the target. Both are described using code snippets compiled from various files. The snippets are not complete and are complied from various files to show the general methodology.

Set up in initiator

```
1  /* In header file */
2  #include "tlm_utils/tlm_quantumkeeper.h"
3  tlm_utils::tlm_quantumkeeper qk; // Declaring a time keeping
       thread
4
5  /* In constructor */
6  qk.set_global_quantum(sc_time(1, SC_US)); // Update the global
       quantum
7  qk.reset(); // Reset the local time offset to 0
8
9  /* Function initiating transactions to targets */
10 void CPU::single_step(int benchmark)
11 {
12     sc_time delay;
13
14     delay = qk.get_local_time(); // Returns the current local time
        offset
15     instr_bus->b_transport(*trans, delay); // Annotate b_transport
        with local time
16     qk.set(delay); // Update qk with time consumed by target
17
```

```
18      // qk.inc( sc_time(100, SC_NS) ); // Further time consumed by
        initiator
19      if (qk.need_sync()) // Check local time offset against quantum
20      {
21          qk.sync(); // Updation of the global time sc_time_stamp
22      }
```

Listing 4.1: Code snippet showing setting up timing in an initiator

The code listing 4.1 above shows how timing is set up in the CPU initiator, the same methodology is followed in all the initiators. In this example, the timing thread is declared as qk in the header file of the CPU model. The global quantum is updated to the required value, taken to be 1us and the local time offset is zeroed by calling the reset function in the constructor function of the CPU called when a new object of CPU is initialized.

In the function which initiates transactions to the target, the SystemC time variable, delay, is updated with the local time offset of the CPU thread using the get_local_time method and then the blocking transport function of the target is called with the transaction and timing arguments. The timing annotation of the b_transport function is active on both the call to and the return from the transport method. The timing argument can be updated in the target to indicate its response time and on return, using the set method, qk is updated with the time consumed in the target.

There is also an option to further increase the time consumed by the initiator by using the inc method. On return from every transaction, the local time offset of the thread checked against the quantum using the need_sync method. If it is equal to or greater than the quantum, wait is implicitly called and the system simulation time is updated using the sync method.

Set up in target

```
1  /* In header file */
2  const sc_time LATENCY;
3
4  /* In constructor */
5  Ram::Ram(sc_module_name name, sc_time latency)
6      : sc_module(name), socket("socket"), LATENCY(latency)
7  {
8      socket.register_b_transport(this, &Ram::b_transport); //
       Registering callback for incoming interface method call
9  }
10
11 /* Function receiving the transactions from the initiator*/
12 void Ram::b_transport(tlm::tlm_generic_payload & trans, sc_time &
       delay)
```

```
13 {
14     delay += LATENCY; // Updating the delay with the latency of
    the target
```

Listing 4.2: Code snippet showing setting up timing in a target

The code listing 4.2 above shows how timing is set up in the RAM target, and the same methodology is followed in all the targets. Latency is given in the constructor as an argument while initialising new objects of the target. In the b_transport method of the target, which is called when it receives transactions from the initiator, this latency is added to the local time offset of the initiator thread calling the target.

It was observed that having a small quantum helps keep the system simulation time more accurate by frequent synchronization but at the same time adds an overhead on the system. This is because when the quantum is small, frequent calls are made to the wait function to synchronize which causes the control to be switched back to the SystemC simulation kernel. This context switch can be expensive in terms of simulation performance. On the other side, having a large quantum reduces the overhead as the synchronization is not as frequent, but the system simulation time might not be as accurate. If the quantum is big, code can execute full speed for a long time without having to stop frequently for SystemC kernel context switch. All the initiators in the simulator have a global quantum of 1us which can be changed and the latency of the targets is configurable.

4.3 Benchmarking framework for the application code

The software running on the simulator is written using the LVGL GUI library. The GUI is created using the 30+ widgets present in the library which can be customized and drawn on the screen using the various drawing functions in the library. The LVGL repository has programs to test the drawing of its widgets. 5 widgets were arbitrarily chosen and tested. The total cycles taken to produce these widgets on the display was calculated. The total cycles is the sum of the CPU executing 1 instruction per clock cycle and the cycles consumed by waiting for the read/write transactions to the bus/memory. The total cycles consists of 3 components:

1. Cycles taken to render the frame, that is draw the frame in the frame buffer
2. Cycles taken to flush the contents of the frame from the frame buffer to the display via the display controller
3. Cycles taken to do miscellaneous operations other than the drawing and flushing operations, cycles taken due to the waits in the components, etc

All these 3 components were calculated to understand the breakdown of the total cycles spent in producing the widget on the screen. Out of these three, the cycles taken to render the frame was studied in detail. The aim here is to benchmark the cycles taken to perform various operations by the application code on our simulator, so that the operations which hog the most cycles can be found.

The methodology for setting up a benchmarking framework in the application code is described using the example of one drawing function - drawing an arc. LVGL has various drawing functions, each of these are benchmarked by adding a csr_write instruction at the entry and exit of the function. CSR is a register in the CPU for storing additional information. A unique value is written to this register both when the function is entered and when it is exited.

```
1 void lv_draw_arc(lv_coord_t center_x, lv_coord_t center_y,
    uint16_t radius, const lv_area_t * mask,
2                 uint16_t start_angle, uint16_t end_angle, const
    lv_style_t * style, lv_opa_t opa_scale)
3 {
4     uint32_t value = 0x0009;
5     csr_write(CSR_MCYCLE, value);
```

Listing 4.3: Code snippet showing adding a csr instruction in an LVGL draw function

The code snippet 4.3 shows a value of 0x0009 written to the CSR when the arc drawing function is entered and a value of 0x0010 is written when the function is

exited (not shown here in the snippet). There are many CSR registers in the CPU and one register having the address CSR_MCYCLE is used for benchmarking all the drawing functions. Different values are written to this register for identifying if it is the start/end of which function.

Next, the CPU model is modified so that it reacts to these csr_writes. We can see how the timing implemented in the simulator and explained in the previous section is used in this snippet. qk is the timing thread which holds the timing information and the function get_current_time returns the sum of the SystemC simulation time and the local time offset thereby giving the effective local time for accuracy. The cycle time is the time taken by the CPU to process one instruction and it is assumed to be 10ns in our simulator.

```
1  uint32_t csr_temp;
2  sc_time cycle_time(10, SC_NS);
3
4  csr_temp = register_bank->getCSR(CSR_MSTATUS); // Read the CSR
     register
5
6  else if (csr_temp == 0x0009)  {
7    draw_arc_num+=1;
8    draw_arc_start = qk.get_current_time()/cycle_time;
9    if(lvgl_print) cout << "START arc" << endl;
10   register_bank->setCSR(CSR_MSTATUS, 0x0); //Reset the CSR
     register
11 }
12 else if (csr_temp == 0x0010) {
13   draw_arc_total = qk.get_current_time()/cycle_time -
     draw_arc_start;
14   draw_arc_sumtotal+= draw_arc_total;
15   if(lvgl_print) cout << "CYCLES arc:" << dec << (int)
     draw_arc_total << endl;
16   register_bank->setCSR(CSR_MSTATUS, 0x0); //Reset the CSR
     register
17 }
```

Listing 4.4: Code snippet showing the CPU model reacting to the CSR write instruction

The code snippet 4.4 shows how the CSR register is read and if it contains the value 0x0009, it signifies the start of the arc draw and the current CPU cycle is calculated by dividing the current effective time by the cycle time. When it reads the value 0x0010 signifying the end of the arc draw, the total cycles spent in this drawing function is calculated. In both the cases, a message is printed to the log file at the entry and exit of the function along with clearing the CSR register.

These prints written in a text file are input to a python script which parses this data and produces a compact data set which contains information about the hierarchy of functions called, that is which drawing function called which other drawing function along with the total times it was called and total cycles spent.

```
[{'rect': {'cycles': 1062557,
           'draw_fill': {'cycles': 1062196, 'num': 1},
           'num': 1}},
 {'arc': {'atan2': {'cycles': 14964, 'num': 175},
          'cycles': 213236,
          'draw_fill': {'cycles': 44780, 'num': 109},
          'draw_px': {'cycles': 55140, 'num': 286},
          'num': 1,
          'sqrt': {'cycles': 27994, 'num': 124}}},
 {'rect': {'cycles': 120, 'num': 1}},
 {'rect': {'cycles': 12, 'num': 1}}]
```

Figure 4.5: Output by the python parser script when the arc widget is drawn

The figure 4.5 shows the raw data produced when an arc widget is drawn on the display. From the figure 4.5, we see that for drawing an arc widget, rectangle and arc drawing functions are called. The rectangle draw function further calls draw fill and the arc draw function calls draw fill, draw pixel, sqrt, and atan2 functions. In 2 instances, the rectangle draw does not call any other draw function. In case of the arc draw functions, it means that the draw fill operation was called by it 109 times and that took 44780 cycles to complete, draw pixel was called 286 times and it took 55140 cycles to complete and so on. The arc draw overall took 213236 cycles to complete including the cycles spent in the other draw and math functions called by it.

This compact data set which is produced by the python parser script is used to draw the relevant diagrams, tables and graphs to make better sense of it. These illustrations are presented in the next section.

4.4 Results

In case of each widget, the image of the widget which is produced on the display is presented with a numerical breakdown of the total cycles taken to produce the widget on the display. It is made of three components, cycles taken to render the frame in the frame buffer, flush the frame from the buffer to the display via the display controller and other miscellaneous operations which do not fall into the other two categories. Next, this numerical breakdown is represented in a graphical format in the form of a pie chart to represent the % of the total cycles taken by each of the 3 components mentioned above.

The cycles taken to render the frame in the buffer is further broken down to the hierarchy of drawing operations in a table. It is split into various categories of drawing operations and the table seeks to explain which lower order operations are called by which higher order operations. The different orders of drawing operations can be easily distinguished using the colors. In every order, the cycles spent in it is the sum of the cycles spent in the lower order functions called by it. The cycles and the % of total cycles spent in each of the drawing operation is also shown. Miscellaneous means the difference of the total cycles taken to render and the total cycles spent in drawing operations. Residual means the cycles spent in a drawing operation but not in the drawing operations called by it, it is the difference between the cycles spent in a drawing operation and the cycles spent in the drawing operations called by it.

In all the widgets, the frame buffer is stored in the local RAM and the size of the frame buffer is equal to the size or resolution of the display. The resolution of the display is taken as 480 * 320 where 480 is the horizontal resolution and 320 is the vertical resolution. The color depth is taken as 32 bits meaning 4 bytes are needed to store each pixel. The total cycles is equal to the sum of the delays of every instruction cycle and 1 cycle latency when the local RAM is accessed each time. Each cycle is taken to be 10ns.

Widget 1 - Image

Figure 4.6: Image of the Image widget produced on the display

Operation	Number of cycles
Render frame	1205967
Flush frame	5073470
Miscellaneous	1911918
Total	8191355

Table 4.1: Numerical breakdown of cycles taken to produce the Image widget

The first widget which was drawn was the image widget as shown in figure 4.6. The table 4.1 shows the numerical breakdown of the total cycles taken to produce the image widget on the display.

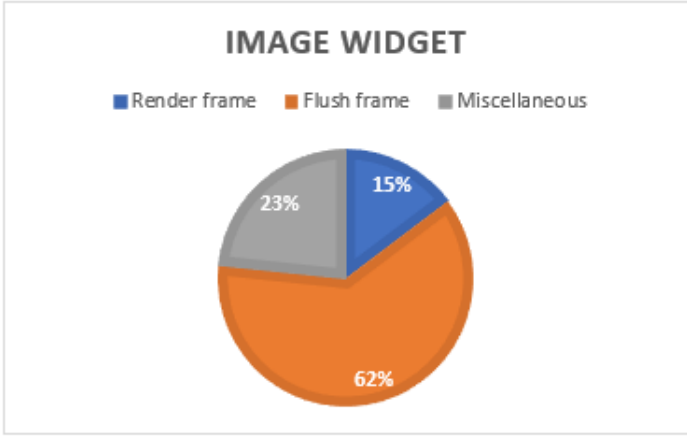

Figure 4.7: Graphical breakdown of cycles taken to produce the Image widget

This numerical breakdown is represented in a graphical format as shown in figure 4.7 to denote the % of the total cycles taken by each of the 3 components. It can be observed that flushing the frame to the display takes the highest number of cycles at 62% of the total cycles and the frame rendering takes the least number of cycles at 15%.

Drawing operation		Number of times	Number of cycles	% of total cycles
Advanced			1165792	96.67
Rectangle		3	1062688	88.12
	Basic		1062194	88.08
	Draw fill	1	1062194	88.08
	Residual		494	0.04
Image		1	103104	8.55
	Basic		98926	8.20
	Draw map	1	98926	8.20
	Residual		4178	0.35
Miscellaneous			40175	3.33
TOTAL			1205967	100.00

Table 4.2: Breakdown of drawing operations in rendering the frame - Image widget

The cycles taken to render the frame in the buffer is further broken down to the hierarchy of drawing operations called to render the image widget as shown in table 4.2. It can be seen that to draw the widget as shown in the figure 4.6, 2 advanced drawing operations - rectangle and image are needed. Rectangle draw calls the basic drawing function - draw fill, and image draw calls the basic drawing function - draw map. We can observe that the draw fill, a basic drawing operation called by rectangle draw takes around 88% of the total rendering cycles thereby hogging the most cycles.

Widget 2 - Arc

Figure 4.8: Image of the Arc widget produced on the display

Operation	Number of cycles
Render frame	1304852
Flush frame	5073469
Miscellaneous	1911917
Total	8290238

Table 4.3: Numerical breakdown of cycles taken to produce the Arc widget

The next widget which was drawn was the arc widget as shown in figure 4.8. The table 4.3 shows the numerical breakdown of the total cycles taken to produce the arc widget on the display.

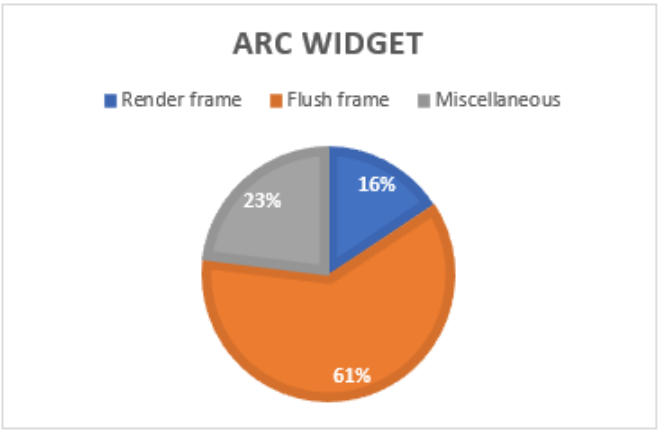

Figure 4.9: Graphical breakdown of cycles taken to produce the Arc widget

This numerical breakdown is represented in a graphical format as shown in figure 4.9 to denote the % of the total cycles taken by each of the 3 components. It can be observed that flushing the frame to the display takes the highest number of cycles at 61% of the total cycles and the frame rendering takes the least number of cycles at 16%.

Drawing Operation		Number of times	Number of cycles	% of total cycles
Advanced			1276033	97.79
Rectangle		3	1062797	81.45
	Basic		1062196	81.40
	Draw fill	1	1062196	81.40
	Residual		601	0.05
Arc		1	213236	16.34
	Basic		99920	7.66
	Draw fill	109	44780	3.43
	Draw pixel	286	55140	4.23
	Math		42958	3.29
	Sqrt	124	27994	2.15
	Atan2	175	14964	1.15
	Residual		70358	5.39
Others			28819	2.21
TOTAL			1304852	100.00

Table 4.4: Breakdown of drawing operations in rendering the frame - Arc widget

The cycles taken to render the frame in the buffer is further broken down to the hierarchy of drawing operations called to render the arc widget as shown in table 4.4. It can be seen that to draw the widget as shown in the figure 4.8, 2 advanced drawing operations - rectangle and arc are needed. Rectangle draw calls the basic drawing function - draw fill. Arc draw calls the basic drawing functions - draw fill and draw pixel, and math operations - sqrt and atan2. We can observe that the draw fill which is a basic drawing operation is called by both the advanced drawing operations and around 85% of the total rendering cycles is spent in this drawing operation thereby hogging the most time.

Widget 3 - Check Box

Figure 4.10: Image of the Checkbox widget produced on the display

Operation	Number of cycles
Render frame	1619708
Flush frame	5073470
Miscellaneous	1911918
Total	8605096

Table 4.5: Numerical breakdown of cycles taken to produce the Checkbox widget

The next widget which was drawn was the check box widget as shown in figure 4.10. The table 4.5 shows the numerical breakdown of the total cycles taken to produce the check box widget on the display.

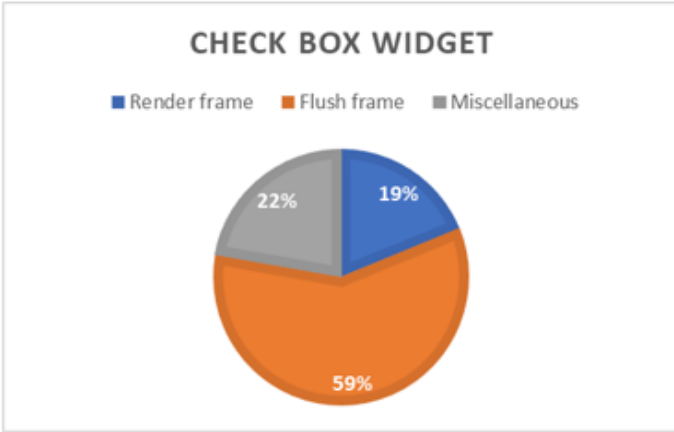

Figure 4.11: Graphical breakdown of cycles taken to produce the Checkbox widget

This numerical breakdown is represented in a graphical format as shown in figure 4.11 to denote the % of the total cycles taken by each of the 3 components. It can be observed that flushing the frame to the display takes the highest number of cycles at 59% of the total cycles and the frame rendering takes the least number of cycles at 19%.

Drawing operation		Number of times	Number of cycles	% of total cycles
Advanced			1264303	78.06
Rectangle		5	1134547	70.05
	Basic		1116032	68.90
	Draw pixel	100	19894	1.23
	Draw fill	56	1096138	67.68
	Residual		18515	1.14
Label		1	129756	8.01
	Basic		117077	7.23
	Draw letter	9	117077	7.23
	Residual		12679	0.78
Miscellaneous			355405	21.94
TOTAL			1619708	100.00

Table 4.6: Breakdown of drawing operations in rendering the frame - Checkbox widget

The cycles taken to render the frame in the buffer is further broken down to the hierarchy of drawing operations called to render the check box widget as shown in table 4.6. It can be seen that to draw the widget as shown in the figure 4.10, 2 advanced drawing operations - rectangle and label are needed. Rectangle draw calls the basic drawing functions - draw fill and draw pixel. Label draw calls the basic drawing function - draw letter. We can observe that the draw fill which is a basic drawing operation takes around 68% of the total rendering cycles thereby hogging the most time.

Widget 4 - Chart

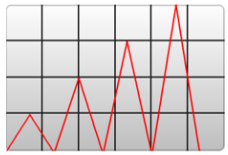

Figure 4.12: Image of the Chart widget produced on the display

Operation	Number of cycles
Render frame	3958866
Flush frame	5073469
Miscellaneous	1912018
Total	10944353

Table 4.7: Numerical breakdown of cycles taken to produce the Chart widget

The next widget which was drawn was the chart widget as shown in figure 4.12. The table 4.7 shows the numerical breakdown of the total cycles taken to produce the chart widget on the display.

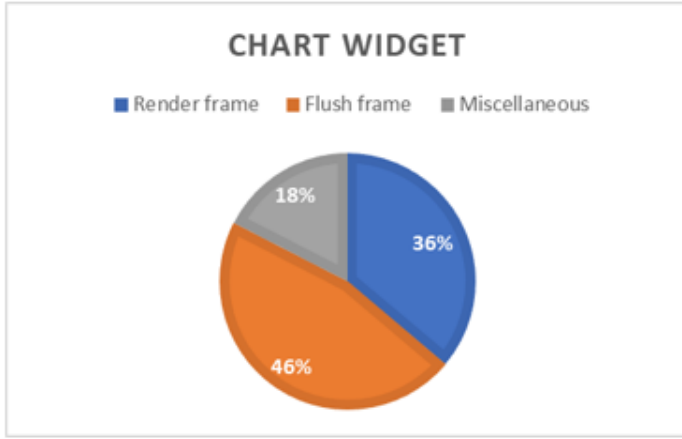

Figure 4.13: Graphical breakdown of cycles taken to produce the Chart widget

This numerical breakdown is represented in a graphical format as shown in figure 4.13 to denote the % of the total cycles taken by each of the 3 components. It can be observed that flushing the frame to the display takes the highest number of cycles at 46% of the total cycles followed by the the frame rendering at 36%.

Drawing operation		Number of times	Number of cycles	% of total cycles
Advanced			3921577	99.06
Line		16	1012239	25.57
	Basic		705091	17.81
	Draw pixel	2032	434200	10.97
	Draw fill	274	270891	6.84
	Residual		307148	7.76
Rectangle		4	2909338	73.49
	Basic		2862509	72.31
	Draw pixel	100	19892	0.50
	Draw fill	237	2842617	71.80
	Residual		46829	1.18
Miscellaneous			37289	0.94
TOTAL			3958866	100

Table 4.8: Breakdown of drawing operations in rendering the frame - Chart widget

The cycles taken to render the frame in the buffer is further broken down to the hierarchy of drawing operations called to render the chart widget as shown in table 4.8. It can be seen that to draw the widget as shown in the figure 4.12, 2 advanced drawing operations - rectangle and line are needed. Both of them call the basic drawing functions - draw fill and draw pixel. We can observe that the draw fill which is a basic drawing operation called by both line draw and rectangle draw takes around 79% of the total rendering cycles thereby hogging the most time.

Widget 5 - Color Picker (CPicker)

Figure 4.14: Image of the Cpicker widget produced on the display

Operation	Number of cycles
Render frame	8300608
Flush frame	5073470
Miscellaneous	1911918
Total	15285996

Table 4.9: Numerical breakdown of cycles taken to produce the Cpicker widget

The last widget which was drawn was the cpicker or color picker widget as shown in figure 4.14. The table 4.9 shows the numerical breakdown of the total cycles taken to produce the cpicker widget on the display.

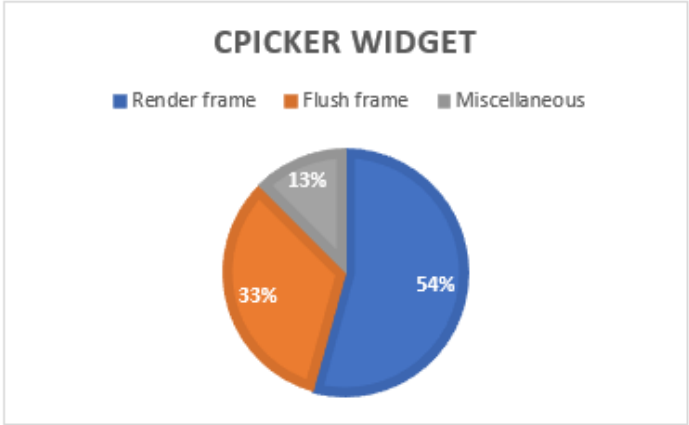

Figure 4.15: Graphical breakdown of cycles taken to produce the Cpicker widget

This numerical breakdown is represented in a graphical format as shown in figure 4.11 to denote the % of the total cycles taken by each of the 3 components. It can be observed that rendering the frame takes the highest number of cycles at 54% of the total cycles followed by flushing the frame to the display at 33%.

Drawing operation	Number of times	Number of cycles	% of total cycles
Advanced		8168982	98.41
Rectangle	5	1867160	22.49
Basic		1759879	21.20
Draw fill	291	1610456	19.40
Draw pixel	708	149423	1.80
Residual		107281	1.29
Triangle	120	6301822	75.92
Basic		4173593	50.28
Draw fill	5063	4173593	50.28
Residual		2128229	25.64
Math		22096	0.27
Trigo_sin	488	22096	0.27
Miscellaneous		109530	1.32
TOTAL		8300608	100.00

Table 4.10: Breakdown of drawing operations in rendering the frame - Cpicker widget

The cycles taken to render the frame in the buffer is further broken down to the hierarchy of drawing operations called to render the cpicker widget as shown in table 4.10. It can be seen that to draw this widget as shown in the figure 4.8, 2 advanced drawing operations - rectangle and triangle are needed. In addition to this, the math function trigo_sin is called independently and not from a higher order drawing function like in the arc widget rendering. Rectangle draw calls the basic drawing functions - draw fill and draw pixel. Triangle draw calls the basic drawing function - draw fill. We can observe that the draw fill which is a basic drawing operation called by both rectangle draw and line draw takes around 70% of the total rendering cycles thereby hogging the most time.

4.5 Discussion

In the results section, 5 widgets were drawn and the results were illustrated. The table 4.11 shows the % of the total cycles taken for rendering and flushing the frame for all the 5 widgets.

Widget	Flush frame	Render frame
Image	62%	15%
Arc	61%	16%
Checkbox	59%	19%
Chart	46%	36%
Cpicker	33%	54%

Table 4.11: % of the total cycles taken to render and flush the frame for each widget

Considering the breakdown of cycles, we observe that the % of cycles taken to flush the frame to the display from the frame buffer via the display controller ranges from 33% in the cpicker widget to 62% of the total cycles in the image widget.

We know that the frame buffer is located in the memory which has a latency of 1 cycle. Flushing the frame involves reading the memory pixel by pixel by the display controller and transferring them to the display. The display controller cannot directly access the memory, so the CPU has to move the data from the memory to the display controller. This consumes the CPU cycles and not a good use of the processing time of the CPU.

Instead if the display controller had DMA, it can read and write to the memory directly which would increase the processing speed of the application and save the CPU cycles thereby freeing up the CPU and allowing it to do something else instead of transferring data from the memory to the display controller.

From the table, it can be also be seen that the % of cycles taken to render the frame increases from 15% in the image widget to 54% in the cpicker widget. Thus, we can observe that as the complexity of the frame being rendered increases, the % of cycles to render also increases and can act as a bottleneck to the overall processing speed. This is the reason why benchmarking the rendering of the frame in detail is important, it helps us identify which drawing operations hog the most cycles and therefore make most sense to be accelerated.

We can observe that there are 3 categories of drawing functions - advanced, basic and math. There are 6 advanced drawing functions for drawing an arc, image, label, line, rectangle and triangle. There are 4 basic drawing functions which are further called by these advanced drawing functions for putting a pixel at a position (draw pixel), filling an area (draw fill), drawing a letter (draw letter) and drawing a color map or image (draw map) in the buffer. In addition to these, there are also some math operations like square root (sqrt), trigonometric sin (trigo_sin), arc tangent of two numbers (atan2)and bezier curve (bezier).

The widgets call the advanced drawing functions which further make calls to the basic drawing functions. The math functions can be called by the advanced drawing functions like the arc draw which called sqrt and atan2 or independently like the cpicker widget called trigo_sin.

LVGL has more than 30+ widgets and analyzing all of them would have been a tedious and lengthy task. So 5 were arbitrarily chosen so that almost all the drawing functions are covered and reasonable conclusions could be drawn.

Drawing Operation	**Draw Pixel**	**Draw fill**	**Draw letter**	**Draw map**
Arc	Yes	Yes	-	-
Image	-	-	-	Yes
Label	-	-	Yes	-
Line	Yes	Yes	-	-
Rectangle	Yes	Yes	-	-
Triangle	-	Yes	-	-

Table 4.12: Basic drawing operations called by the advanced drawing operations in LVGL

The table 4.12 shows which all basic drawing functions can be called by the advanced drawing functions. It is to be noted that rectangle draw can call draw fill and draw pixel, but it does not do so in all situations. Either one or both or none can be called while drawing the rectangle. The same is true for all the advanced drawing functions.

In case of all the 5 widgets, we observe that the draw operation which hogs the most cycles is the draw fill ranging from 68% of the total rendering cycles in case of the arc widget to 88% of the total rendering cycles in case of the image widget. While rendering, the memory has to be accessed to read and write to the frame buffer. Therefore, to truly offload the operation from the CPU and accelerate it, DMA is needed even in this case.

In conclusion, the two architectural improvements which can be done in which application speed can be improved and CPU cycles saved are:

1. Adding DMA capability to the Display Controller
2. Accelerating the draw fill operation which hogs the most cycles when the frame is rendered by means of a hardware accelerator having DMA capability.

These 2 improvements are done and explored further in the next chapter.

Chapter 5

Architectural Exploration

In the previous chapter, the benchmarking was set up for the simulator and the application code running on it. Based on the results, two potential improvements for improving the processing speed and saving the CPU cycles were discussed. One was adding DMA capability to the display controller model and the other was designing a hardware accelerator model with DMA for accelerating the drawing operations which hog the most CPU cycles. The design of these two architectural improvements is explained in the first two sections. The third section is the results sections where these architectural improvements are explored on the simulator under different configurations of the system. The chapter ends with the discussion section which discusses the results in the previous section.

5.1 Improvement 1 - Display Controller with DMA

LVGL has a function which flushes the contents of the frame buffer to the specific area on the display. This function is called after the frame has been rendered in the buffer. This function has 2 parameters, the area parameter which contains the reference to the area on the the frame buffer from where to pick the pixels one by one and a color parameter containing the reference to the colors of the pixels. The area parameter is a structure which has components to represent the area coordinates - 2 x coordinates and 2 y coordinates, and the color parameter which has components to represent the RGB color values.

In the previous display controller design as seen in the figure 3.5, the CPU transfers the data from the frame buffer which is located in the RAM to the display controller by writing to its registers - pixel data and pixel address. The 2D coordinates of the pixel are written to the pixel address register and the RGB color of the pixel is written to the pixel data register. Once the data is written to both the registers, the display controller calls the QDBMP function to place the pixel in the BMP file emulating the display. So, first the data for a pixel is written to the registers of the display controller by the CPU and then the display controller writes to the display that is, the BMP file. This process repeats in a loop, pixel by pixel until the entire area is flushed to the display. In this design, the display controller model is connected as a target to the interconnect model.

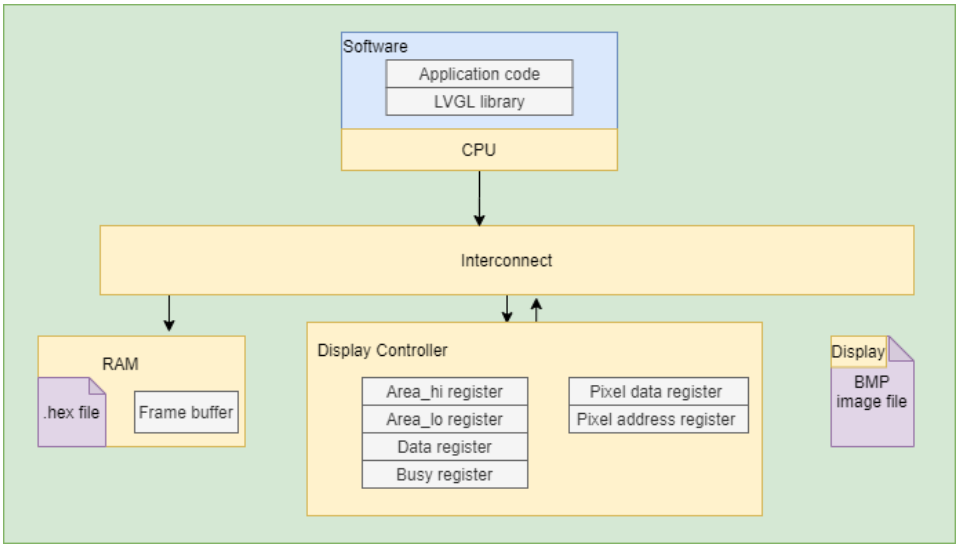

Figure 5.1: Graphics subsystem with display controller having DMA

The old design is modified as shown in figure 5.1 to add DMA capability. 4 new registers are added, one register for storing the x coordinates of the area (area_lo), one register for storing the y coordinates of the area region (area_hi), one register for storing the reference to the pixels in the buffer (data), and one read only register to indicate if the display controller is busy (busy). This display controller is connected both as an initiator and target to the interconnect.

When the flush function is called, the CPU writes to the registers of the display controller once and it is then free to do something else. When the display controller sees that data has been written to its 3 registers, it sets the busy register high and starts flushing the pixels to the screen one by one. It has the reference to the pixels in the data register, it can therefore read the colors of the pixels one by one directly from the RAM (containing the frame buffer) unlike the old design where the CPU had to read the data from RAM and then write to the registers of the display controller. Since it is connected as an initiator to the interconnect, it sends the transactions to the interconnect which routes the transaction to the RAM target by decoding the address.

Once the display controller is done flushing, it sets the busy register low to indicate that is done flushing out all the pixels. On the LVGL side, after the CPU writes to the registers of the display controller, it keeps polling the busy register and when it is cleared, it exits the loop. The number of cycles the CPU spends in this loop is the number of the cycles it is free to do something else like work on another process. In the case of our design, the CPU does not do anything else when it is free and stays in the loop. But, in reality the DMA would actually send an IRQ to the CPU, so the CPU does not have to poll the busy register. If an OS was executing on the CPU, some other processes may be executing in the meantime and the OS would switch back to the graphics process when the DMA IRQ comes.

5.2 Improvement 2 - Hardware Accelerator

It was concluded in the previous chapter that the draw fill function hogs most of the frame rendering time. The draw fill function in LVGL uses 2 static functions - blend and fill.

In the blend function, the source and destination memories are looped over their entire length. For every memory location, the source color is blended with the destination color using the opacity and the resulting color returned by the lv_color_mix function is written to the destination buffer. The code snippet 5.1 shows the lv_color_mix function which is a mix of multiplications and additions in terms of calculations. These calculations must be moved from the application code to the SystemC code to accelerate them.

```
1 static inline lv_color_t lv_color_mix(lv_color_t c1, lv_color_t c2
    , uint8_t mix)
2 {
3     lv_color_t ret;
4 #if LV_COLOR_DEPTH != 1
5     /*LV_COLOR_DEPTH == 8, 16 or 32*/
6     LV_COLOR_SET_R(ret, (uint16_t)((uint16_t) LV_COLOR_GET_R(c1) *
       mix + LV_COLOR_GET_R(c2) * (255 - mix)) >> 8);
7     LV_COLOR_SET_G(ret, (uint16_t)((uint16_t) LV_COLOR_GET_G(c1) *
       mix + LV_COLOR_GET_G(c2) * (255 - mix)) >> 8);
8     LV_COLOR_SET_B(ret, (uint16_t)((uint16_t) LV_COLOR_GET_B(c1) *
       mix + LV_COLOR_GET_B(c2) * (255 - mix)) >> 8);
```

Listing 5.1: Code snippet of the lv_color_mix function in LVGL

In the fill function, the entire fill area is looped over using the x and y coordinates and the area is filled with the color by writing to the destination buffer. Both these functions are accelerated by moving the logic which performs these operations from the application code to the SystemC code, since the hardware accelerator model is implemented in SystemC.

The display driver in LVGL has the option to include GPU callback functions which can be called instead of the software functions if the MCU has a hardware accelerator/GPU. The callback functions are made to call the accelerator by writing to its registers and the functionality is defined in the accelerator.

In the design in the previous section, the application code is run on the CPU, and the CPU renders or makes the frame in the frame buffer. In this section, a hardware accelerator is designed as shown in 5.2 to offload the blend and fill operations from the CPU to the accelerator thus freeing up the CPU to do something else.

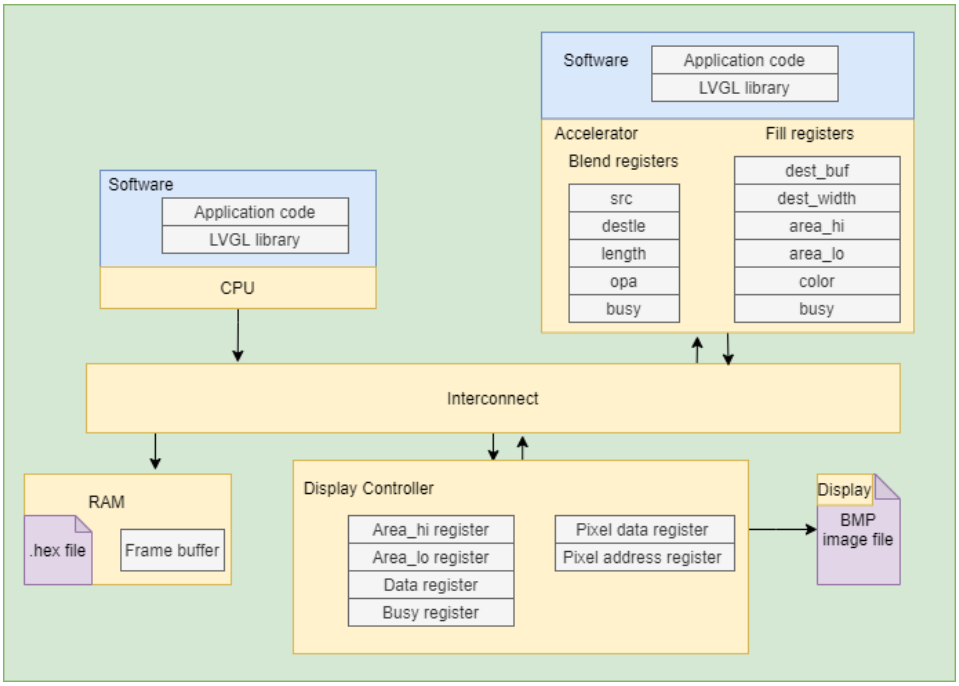

Figure 5.2: Graphics subsystem with display controller having DMA and hardware accelerator to offload some application code from the CPU

As shown in figure 5.2, the hardware accelerator is connected both as an initiator and target to the interconnect. The accelerator not only offloads a part of the application code from the CPU, it also has DMA to read and write from the RAM memory directly via the interconnect which decodes the address and routes the transactions initiated by the accelerator to the RAM. It has 4 registers for storing the data needed for blend operation and 1 register to indicate that it is busy doing the blend operation. Similarly, it has 5 registers for storing the data needed for fill operation and 1 register to indicate that it is busy doing the fill operation.

Blend and fill are added as GPU callback functions in the LVGL display driver. The LVGL library is written in such a way that if the GPU interface is active (this can be set in the LVGL's configuration header file), then instead of using software drawing functions, the GPU callback functions are called instead when the frame is being rendered.

When the blend function is called, the references to the destination and src memory, and the values of length and opacity are written by the CPU to the blend registers of the accelerator. When all these registers are written into, the blend busy register is set and the blending operation begins. The color values of the source and destination are read from the RAM. The source and destination colors are then mixed using the opacity ratio and the resulting color is written to the destination (that is the frame buffer which is in the RAM). After the entire blending operation is over, the blend busy register is cleared.

The fill function also works similarly. When the fill function is called, the reference to the destination buffer, and the values of the destination width, area coordinates, and fill color are written by the CPU to the fill registers of the accelerator. When all these registers are written into, the fill busy register is set and the fill operation begins. The area of fill is looped over and the fill color is written to the destination buffer (that is the frame buffer which is in the RAM memory). After the entire fill operation is over, the fill busy register is cleared.

On the LVGL side, after the CPU writes to the registers of the blend or the fill function, it keeps polling the busy register and when it is cleared, it exits the loop. The number of cycles the CPU spends in this loop is the number of the cycles it is free to do something else like work on another process. In the case of our design, the CPU does not do anything else when it is free and stays in the loop. As mentioned in the previous section, in reality, the accelerator would send an IRQ to the CPU, so that it does not have to poll the busy register and in case an OS is executing on the CPU, other processes can be executing in the meantime.

5.3 Results

The architectural improvements done in the first two subsections are explored in this section. 4 architectural configurations are explored:

1. **Baseline** - We start with the baseline architecture like in the figure 3.5 with the application code running on the CPU which prepares the frame buffer and afterwards reads the data from the buffer and transfers to the display controller which drives it to the display.
2. **DC with DMA** - Next, the display controller with DMA as shown in figure 5.1 is used which can pick up the data from the frame buffer directly.
3. **Accelerator** - Next, the accelerator as shown in figure 5.2 is connected with the baseline display controller (unlike in figure 5.2 where it is connected with a DMA display controller). The accelerator offloads some part of frame rendering (fill and blend operations) from the CPU.
4. **DC with DMA + Accelerator** - Finally, both the architectural improvements - display controller with DMA and the accelerator are connected together as shown in figure 5.2.

Two scenarios are further explored and discussed:

1. Drawing the same widget by varying the frame buffer configuration and architectural configuration.
2. Drawing different widgets by keeping the frame buffer configuration fixed and varying the architectural configuration.

5.3.1 Same widget with varying frame buffer configurations

The chart widget's performance is explored in the different architectural configurations by varying the frame buffer location and latency. 3 configurations of the frame buffer are explored:

1. **Local FB1** - Frame buffer in the local RAM (with a latency of 1 cycle) of the CPU on which the application code is run.
2. **Local FB2** - Frame buffer in the local RAM (with a latency of 2 cycles) of the CPU on which the application code is run.
3. **Shared FB** - Frame buffer in the shared RAM (with a latency of 1 cycles) of the local cores. A latency of 5 cycles is taken for the transaction moving from the local core to the shared core through the bridge model.

Architecture	Local FB1	Local FB2	Shared FB
Baseline	0%	0%	0
DC with DMA	45%	44%	46%
Accelerator	23%	22%	23%
DC with DMA + Accelerator	68%	66%	69%

Table 5.1: The % reduction in total cycles relative to the baseline architecture in the chart widget, under different frame buffer and architecture configurations

The table 5.1 shows the % of reduction in the total cycles relative to the total cycles of the baseline architecture, under different architectural and frame buffer configurations for the chart widget. Since the % reductions are with respect to the baseline architecture, the row of baseline is 0%. We can observe that in all the three cases of the frame buffer, the % reduction of total cycles relative to the baseline is almost the same with very slight variation. The average % reductions obtained when using the DC with DMA alone is 45%, when using accelerator alone is 23%, and when using both of them together we get 68% which is the sum of using the two improvements individually.

Architecture	Local FB1	Local FB2	Shared FB
Baseline	0%	0%	0%
DC with DMA	3%	3%	12%
Accelerator	3%	3%	12%
DC with DMA + Accelerator	11%	13%	50%

Table 5.2: % of total cycles saved in different widgets under same frame buffer configuration and different architecture configurations

In addition to the reduction in total cycles relative to the baseline architecture, some of the cycles are also saved when the DC with DMA or the accelerator are active. The table 5.2 shows the % of total cycles saved, under different architectural and frame buffer configurations for the chart widget. It is seen that when the latency of the local frame buffer is increased, the % savings increases slightly from 11% to 13% in case of using both the architectural improvements. The savings is more dramatic and reaches 50% when the buffer is in the shared core. This drastic change can be attributed to the increased delay cycles due to the latency in moving via the bridge from the local to the shared core. Every transaction moving via the bridge takes 5 clock cycles thus significantly adding to the total cycles, but a good % of these total cycles are saved due to the DMA capability of both the architectural improvements.

5.3.2 Different widgets with the same frame buffer configuration

3 different widgets with varying percentages of render and flush cycles making up the total cycles taken to draw them on the display are explored under the same frame buffer configuration. The frame buffer is in the local RAM with a latency of 1 cycle.

1. **Chart** - The chart widget has flush cycles making up 46% and the render cycles making up 36% of the total cycles taken to draw it on the display, as seen in the figure 4.13. Fill drawing operation makes up 79% of the total rendering cycles.
2. **Checkbox** - The checkbox widget has flush cycles making up 59% and the render cycles making up 19% of the total cycles taken to draw it on the display, as seen in the figure 4.11. Fill drawing operation makes up 68% of the total rendering cycles.
3. **Arc** - The arc widget has flush cycles making up 61% and the render cycles making up 16% of the total cycles taken to draw it on the display, as seen in the figure 4.9. Fill drawing operation makes up 85% of the total rendering cycles.

Architecture	Arc	Checkbox	Chart
Baseline	0%	0%	0%
DC with DMA	59%	58%	45%
Accelerator	11%	9%	23%
DC with DMA + Accelerator	70%	67%	68%

Table 5.3: The % reduction in total cycles relative to the baseline architecture in the different widgets, under different architecture configurations and same frame buffer configuration

The table 5.3 shows the % of reduction in the total cycles relative to the baseline architecture, under different architectural for the different widgets for the same frame buffer configuration. The % reduction in total cycles when using both the architectural improvements is almost the same. The total % reduction is the sum of % reduction when using both the improvements individually.

We can observe that for the arc, the DC with DMA contributes 59% while the % reduces slightly to 58% in the checkbox and decreases to 45 % in the chart. This is because of the % of flush cycles making up the total cycles decreases from 61% in arc to 59% in checkbox to 46% in chart.

On the other hand, the % of render cycles making up the total cycles increases from 16% in arc to 19% in checkbox to 36% in the chart. Therefore, the contribution of the accelerator to the total reduction increases from 11% in the arc to 23% in the chart. The one anomaly is the 9% contribution by the accelerator in case of the checkbox though the % of render cycles is greater in the checkbox than the arc. The contribution is expected to be greater than that of the arc which is 11%. To understand this, we need to look at the % of render cycles spent in the draw fill operation. The accelerator is designed to offload only the draw fill operation from the CPU. Though the % of render cycles is more in the checkbox than the arc, the % of render cycles spent in the draw fill operation in checkbox is 68% whereas in the arc it is 85%.

Architecture	Arc	Checkbox	Chart
Baseline	0%	0%	0%
DC with DMA	5%	4%	3%
Accelerator	2%	2%	3%
DC with DMA + Accelerator	12%	11%	11%

Table 5.4: % of total cycles saved in different widgets under same frame buffer configuration and different architecture configurations

The table 5.4 shows the % of total cycles saved when drawing the different widgets, under different architectural configurations for the same frame buffer configuration. It can observed that the difference is not very drastic for the three widgets under all architectural configurations. This is because they all use the same frame buffer configuration.

5.4 Discussion

Reduction in the total cycles relative to the baseline

When rendering the same widget, relative to the total cycles taken to produce the widget on the display in the baseline architecture, significant reduction in the total cycles was obtained. The reductions did not change much even when frame buffer configuration was varied from the local RAM with 1 cycle latency, to local RAM with 2 cycles latency and shared RAM with 1 cycle latency plus 5 cycles latency for movement from the local to the shared core.

A 68% reduction is obtained when using both the DC with DMA and accelerator. The average (of the three frame buffer configurations) total reduction 68% is the sum of the reductions due to the DC with DMA and accelerator when used individually at 45% and 23% respectively.

When rendering different widgets, the total reduction was the same with an average (of the three different widgets) of 68%, but the contributions by the DC with DMA and accelerator individually varied based on the % of render and flush cycles making up the total cycles. As the % of flush or render cycles making up the total cycles increases, the % of reduction due to the DC with DMA or accelerator alone also increases respectively. Even if the % of render cycles is high, the contribution of the accelerator to the total reduction depends on the % of these render cycles spent in the draw fill operation which the accelerator is designed to offload.

Savings in the total cycles

While rendering the same widget, the % of total cycles saved increases when the wait cycles in the bus/memory increases. This was observed when the % of total cycles saved jumped from 11% to 50% when the frame buffer location was changed from the local to the shared memory. The average of all the three frame buffer configurations was 25%.

While rendering different widgets by using the same frame buffer location in the local RAM, it was observed to be almost constant with no major jumps with an average(of the three widgets) of 11%.

Summary

To summarize, using both the DC with DMA and accelerator together gives significant reduction in the total cycles needed to render a GUI on the screen with an average of 68% reduction relative to the total cycles in the baseline architecture, where both these improvements are not used. This reduction in the total cycles leads to increase in the processing speed of the application. Not only are the total cycles significantly reduced, an average of 18% of the total cycles are also freed and the CPU is free to do do anything else like prepare for the next instruction, or switch processes or work on another task. In a nutshell, the two architectural improvements done have the dual benefit of speeding up the application and freeing up some cycles.

Chapter 6

Conclusion

This thesis successfully did what it set out to do in the beginning, that is analyze and improve the graphics processing performance in a typical microcontroller environment. This was achieved through three phases - setting up the simulator, setting up a benchmarking framework and doing baseline performance analysis, and finally architectural improvement and exploration phase.

The first part of the thesis was the setting up phase. To analyze the performance, a RISC-V ISA based simulator of a generic, heterogeneous, and multi-core SoC with shared memory and I/O was set up using SystemC + TLM. LVGL - a GUI library used for writing application code was ported to run on the simulator set up. The entire graphics subsystem was emulated on the simulator complete with the frame buffer in the RAM model, the GUI code written using LVGL running on the CPU model, a simple display controller model to which the contents of the frame buffer are driven to by the CPU model and which further outputs these contents to a BMP image file emulating a display.

The next natural phase was setting up a benchmarking framework for the simulator, so that the graphics processing performance can be quantified. Timing concepts in TLM models were studied and timing was set up in the hardware models of the simulator to emulate latencies of real-life hardware components. The application code was benchmarked to analyze the time taken by the various drawing operations in rendering a GUI scene.

Detailed baseline performance analysis was done by breaking down the total cycles taken to produce different GUI scenes on the display. This helped to note the operations which hogged the most cycles and therefore had potential for improvement. It was observed that flushing the frame from the buffer to the display by the baseline display controller took 33-62% of the total cycles and rendering the frame in the buffer by the CPU took between 15-54% of the total cycles. The draw fill operation took 68-88% of the rendering cycles. As the complexity of the frame being rendered increased, the % of render cycles making up the total cycles also increased with a corresponding decrease in the % of flush cycles.

Based on the results from the baseline performance analysis, two architectural improvements were designed. They were adding DMA capability to the display controller model and designing an accelerator model (also with DMA capability) for offloading the blend and fill operations from the CPU model. These architectural improvements were explored under various scenarios - same GUI scene with different frame buffer configurations and different GUI scenes with same frame buffer configuration. TLM modelling enabled efficient architectural exploration. Components could easily be added and defined at a high level of abstraction, topologies could easily be altered and explored. Doing the same in RTL modelling would be tedious and slow. Loosely timed TLM modelling could reproduce the behavior of RTL modelling while accelerating the simulation upto 100x [1], thus enabling the creation of early prototypes for software development and design space exploration.

To conclude, the architectural improvements were identified by setting up the simulator with a benchmarking framework and analyzing the baseline graphics performance. The improvements were then designed and analyzed by comparing to the baseline performance. They helped to speed up the application by providing an average 68% reduction in the total cycles compared to baseline cycles and also freed up an average of 18% of the total cycles, thus freeing up the CPU to do another task during these free cycles.

Coming to the limitations of the thesis, adding additional hardware to a chip design is a trade off between device cost increasing with increased die area, and the performance increase provided to the customer. Also, the development effort of specialized hardware might be prohibitive for a project, depending on the complexity. The accelerators provided here should be relatively small in terms of area, and of medium complexity in terms of design effort. Detailed analysis of the design effort and area estimates falls outside the scope of this thesis.

6.1 Future Work

This thesis has huge scope for future work. Two architectural improvements were explored in this work. Other potential architectural improvements which can be explored are: offloading the critical cycle-intensive tasks from the CPU to a co-processor and having custom RISC-V instructions for handling graphics operations in the simulator.

Other operations which have the potential for being accelerated could also be identified in LVGL and added to the accelerator. The architectural improvements could also be explored under more scenarios of the system, like changing the display resolution, CPU clock frequency, adding delays to other components and so on.

In this thesis, the improvements were explored on a SystemC + TLM simulator environment. In the future, the most promising improvement found in the architectural exploration phase can be implemented as synthesizable HDL code.

Bibliography

[1] Maman Abdurohman et al. “Transaction Level Modeling for Early Verification on Embedded System Design”. In: *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. 2009 Eighth IEEE/ACIS International Conference on Computer and Information Science. June 2009, pp. 277–282. DOI: 10.1109/ICIS.2009.41.

[2] *About SystemC*. URL: https://www.accellera.org/community/systemc/about-systemc (visited on 08/03/2020).

[3] Tomas Akenine-Moller and Jacob Strom. “Graphics Processing Units for Handhelds”. In: *Proceedings of the IEEE* 96 (June 1, 2008), pp. 779–789. DOI: 10.1109/JPROC.2008.917719.

[4] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, Aug. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[5] John Aynsley. *OSCI TLM 2.0 User Manual*. English. Duolos. July 2009. 194 pp. URL: https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf.

[6] Jeremy Bennett. “Building a Loosely Timed SoC Model with OSCI TLM 2.0 - A Case Study Using an Open Source ISS and Linux 2.6 Kernel”. In: (2008), p. 112.

[7] W. Cheng et al. “Transaction level model-based design methodology for fast architectural exploration and verification”. In: *2003 46th Midwest Symposium on Circuits and Systems*. 2003 46th Midwest Symposium on Circuits and Systems. Vol. 3. ISSN: 1548-3746. Dec. 2003, 1371–1374 Vol. 3. DOI: 10.1109/MWSCAS.2003.1562550.

[8] *GNU General Public License v3*. Library Catalog: www.gnu.org. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 07/19/2020).

[9] Daniel Hallmans, Mikael Åsberg, and Thomas Nolte. “Towards using the Graphics Processing Unit (GPU) for embedded systems”. In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*. ISSN: 1946-0759. Sept. 2012, pp. 1–4. DOI: 10.1109/ETFA.2012.6489715.

[10] F. Javaheri and Z. Navabi. “ESL design methodology for architecture exploration”. In: *2010 East-West Design & Test Symposium (EWDTS)*. Test Symposium (EWDTS). St. Petersburg, Russia: IEEE, Sept. 2010, pp. 395–401. ISBN: 978-1-4244-9555-9. DOI: 10.1109/EWDTS.2010.5742064. URL: http://ieeexplore.ieee.org/document/5742064/ (visited on 08/03/2020).

[11] W. Klingauf. “Systematic transaction level modeling of embedded systems with SystemC”. In: *Design, Automation and Test in Europe*. Design, Automation and Test in Europe. ISSN: 1558-1101. Mar. 2005, 566–567 Vol. 1. DOI: 10.1109/DATE.2005.293.

[12] LVGL LLC. *LVGL - Light and Versatile Embedded Graphics Library*. LVGL. Library Catalog: lvgl.io. URL: https://lvgl.io/ (visited on 08/02/2020).

[13] *lvgl/lvgl*. original-date: 2016-06-08T04:14:34Z. Aug. 4, 2020. URL: https://github.com/lvgl/lvgl (visited on 08/04/2020).

[14] madwyn. *madwyn/qdbmp*. original-date: 2014-06-06T10:24:46Z. Apr. 12, 2020. URL: https://github.com/madwyn/qdbmp (visited on 08/04/2020).

[15] *MCU Guided Selection Tool — Microchip Technology*. URL: https://www.microchip.com/design-centers/graphics/32-bit-mcu-guided-selection-tool (visited on 07/20/2020).

[16] *Microcontrollers with Graphics Display Capabilities — DigiKey*. URL: https://www.digikey.com/en/articles/techzone/2012/nov/microcontrollers-with-graphics-display-capabilities-are-in-high-demand (visited on 07/15/2020).

[17] H.W.M. van Moll et al. “Fast and accurate protocol specific bus modeling using TLM 2.0”. In: *Automation Test in Europe Conference Exhibition 2009 Design*. Automation Test in Europe Conference Exhibition 2009 Design. ISSN: 1558-1101. Apr. 2009, pp. 316–319. DOI: 10.1109/DATE.2009.5090680.

[18] Màrius Montón. *mariusmm/RISC-V-TLM*. original-date: 2018-09-10T16:41:14Z. July 19, 2020. URL: https://github.com/mariusmm/RISC-V-TLM (visited on 07/19/2020).

[19] Chinmayi Hassan Shyamprasad Nadig. *Graphics Processing in Embedded Applications*. Tech. rep. NTNU Specialization Project Report - Fall 2019. Norwegian University of Science and Technology, 2020.

[20] Luis Olea and Ioseph Martinez. “Introduction to Embedded Graphics with Freescale Devices”. In: (), p. 14.

[21] *RISC-V ISA*. RISC-V International. Library Catalog: riscv.org. URL: https://riscv.org/risc-v-isa/ (visited on 08/03/2020).

[22] *RISC-V Offers Simple, Modular ISA*. Library Catalog: riscv.org Section: RISC-V Foundation News. Apr. 1, 2016. URL: https://riscv.org/2016/04/risc-v-offers-simple-modular-isa/ (visited on 08/03/2020).

[23] *Simple DirectMedia Layer - Homepage*. URL: https://www.libsdl.org/ (visited on 08/04/2020).

[24] *Simulator on PC — LittlevGL 6.1.2 documentation*. URL: https://docs.lvgl.io/v6/en/html/get-started/pc-simulator.html (visited on 07/21/2020).

[25] Manuel F. Soto, J. Agustín Rodriguez, and Pablo R. Fillottrani. “SystemC/TLM flow for SoC design and verification”. In: *2015 Argentine School of Micro-Nanoelectronics, Technology and Applications (EAMTA)*. 2015 Argentine School of Micro-Nanoelectronics, Technology and Applications (EAMTA). July 2015, pp. 37–42. DOI: 10.1109/EAMTA.2015.7237376.

[26] S. Swan. “SystemC transaction level models and RTL verification”. In: *2006 43rd ACM/IEEE Design Automation Conference*. 2006 43rd ACM/IEEE Design Automation Conference. ISSN: 0738-100X. July 2006, pp. 90–92. DOI: 10.1145/1146909.1146937.

[27] *System overview — LittlevGL 6.0 documentation*. URL: https://docs.littlevgl.com/en/html/porting/sys.html (visited on 08/02/2020).

[28] *SystemC TLM*. URL: https://www.accellera.org/community/systemc/about-systemc-tlm (visited on 08/03/2020).

[29] *SystemC TLM-2.0*. URL: https://www.doulos.com/knowhow/systemc/tlm2/ (visited on 08/07/2020).

[30] Andrew Shell Waterman. “Design of the RISC-V Instruction Set Architecture”. PhD thesis. UC Berkeley, 2016. URL: https://escholarship.org/uc/item/7zj0b3m7 (visited on 08/03/2020).

Appendix A

Code Files

A.1 LVGL Files

A.1.1 Display Driver

```
/**
 * @file lv_port_disp_templ.h
 * @brief Display driver header file
 * @author Chinmayi Nadig
 * @date September 2020
 * Master's Thesis in Electronic Systems Design
 * Supervisors - Martin Olsson, Trond Ytterdal
 */

/*Copy this file as "lv_port_disp.h" and set this value to "1" to
    enable content*/
#if 1

#ifndef LV_PORT_DISP_TEMPL_H
#define LV_PORT_DISP_TEMPL_H

#ifdef __cplusplus
extern "C"
{
#endif

/*********************
 *      INCLUDES
 *********************/
#include "lvgl/lvgl.h"
```

```
25
26 /*********************
27  *      DEFINES
28  *********************/
29
30 /**********************
31  *      TYPEDEFS
32  **********************/
33
34 /**********************
35  * GLOBAL PROTOTYPES
36  **********************/
37
38 void lv_port_disp_init(void);
39 void lv_port_disp_flush(lv_disp_drv_t *disp_drv, const lv_area_t *
       area, lv_color_t *color_p);
40
41 /**********************
42  *      MACROS
43  **********************/
44
45 #ifdef __cplusplus
46 } /* extern "C" */
47 #endif
48
49 #endif /*LV_PORT_DISP_TEMPL_H*/
50
51 #endif /*Disable/Enable content*/
```

Listing A.1: Display Driver header file

```
1  /**
2   * @file lv_port_disp_templ.c
3   * @brief Display driver source file
4   * @author Chinmayi Nadig
5   * @date September 2020
6   * Master's Thesis in Electronic Systems Design
7   * Supervisors - Martin Olsson, Trond Ytterdal
8   */
9
10 /*Copy this file as "lv_port_disp.c" and set this value to "1" to
       enable content*/
11 #if 1
12
13 /*********************
14  *      INCLUDES
15  *********************/
16 #include "lv_port_disp.h"
17 #include <stdio.h>
18 /*********************
```

```
 *      DEFINES
 *********************/

/**********************
 *      TYPEDEFS
 **********************/

/**********************
 *  STATIC PROTOTYPES
 **********************/
static void disp_init(void);
#if LV_USE_GPU
static void gpu_blend(lv_disp_drv_t *disp_drv, lv_color_t *dest,
    const lv_color_t *src, uint32_t length, lv_opa_t opa);
static void gpu_fill(lv_disp_drv_t *disp_drv, lv_color_t *dest_buf
    , lv_coord_t dest_width, const lv_area_t *fill_area,
    lv_color_t color);
#endif

/**********************
 *  STATIC VARIABLES
 **********************/

/**********************
 *      MACROS
 **********************/
#define CSR_MCYCLE 0x300 // Address of the CSR register used for
    benchmarking

/* Registers of the Display controller (without DMA) */
#if LV_USE_DC
volatile unsigned int *pixel_address = (unsigned int *)0x0C00000C;
volatile unsigned int *pixel_data = (unsigned int *)0x0C000010;
#endif

/* Registers of the Display controller (with DMA) */
#if LV_USE_DC_DMA
volatile unsigned int *dc_area_lo = (unsigned int *)0x0C000000;
volatile unsigned int *dc_area_hi = (unsigned int *)0x0C000004;
volatile unsigned int *dc_data = (unsigned int *)0x0C000008;
volatile unsigned int *dc_busy = (unsigned int *)0x0C000014;
#endif

/* Registers of the Accelerator */
#if LV_USE_GPU
/* Blend registers */
volatile unsigned int *blend_src = (unsigned int *)0x0D000000;
volatile unsigned int *blend_dest = (unsigned int *)0x0D000004;
volatile unsigned int *blend_length = (unsigned int *)0x0D000008;
```

```
volatile unsigned int *blend_opa = (unsigned int *)0x0D00000C;

/* Fill registers */
volatile unsigned int *fill_dest_buf = (unsigned int *)0x0D000010;
volatile unsigned int *fill_area_lo = (unsigned int *)0x0D000014;
volatile unsigned int *fill_area_hi = (unsigned int *)0x0D000018;
volatile unsigned int *fill_dest_width = (unsigned int *)0
    x0D00001C;
volatile unsigned int *fill_color = (unsigned int *)0x0D000020;

/* Busy registers */
volatile unsigned int *blend_busy = (unsigned int *)0x0D000024;
volatile unsigned int *fill_busy = (unsigned int *)0x0D000028;
#endif

/* Declaration of the frame buffer in shared memory */
// static lv_color_t buf1_1[LV_HOR_RES_MAX * LV_VER_RES_MAX]
    __attribute__ ((section (".globalram")));

/**********************
 *   GLOBAL FUNCTIONS
 **********************/

void lv_port_disp_init(void)
{
    /*-------------------------
     * Initialize your display
     * -----------------------*/
    disp_init();

    /*-----------------------------
     * Create a buffer for drawing
     *-----------------------------*/

    static lv_disp_buf_t disp_buf_1;

    static lv_color_t buf1_1[LV_HOR_RES_MAX * LV_VER_RES_MAX];
                    // Declaration of the frame buffer in local
    memory
    lv_disp_buf_init(&disp_buf_1, buf1_1, NULL, LV_HOR_RES_MAX *
    LV_VER_RES_MAX); // Initialize the display buffer

    /*-----------------------------------
     * Register the display in LittlevGL
     *-----------------------------------*/

    lv_disp_drv_t disp_drv;      /*Descriptor of a display driver
    */
    lv_disp_drv_init(&disp_drv); /*Basic initialization*/
```

```
107
108     /*Set up the functions to access to your display*/
109
110     /*Set the resolution of the display*/
111     disp_drv.hor_res = 480;
112     disp_drv.ver_res = 320;
113
114     /*Used to copy the buffer's content to the display*/
115     disp_drv.flush_cb = lv_port_disp_flush;
116
117     /*Set a display buffer*/
118     disp_drv.buffer = &disp_buf_1;
119
120 #if LV_USE_GPU
121     /*Optionally add functions to access the GPU. (Only in
        buffered mode, LV_VDB_SIZE != 0)*/
122
123     /*Blend two color array using opacity*/
124     disp_drv.gpu_blend_cb = gpu_blend;
125
126     /*Fill a memory array with a color*/
127     disp_drv.gpu_fill_cb = gpu_fill;
128 #endif
129
130     /*Finally register the driver*/
131     lv_disp_drv_register(&disp_drv);
132 }
133
134 /* Flush the content of the internal buffer the specific area on
        the display
135  * You can use DMA or any hardware acceleration to do this
        operation in the background but
136  * 'lv_disp_flush_ready()' has to be called when finished. */
137 void lv_port_disp_flush(lv_disp_drv_t *disp_drv, const lv_area_t *
        area, lv_color_t *color_p)
138 {
139     /* CSR write to indicate end of frame rendering */
140     uint32_t value = 0x00C2;
141     csr_write(CSR_MCYCLE, value);
142
143     /* CSR write to indicate the start of frame flushing */
144     value = 0x00B1;
145     csr_write(CSR_MCYCLE, value);
146
147 #if LV_USE_DC_DMA
148     /* Writing to the registers of display controller. Pixel area
        (high and low bytes) and the pixel data */
149     *dc_area_lo = area->x2 << 16 | area->x1;
150     *dc_area_hi = area->y2 << 16 | area->y1;
```

```
151     *dc_data = color_p;
152 
153     /* Adding CSR writes to count the CPU cycles saved by calling
    the display controller with DMA */
154     value = 0x000C;
155     csr_write(CSR_MCYCLE, value);
156 
157     /* Polling the busy register of the display controller till it
     is cleared to exit the loop */
158     volatile int i = 1;
159     while (i)
160     {
161         i = *dc_busy;
162     }
163 
164     value = 0x000D;
165     csr_write(CSR_MCYCLE, value);
166 
167 #endif
168 
169 #if LV_USE_DC
170     /* The most simple case (but also the slowest) to put all
    pixels to the screen one-by-one */
171     int16_t x;
172     int16_t y;
173     for (y = area->y1; y <= area->y2; y++)
174     {
175         for (x = area->x1; x <= area->x2; x++)
176         {
177             /* Writing to the registers of display controller.
    Both pixel data and address */
178             *pixel_data = color_p->ch.red << 16 | color_p->ch.
    green << 8 | color_p->ch.blue;
179             color_p++;
180             *pixel_address = y << 16 | x;
181         }
182     }
183     *pixel_address = 0xffffffff;
184 #endif
185 
186     /* IMPORTANT!!!
187      * Inform the graphics library that you are ready with the
    flushing */
188     lv_disp_flush_ready(disp_drv);
189 
190     /* CSR write to indicate the end of frame flushing */
191     value = 0x00B2;
192     csr_write(CSR_MCYCLE, value);
193 }
```

```
194
195 /**********************
196  *   STATIC FUNCTIONS
197  **********************/
198
199 /* Initialize your display and the required peripherals. */
200 static void disp_init(void)
201 {
202     printf("In disp init function\n");
203 }
204
205 /*OPTIONAL: GPU INTERFACE*/
206 #if LV_USE_GPU
207
208 /* If your MCU has hardware accelerator (GPU) then you can use it
    to blend to memories using opacity
209  * It can be used only in buffered mode (LV_VDB_SIZE != 0 in
    lv_conf.h) */
210 static void gpu_blend(lv_disp_drv_t *disp_drv, lv_color_t *dest,
    const lv_color_t *src, uint32_t length, lv_opa_t opa)
211 {
212
213     /* Writing to the registers of the accelerator */
214     *blend_dest = dest;
215     *blend_src = src;
216     *blend_length = length;
217     *blend_opa = opa;
218
219     /* Adding CSR writes to count the CPU cycles saved by calling
    the blend function of the accelerator */
220     uint32_t value = 0x000E;
221     csr_write(CSR_MCYCLE, value);
222
223     /* Polling the blend busy register of the accelerator to exit
    the loop
224      * It is cleared when the accelerator is done with blending */
225     volatile int i = 1;
226     while (i)
227     {
228         i = *blend_busy;
229     }
230
231     value = 0x000F;
232     csr_write(CSR_MCYCLE, value);
233 }
234
235 /* If your MCU has hardware accelerator (GPU) then you can use it
    to fill a memory with a color
236  * It can be used only in buffered mode (LV_VDB_SIZE != 0 in
```

```
    lv_conf.h)*/
237 static void gpu_fill(lv_disp_drv_t *disp_drv, lv_color_t *dest_buf
    , lv_coord_t dest_width,
238                       const lv_area_t *fill_area, lv_color_t color)
239 {
240
241     /* Writing to the registers of the accelerator */
242     *fill_dest_buf = dest_buf;
243     *fill_dest_width = dest_width;
244     *fill_area_lo = fill_area->x2 << 16 | fill_area->x1;
245     *fill_area_hi = fill_area->y2 << 16 | fill_area->y1;
246     *fill_color = color.ch.alpha << 24 | color.ch.red << 16 |
    color.ch.green << 8 | color.ch.blue;
247
248     /* Adding CSR writes to count the CPU cycles saved by calling
    the fill function of the accelerator */
249     uint32_t value = 0x0021;
250     csr_write(CSR_MCYCLE, value);
251
252     /* Polling the fill busy register of the accelerator to exit
    the loop
253      * It is cleared when the accelerator is done with filling */
254     volatile int i = 1;
255     while (i)
256     {
257         i = *fill_busy;
258     }
259
260     value = 0x0022;
261     csr_write(CSR_MCYCLE, value);
262 }
263
264 #endif /*LV_USE_GPU*/
265
266 #else /* Enable this file at the top */
267
268 /* This dummy typedef exists purely to silence -Wpedantic. */
269 typedef int keep_pedantic_happy;
270 #endif
```

Listing A.2: Display Driver source file

A.1.2 Main file

```
1 /**
2  * @file main.c
3  * @brief LVGL main file
4  * @author Chinmayi Nadig
5  * @date September 2020
6  * Master's Thesis in Electronic Systems Design
```

```
 * Supervisors - Martin Olsson, Trond Ytterdal
 */

/*********************
 *      INCLUDES
 *********************/
#include "lvgl/lvgl.h"
#include "lv_conf.h"
#include "lv_port_disp.h"
#include "lv_ex_conf.h"
#include "lv_examples/lv_examples.h"
#include "lv_examples/lv_apps/benchmark/benchmark.h"
#include "lv_examples/lv_tests/lv_test.h"

extern int printf(const char *format, ...);

/*********************
 *      DEFINES
 *********************/
#define CSR_MCYCLE 0x300
/*********************
 *      TYPEDEFS
 *********************/

/**********************
 *  STATIC PROTOTYPES
 **********************/
static void gui_create(void);

/**********************
 *  STATIC VARIABLES
 **********************/

/*********************
 *      MACROS
 *********************/

/**********************
 *   GLOBAL FUNCTIONS
 **********************/

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

    (void)argc; /*Unused*/
    (void)argv; /*Unused*/
    printf("test hello\n");

    /*Initialize LittlevGL*/
```

```
56      lv_init();
57      printf("lvgl initialized\n");
58
59      /*Initialize the HAL (display, input devices, tick) for
        LittlevGL*/
60      lv_port_disp_init();
61      printf("HAL initialized\n");
62
63      /* CSR write to indicate the start of frame rendering */
64      uint32_t value = 0x00C1;
65      csr_write(CSR_MCYCLE, value);
66
67      /* Create a demo */
68      gui_create();
69      printf("GUI created\n");
70      lv_task_handler();
71      return 0;
72  }
73
74  /**********************
75   *   STATIC FUNCTIONS
76   **********************/
77
78  static void gui_create(void)
79
80  {
81
82      /* Uncomment depending on which widget you want to draw */
83
84      /* Create a default arc widget*/
85      // lv_obj_t * arc1 = lv_arc_create(lv_disp_get_scr_act(NULL),
        NULL);
86      // lv_obj_set_pos(arc1, 10, 10);
87
88      /* Create a default checkbox widget*/
89      //lv_obj_t * cb1 = lv_cb_create(lv_disp_get_scr_act(NULL),
        NULL);
90
91      /* Create a default chart widget*/
92      // lv_obj_t * chart1 = lv_chart_create(lv_disp_get_scr_act(
        NULL), NULL);
93      // lv_chart_series_t * dl1_1 = lv_chart_add_series(chart1,
        LV_COLOR_RED);
94      // dl1_1->points[0] = 0;
95      // dl1_1->points[1] = 25;
96      // dl1_1->points[2] = 0;
97      // dl1_1->points[3] = 50;
98      // dl1_1->points[4] = 0;
99      // dl1_1->points[5] = 75;
```

```
    // dl1_1->points[6] = 0;
    // dl1_1->points[7] = 100;
    // dl1_1->points[8] = 0;
    // lv_chart_series_t * dl1_2 = lv_chart_add_series(chart1,
    LV_COLOR_BLUE);
    // dl1_2->points[0] = 100;
    // lv_chart_refresh(chart1);

    /* Create a default image widget*/
    lv_test_img_1();

    /* Create a default color picker widget*/
    // lv_test_cpicker_1();
}
```

Listing A.3: GUI code main file

A.2 Simulator Files

A.2.1 Common header file

```
/**
 * @file Defines.h
 * @brief Common macros
 * @author Chinmayi Nadig
 * @date September 2020
 * Master's Thesis in Electronic Systems Design
 * Supervisors - Martin Olsson, Trond Ytterdal
 */

/* App domain registers */
#define printk_app          0x0B000100
#define exit_app            0x0B000130
#define startPC_app         0x0B000000

/* App1 domain registers */
#define printk_app1         0x07000100
#define exit_app1           0x07000130
#define startPC_app1        0x07000000

/* Shared domain registers */
#define printk_shared       0x0A000100
#define exit_shared         0x0A000130
#define startPC_shared      0x0A000000

/* Baseline display controller registers */
#define PIXEL_ADDRESS       0x0C00000C
#define PIXEL_DATA          0x0C000010

```

```
/* Display controller with DMA registers */
#define DC_AREA_LO          0x0C000000
#define DC_AREA_HI          0x0C000004
#define DC_DATA             0x0C000008
#define DC_BUSY             0x0C000014

/* Blend registers of the accelerator */
#define ACC_BLEND_SRC       0x0D000000
#define ACC_BLEND_DEST      0x0D000004
#define ACC_BLEND_LENGTH    0x0D000008
#define ACC_BLEND_OPA       0x0D00000C
#define ACC_BLEND_BUSY      0x0D000024

/* Fill registers of the accelerator */
#define ACC_FILL_DEST_BUF   0x0D000010
#define ACC_FILL_AREA_LO    0x0D000014
#define ACC_FILL_AREA_HI    0x0D000018
#define ACC_FILL_DEST_WIDTH 0x0D00001C
#define ACC_FILL_COLOR      0x0D000020
#define ACC_FILL_BUSY       0x0D000028
```

Listing A.4: Common macros of the system

A.2.2 Parser script for benchmarking

```
#
  #file benchmark.py
  #brief Parser script for producing compact data sets
  #author Chinmayi Nadig
  #date September 2020
  #Master's Thesis in Electronic Systems Design
  #Supervisors - Martin Olsson, Trond Ytterdal
#

import pprint
f = open ("cpicker.txt", "r") # Choose which widget's log file to
    open
func_adv = {}
func = []
datas = []
depth = 0
i = 0
for line in f: # Read line by line
    i = i + 1
    if line[0:5] == "START":
        depth = depth+1
        if depth == 1:
            func_adv[line[6:-1]] = {}
        func.append(line[6:-1])

```

```
25
26      elif line[0:5] == "CYCLE":
27          cycles = int(line[:-1].split(':')[1])
28
29          if len(func) > 1:
30              if func[depth-3] in func_adv[func[depth-4]]:
31                  func_adv[func[depth-4]][func[depth-3]]['num'] =
    func_adv[func[depth-4]][func[depth-3]]['num'] + 1
32                  func_adv[func[depth-4]][func[depth-3]]['cycles'] =
     func_adv[func[depth-4]][func[depth-3]]['cycles'] + cycles
33
34              else:
35                  func_adv[func[depth-4]][func[depth-3]] = {}
36                  func_adv[func[depth-4]][func[depth-3]]['num'] = 1
37                  func_adv[func[depth-4]][func[depth-3]]['cycles'] =
     cycles
38
39          else:
40              func_adv[func[0]]['num'] = 1
41              func_adv[func[0]]['cycles'] = cycles
42
43          depth = depth-1
44          func.pop()
45          if depth == 0:
46              datas.append(func_adv)
47              func_adv = {}
48      else:
49          print ("got nothing")
50
51  pp = pprint.PrettyPrinter(depth=6)
52  pp.pprint(datas)
```

Listing A.5: Parser script for benchmarking

A.2.3 Display Controller Model

```
1  /*!
2    \file Disp_Controller.h
3    \brief Top level header file of the display controller
4    \author Chinmayi Nadig
5    \date September 2020
6    Master's Thesis in Electronic Systems Design
7    Supervisors - Martin Olsson, Trond Ytterdal
8  */
9
10 #ifndef __DISP_CONTROLLER_H__
11 #define __DISP_CONTROLLER_H__
12
13 #include <iostream>
14 #include <fstream>
```

```
15
16 #define SC_INCLUDE_DYNAMIC_PROCESSES
17
18 #include "systemc"
19 #include "Defines.h"
20
21 #include "tlm.h"
22 #include "tlm_utils/simple_target_socket.h"
23 #include "tlm_utils/simple_initiator_socket.h"
24 #include "tlm_utils/tlm_quantumkeeper.h"
25
26 using namespace sc_core;
27 using namespace sc_dt;
28 using namespace std;
29
30 /**
31  * @class Disp_Controller
32  * @brief Display Controller model which can be used with and
       without DMA
33  *
34  */
35 class Disp_Controller : sc_module
36 {
37 public:
38   /*TLM-2 sockets, defaults to 32-bits wide, base protocol */
39   tlm_utils::simple_target_socket<Disp_Controller> target_socket;
40   tlm_utils::simple_initiator_socket<Disp_Controller> data_bus;
41
42   /**
43      * @brief Constructor
44      * @param name module name
45      * @param latency latency of the module
46      */
47   Disp_Controller(sc_module_name name, sc_time latency =
       SC_ZERO_TIME);
48
49   /**
50      * @brief TLM-2.0 socket implementation
51      * @param trans TLM-2.0 transaction
52      * @param delay transaction delay time
53      */
54   virtual void b_transport(tlm::tlm_generic_payload &trans,
       sc_time &delay);
55
56   /**
57   * Access data memory to get data for LOAD OPs
58   * @param addr address to access to
59   * @param size size of the data to read in bytes
60   * @return data value read
```

```
61    */
62    uint32_t readDataMem(uint32_t addr, int size);
63
64    /**
65    * Acces data memory to write data for STORE ops
66    * @brief
67    * @param addr addr address to access to
68    * @param data data to write
69    * @param size size of the data to write in bytes
70    */
71    void writeDataMem(uint32_t addr, uint32_t data, int size);
72
73    /**
74      * @brief Display controller flushing thread
75      */
76    void flush_loop();
77
78  private:
79    tlm_utils::tlm_quantumkeeper qk; /* Time keeping thread */
80    const sc_time LATENCY;           /* Latency of the module */
81
82    sc_uint<32> pixel_address; /* Register for storing 2D
        coordinates of pixel */
83    sc_uint<32> pixel_data;    /* Register for storing color data of
         pixel*/
84
85    sc_uint<64> dc_area; /* Register for storing the frame buffer
        area coordinates */
86    sc_uint<32> dc_data; /* Register for storing reference to the
        frame buffer*/
87    sc_uint<32> dc_busy; /* Register to indicate the controller is
        busy flushing */
88
89    uint8_t r, g, b;         /* Variables for storing the RGB color
        data */
90    uint16_t x1, x2, y1, y2; /* Variables for storing the area
        coordinates */
91    uint16_t x, y;           /* Variables for storing the pixel
        coordinates */
92    sc_uint<32> color;       /* Variable for storing the color value
         read from the frame buffer */
93  };
94
95  #endif
```

Listing A.6: Display Controller header file

```
1  /*!
2    \file Disp_Controller.cpp
3    \brief Top level source file of the display controller
```

```
4   \author Chinmayi Nadig
5   \date September 2020
6   Master's Thesis in Electronic Systems Design
7   Supervisors - Martin Olsson, Trond Ytterdal
8 */
9 #include "qdbmp.h"
10 #include "Disp_Controller.h"
11 BMP *bmp;
12 SC_HAS_PROCESS(Disp_Controller);
13 Disp_Controller::Disp_Controller(sc_module_name name, sc_time
       latency) : sc_module(name), target_socket("target_socket"),
       data_bus("initiator_socket"), LATENCY(latency)
14 {
15
16   target_socket.register_b_transport(this, &Disp_Controller::
       b_transport);
17   qk.set_global_quantum(sc_time(100, SC_NS)); // Replace the
       global quantum
18   qk.reset();                                 // Re-calculate the
       local quantum
19   SC_THREAD(flush_loop);
20 }
21
22 void Disp_Controller::b_transport(tlm::tlm_generic_payload &trans,
        sc_time &delay)
23 {
24
25   tlm::tlm_command cmd = trans.get_command();
26   sc_dt::uint64 addr = trans.get_address();
27   unsigned char *ptr = trans.get_data_ptr();
28   unsigned int len = trans.get_data_length();
29   delay += LATENCY; // Updating the local time offset with the
       latency of the component
30   uint32_t value = 0x0000;
31
32   if (cmd == tlm::TLM_WRITE_COMMAND)
33   {
34     memcpy(&value, ptr, len);
35     switch (addr)
36     {
37     case PIXEL_ADDRESS:
38       pixel_address.range(31, 0) = value;
39       x = (unsigned long)pixel_address.range(15, 0);
40       y = (unsigned long)pixel_address.range(31, 16);
41       if (value == 0x00000000)
42       {
43         bmp = BMP_Create(480, 320, 32); // Create a blank BMP
      image with the specified dimensions and 32 bit depth
44       }
```

```
45        if (value == 0xffffffff)
46        {
47          BMP_WriteFile(bmp, "file1.bmp"); // Writing to the BMP
      image file
48          BMP_CHECK_ERROR(stdout, -1);
49        }
50        break;
51
52      case PIXEL_DATA:
53        dc_data.range(23, 0) = value;
54        r = (unsigned char)dc_data.range(23, 16);
55        g = (unsigned char)dc_data.range(15, 8);
56        b = (unsigned char)dc_data.range(7, 0);
57        BMP_SetPixelRGB(bmp, x, y, r, g, b); // Setting the value of
       the pixel in the BMP image file
58        break;
59
60      case DC_AREA_LO:
61        dc_area.range(31, 0) = value;
62        x2 = (unsigned long)dc_area.range(31, 16);
63        x1 = (unsigned long)dc_area.range(15, 0);
64        break;
65
66      case DC_AREA_HI:
67        dc_area.range(63, 32) = value;
68        y2 = (unsigned long)dc_area.range(63, 48);
69        y1 = (unsigned long)dc_area.range(47, 32);
70        break;
71
72      case DC_DATA:
73        dc_data.range(31, 0) = value;
74        bmp = BMP_Create(480, 320, 32); // Create a blank BMP image
      with the specified dimensions and 32 bit depth
75        dc_busy.range(31, 0) = 1;       // Setting the dc_busy
      register high
76        break;
77
78      default:
79        cout << "Attempting to write to wrong register" << endl;
80      }
81    }
82    else if (cmd == tlm::TLM_READ_COMMAND)
83    {
84
85      switch (addr)
86      {
87      case PIXEL_ADDRESS:
88        value = pixel_address.range(31, 0);
89        break;
```

```
90
91      case PIXEL_DATA:
92        value = pixel_data.range(31, 0);
93        break;
94
95      case DC_AREA_LO:
96        value = dc_area.range(31, 0);
97        break;
98
99      case DC_AREA_HI:
100       value = dc_area.range(63, 32);
101       break;
102
103     case DC_DATA:
104       value = dc_data.range(31, 0);
105       break;
106
107     case DC_BUSY:
108       value = dc_busy.range(31, 0);
109       break;
110
111     default:
112       cout << "Attempting to read wrong register" << endl;
113     }
114     memcpy(ptr, &value, len);
115   }
116
117   trans.set_response_status(tlm::TLM_OK_RESPONSE);
118 }
119
120 void Disp_Controller::flush_loop()
121 {
122   cout << "Disp_Controller flush thread" << endl;
123   while (true)
124   {
125     int i = 0;
126     while (!dc_busy) // Waits until the dc_busy register is set
127       sc_core::wait(10, SC_NS);
128     /* The most simple case (but also the slowest) to put all
      pixels to the screen one-by-one */
129     for (y = y1; y <= y2; y++)
130     {
131       for (x = x1; x <= x2; x++)
132       {
133         color = readDataMem(dc_data + i * 4, 4); // Reading the
      color of the pixel from the frame buffer
134         r = color.range(23, 16);
135         g = color.range(15, 8);
136         b = color.range(7, 0);
```

```
137         BMP_SetPixelRGB(bmp, x, y, r, g, b); // Setting the value
    of the pixel in the BMP image file
138         i++;
139       }
140     }
141     if (dc_busy)
142       dc_busy = 0;                    // Clearing the dc_busy
    register
143     BMP_WriteFile(bmp, "file1.bmp"); // Writing to the BMP image
    file
144     BMP_CHECK_ERROR(stdout, -1);
145   }
146 }
147
148 uint32_t Disp_Controller::readDataMem(uint32_t addr, int size)
149 {
150   uint32_t data;
151   tlm::tlm_generic_payload trans;
152   sc_time delay = qk.get_local_time(); // Current local time
    offset assigned to delay
153
154   trans.set_command(tlm::TLM_READ_COMMAND);
155   trans.set_data_ptr(reinterpret_cast<unsigned char *>(&data));
156   trans.set_data_length(size);
157   trans.set_streaming_width(4); // = data_length to indicate no
    streaming
158   trans.set_byte_enable_ptr(0); // 0 indicates unused
159   trans.set_dmi_allowed(false); // Mandatory initial value
160   trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
161   trans.set_address(addr);
162
163   data_bus->b_transport(trans, delay); // Annotate b_transport
    with local time
164
165   qk.set(delay);       // Update qk with time consumed by target
166   if (qk.need_sync()) // Check local time against quantum
167     qk.sync();         // Updation of the global time
168
169   if (trans.is_response_error())
170   {
171     cout << "Failed to read " << dec << size << " bytes from data
    memory @0x" << hex << addr << ": 0x" << data << ", status: "
    << dec << trans.get_response_status() << endl;
172     SC_REPORT_WARNING("Memory", "Read memory");
173   }
174   return data;
175 }
176
177 void Disp_Controller::writeDataMem(uint32_t addr, uint32_t data,
```

```
    int size)
{
  tlm::tlm_generic_payload trans;
  sc_time delay = qk.get_local_time(); // Current local time
    offset assigned to delay

  trans.set_command(tlm::TLM_WRITE_COMMAND);
  trans.set_data_ptr(reinterpret_cast<unsigned char *>(&data));
  trans.set_data_length(size);
  trans.set_streaming_width(4); // = data_length to indicate no
    streaming
  trans.set_byte_enable_ptr(0); // 0 indicates unused
  trans.set_dmi_allowed(false); // Mandatory initial value
  trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
  trans.set_address(addr);

  data_bus->b_transport(trans, delay); // Annotate b_transport
    with local time

  qk.set(delay);      // Update qk with time consumed by target
  if (qk.need_sync()) // Check local time against quantum
    qk.sync();        // Updation of the global time

  if (trans.is_response_error())
  {
    cout << "Failed to write " << dec << size << " bytes to data
    memory @0x" << hex << addr << ": 0x" << data << ", status: "
    << dec << trans.get_response_status() << endl;
    SC_REPORT_WARNING("Memory", "Write memory");
  }
}
```

Listing A.7: Display Controller source file

A.2.4 Accelerator Model

```
/*!
  \file Accelerator.h
  \brief Top level header file of the accelerator
  \author Chinmayi Nadig
  \date September 2020
  Master's Thesis in Electronic Systems Design
  Supervisors - Martin Olsson, Trond Ytterdal
*/

#ifndef __ACCELERATOR_H__
#define __ACCELERATOR_H__

#include <iostream>
#include <fstream>
```

```
15
16 #define SC_INCLUDE_DYNAMIC_PROCESSES
17
18 #include "systemc"
19 #include "Defines.h"
20
21 #include "tlm.h"
22 #include "tlm_utils/simple_target_socket.h"
23 #include "tlm_utils/simple_initiator_socket.h"
24 #include "tlm_utils/tlm_quantumkeeper.h"
25
26 using namespace sc_core;
27 using namespace sc_dt;
28 using namespace std;
29
30 /**
31  * @class Accelerator
32  * @brief A hardware accelerator model for blend and fill
       operations
33  *
34  */
35 class Accelerator : sc_module
36 {
37 public:
38   /*TLM-2 sockets, defaults to 32-bits wide, base protocol */
39   tlm_utils::simple_target_socket<Accelerator> target_socket;
40   tlm_utils::simple_initiator_socket<Accelerator> data_bus;
41
42   /**
43      * @brief Constructor
44      * @param name module name
45      */
46   Accelerator(sc_module_name name, sc_time latency = SC_ZERO_TIME)
       ;
47
48   /**
49      * @brief TLM-2.0 socket implementation
50      * @param trans TLM-2.0 transaction
51      * @param delay transaction delay time
52      */
53   virtual void b_transport(tlm::tlm_generic_payload &trans,
       sc_time &delay);
54
55   /**
56     * Access data memory to get data for LOAD OPs
57     * @param addr address to access to
58     * @param size size of the data to read in bytes
59     * @return data value read
60     */
```

```
61   uint32_t readDataMem(uint32_t addr, int size);
62
63   /**
64     * Acces data memory to write data for STORE ops
65     * @brief
66     * @param addr addr address to access to
67     * @param data data to write
68     * @param size size of the data to write in bytes
69     */
70   void writeDataMem(uint32_t addr, uint32_t data, int size);
71
72   /**
73     * @brief Accelerator blending thread
74     */
75   void blend_loop();
76
77   /**
78     * @brief Accelerator filling thread
79     */
80   void fill_loop();
81
82 private:
83   tlm_utils::tlm_quantumkeeper qk; /* Time keeping thread */
84   const sc_time LATENCY;           /* Latency of the module */
85
86   sc_uint<32> blend_dest;   /* Register for storing reference to
       the blend destination memory */
87   sc_uint<32> blend_src;    /* Register for storing reference to
       the blend source memory */
88   sc_uint<32> blend_length; /* Register for storing length of
       blending */
89   sc_uint<8> blend_opa;     /* Register for storing opacity of
       blending */
90
91   sc_uint<32> fill_dest_buf;   /* Register for storing the
       reference to the fill destination buffer */
92   sc_uint<64> fill_area;       /* Register for storing area
       coordinates of fill area */
93   sc_uint<16> fill_dest_width; /* Register for storing fill
       destination buffer width */
94   sc_uint<32> fill_color;      /* Register for storing fill color
       */
95
96   sc_uint<32> fill_busy;  /* Register to indicate the accelerator
       is busy filling */
97   sc_uint<32> blend_busy; /* Register to indicate the accelerator
       is busy blending */
98
99   /* Variables for storing the RGB color data */
```

```
  uint8_t r, g, b, a;
  uint8_t dest_r, dest_g, dest_b;
  uint8_t src_r, src_g, src_b;
  uint32_t color;

  uint8_t i, opa, length;  /* Variables for storing opacity &
    length of blending */
  uint16_t x1, x2, y1, y2; /* Variables for storing the area
    coordinates */
  uint16_t x, y;           /* Variables for storing the pixel
    coordinates */

  sc_uint<32> src_data, dest_data; /* Variables for storing color
    values read from the frame buffer */
};

#endif
```

Listing A.8: Accelerator header file

```
/*!
  \file Accelerator.cpp
  \brief Top level source file of the accelerator
  \author Chinmayi Nadig
  \date September 2020
  Master's Thesis in Electronic Systems Design
  Supervisors - Martin Olsson, Trond Ytterdal
*/

#include "Accelerator.h"

SC_HAS_PROCESS(Accelerator);
Accelerator::Accelerator(sc_module_name name, sc_time latency) :
    sc_module(name), target_socket("target_socket"), data_bus("
    initiator_socket"), LATENCY(latency)
{
    target_socket.register_b_transport(this, &Accelerator::
    b_transport);
    qk.set_global_quantum(sc_time(100, SC_NS)); // Replace the
    global quantum
    qk.reset();                                 // Re-calculate
    the local quantum
    SC_THREAD(blend_loop);
    SC_THREAD(fill_loop);
}

void Accelerator::b_transport(tlm::tlm_generic_payload &trans,
    sc_time &delay)
{

```

```
    tlm::tlm_command cmd = trans.get_command();
    sc_dt::uint64 addr = trans.get_address();
    unsigned char *ptr = trans.get_data_ptr();
    unsigned int len = trans.get_data_length();
    delay += LATENCY; // Updating the local time offset with the
    latency of the component

    uint32_t value = 0x0000;

    if (cmd == tlm::TLM_WRITE_COMMAND)
    {
        memcpy(&value, ptr, len);
        switch (addr)
        {
        case ACC_BLEND_DEST:
            blend_dest.range(31, 0) = value;
            break;

        case ACC_BLEND_SRC:
            blend_src.range(31, 0) = value;
            break;

        case ACC_BLEND_LENGTH:
            blend_length.range(31, 0) = value;
            length = (unsigned int)blend_length.range(31, 0);
            break;

        case ACC_BLEND_OPA:
            blend_opa.range(7, 0) = value;
            opa = (unsigned int)blend_opa.range(7, 0);
            blend_busy.range(31, 0) = 1; // Setting blend_busy
    register high
            break;

        case ACC_FILL_AREA_LO:
            fill_area.range(31, 0) = value;
            x2 = (unsigned long)fill_area.range(31, 16);
            x1 = (unsigned long)fill_area.range(15, 0);
            break;

        case ACC_FILL_AREA_HI:
            fill_area.range(63, 32) = value;
            y2 = (unsigned long)fill_area.range(63, 48);
            y1 = (unsigned long)fill_area.range(47, 32);
            break;

        case ACC_FILL_DEST_BUF:
            fill_dest_buf.range(31, 0) = value;
            break;
```

```
72
73          case ACC_FILL_DEST_WIDTH:
74              fill_dest_width.range(15, 0) = value;
75              break;
76
77          case ACC_FILL_COLOR:
78              fill_color.range(31, 0) = value;
79              r = fill_color.range(23, 16);
80              g = fill_color.range(15, 8);
81              b = fill_color.range(7, 0);
82
83              color = r << 16 | g << 8 | b;
84
85              fill_busy.range(31, 0) = 1; // Setting the fill_busy
    register high
86
87              break;
88          }
89      }
90      else if (cmd == tlm::TLM_READ_COMMAND)
91      {
92
93          switch (addr)
94          {
95          case ACC_BLEND_BUSY:
96              value = blend_busy.range(31, 0);
97              break;
98
99          case ACC_FILL_BUSY:
100             value = fill_busy.range(31, 0);
101             break;
102
103         default:
104             cout << "Attempting to read wrong register" << endl;
105         }
106         memcpy(ptr, &value, len);
107     }
108
109     trans.set_response_status(tlm::TLM_OK_RESPONSE);
110 }
111
112 void Accelerator::blend_loop()
113 {
114     cout << "Accelerator blend thread" << endl;
115     while (true)
116     {
117         while (!blend_busy) // Waits until the blend_busy register
     is set
118             sc_core::wait(10, SC_NS);
```

```
119
120         for (i = 0; i <= length + 1; i++)
121         {
122             src_data = readDataMem(blend_src + i * 4, 4);   //
    Reading the color of the source pixel from frame buffer
123             dest_data = readDataMem(blend_dest + i * 4, 4); //
    Reading the color of the destination pixel from frame buffer
124
125             src_r = src_data.range(23, 16);
126             src_g = src_data.range(15, 8);
127             src_b = src_data.range(7, 0);
128
129             dest_r = dest_data.range(23, 16);
130             dest_g = dest_data.range(15, 8);
131             dest_b = src_data.range(7, 0);
132
133             r = (uint32_t)(((uint16_t)((uint16_t)dest_r * opa +
    src_r * (255 - opa)) >> 8) & 0xFF);
134             g = (uint32_t)(((uint16_t)((uint16_t)dest_g * opa +
    src_g * (255 - opa)) >> 8) & 0xFF);
135             b = (uint32_t)(((uint16_t)((uint16_t)dest_b * opa +
    src_b * (255 - opa)) >> 8) & 0xFF);
136
137             color = r << 16 | g << 8 | b;
138             writeDataMem(blend_dest + i * 4, color, 4); // Writing
     the blended color to the destination pixel in frame buffer
139         }
140
141         if (blend_busy)
142             blend_busy = 0; // Clearing the blend_busy register
143     }
144 }
145
146 void Accelerator::fill_loop()
147 {
148     cout << "Accelerator fill thread" << endl;
149     while (1)
150     {
151         while (!fill_busy) // Waits until the fill_busy register
    is set
152             sc_core::wait(10, SC_NS);
153
154         fill_dest_buf += fill_dest_width * 4 * y1; // Go to the
    first0 line
155         for (y = y1; y <= y2; y++)
156         {
157             for (x = x1; x <= x2; x++)
158             {
159                 writeDataMem(fill_dest_buf + x * 4, color, 4); //
```

```
    Writing to the pixel in the frame buffer
160             }
161             fill_dest_buf += fill_dest_width * 4; // Go to the
    next line
162         }
163         if (fill_busy)
164             fill_busy = 0; // Clearing the fill_busy register
165     }
166 }
167
168 uint32_t Accelerator::readDataMem(uint32_t addr, int size)
169 {
170     uint32_t data;
171     tlm::tlm_generic_payload trans;
172
173     sc_time delay = qk.get_local_time(); // Current local time
    offset assigned to delay
174
175     trans.set_command(tlm::TLM_READ_COMMAND);
176     trans.set_data_ptr(reinterpret_cast<unsigned char *>(&data));
177     trans.set_data_length(size);
178     trans.set_streaming_width(4); // = data_length to indicate no
    streaming
179     trans.set_byte_enable_ptr(0); // 0 indicates unused
180     trans.set_dmi_allowed(false); // Mandatory initial value
181     trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
182     trans.set_address(addr);
183
184     data_bus->b_transport(trans, delay); // Annotate b_transport
    with local time
185
186     qk.set(delay);      // Update qk with time consumed by target
187     if (qk.need_sync()) // Check local time against quantum
188         qk.sync();      // Updation of the global time
189
190     if (trans.is_response_error())
191     {
192         cout << "Failed to read " << dec << size << " bytes from
    data memory @0x" << hex << addr << ": 0x" << data << ", status
    : " << dec << trans.get_response_status() << endl;
193         SC_REPORT_WARNING("Memory", "Read memory");
194     }
195     return data;
196 }
197
198 void Accelerator::writeDataMem(uint32_t addr, uint32_t data, int
    size)
199 {
200     tlm::tlm_generic_payload trans;
```

```
201
202     sc_time delay = qk.get_local_time(); // Current local time
        offset assigned to delay
203
204     trans.set_command(tlm::TLM_WRITE_COMMAND);
205     trans.set_data_ptr(reinterpret_cast<unsigned char *>(&data));
206     trans.set_data_length(size);
207     trans.set_streaming_width(4); // = data_length to indicate no
        streaming
208     trans.set_byte_enable_ptr(0); // 0 indicates unused
209     trans.set_dmi_allowed(false); // Mandatory initial value
210     trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
211     trans.set_address(addr);
212
213     data_bus->b_transport(trans, delay); // Annotate b_transport
        with local time
214
215     qk.set(delay);       // Update qk with time consumed by target
216     if (qk.need_sync()) // Check local time against quantum
217         qk.sync();       // Updation of the global time
218
219     if (trans.is_response_error())
220     {
221         cout << "Failed to write " << dec << size << " bytes to
        data memory @0x" << hex << addr << ": 0x" << data << ", status
        : " << dec << trans.get_response_status() << endl;
222         SC_REPORT_WARNING("Memory", "Write memory");
223     }
224 }
```

Listing A.9: Accelerator source file

A.2.5 Domain Top files

```
1
2  /*!
3    \file Top_App.h
4    \brief Top level header file of the app domain
5    \author Chinmayi Nadig
6    \date September 2020
7    Master's Thesis in Electronic Systems Design
8    Supervisors - Martin Olsson, Trond Ytterdal
9  */
10
11 #ifndef __TOP_APP_H__
12 #define __TOP_APP_H__
13
14 #include <iostream>
15 #include <fstream>
16
```

```
17 #define SC_INCLUDE_DYNAMIC_PROCESSES
18
19 #include "systemc"
20
21 #include "tlm.h"
22 #include <signal.h>
23 #include <unistd.h>
24 #include "tlm_utils/simple_initiator_socket.h"
25 #include "tlm_utils/simple_target_socket.h"
26 #include "CPU.h"
27 #include "Ram.h"
28 #include "Interconnect.h"
29 #include "Bridge.h"
30 #include "Log.h"
31 #include "Irqmapper.h"
32 #include "Cache.h"
33 #include "Disp_Controller.h"
34 #include "Accelerator.h"
35
36 using namespace sc_core;
37 using namespace sc_dt;
38 using namespace std;
39
40 // RAM0 & 1
41 #define APPS1_R1_START_ADDR 0x0B000000
42 #define APPS1_R1_END_ADDR   0x0BFFFFFF
43 #define APPS2_R1_START_ADDR 0x2B000000
44 #define APPS2_R1_END_ADDR   0x2BFFFFFF
45
46 // BRIDGE FROM APP
47 #define APPS3_R1_START_ADDR 0x00000000
48 #define APPS3_R1_END_ADDR   0x0affffff
49 #define APPS3_R2_START_ADDR 0x0E000000
50 #define APPS3_R2_END_ADDR   0x2affffff
51 #define APPS3_R3_START_ADDR 0x2C000000
52 #define APPS3_R3_END_ADDR   0xffffffff
53
54 // DISPLAY CONTROLLER
55 #define APPS4_R1_START_ADDR 0x0C000000
56 #define APPS4_R1_END_ADDR   0x0Cffffff
57
58 // ACCELERATOR
59 #define APPS5_R1_START_ADDR 0x0D000000
60 #define APPS5_R1_END_ADDR   0x0Dffffff
61
62 /**
63  * @class Top_App
64  * @brief This class instantiates all necessary components
65  * of the app domain and connects their ports
```

```
 *
 */
class Top_App : sc_module
{
public:
    /* Declaring objects of the components */
    CPU *cpu;
    Irqmapper *irqmap;
    Bridge *bridge_to_app;
    Cache *cache;

    Ram *ram0;
    Ram *ram1;
    Bridge *bridge_from_app;

    Disp_Controller *disp_c;
    Accelerator *acc;
    Interconnect *bus;

    /* Initialize program counter value */
    int start_PC = APPS1_R1_START_ADDR;

    /**
   * @brief Constructor
   * @param name Module name
   *
   */
    Top_App(sc_module_name name);

    /**
   * @brief Destructor
   */
    ~Top_App();
};
#endif
```

Listing A.10: App domain top header

```


/*!
  \file Top_App.cpp
  \brief Top level source file of the app domain
  \author Chinmayi Nadig
  \date September 2020
  Master's Thesis in Electronic Systems Design
  Supervisors - Martin Olsson, Trond Ytterdal
*/

#include "Top_App.h"
```

```
13
14 SC_HAS_PROCESS(Top_App);
15 Top_App::Top_App(sc_module_name name) : sc_module(name)
16 {
17     /* Instantiating the objects of the components */
18     cpu = new CPU("cpu");
19     irqmap = new Irqmapper("irqmap");
20     bridge_to_app = new Bridge("bridge_to_app", sc_time(0, SC_NS))
       ;
21     cache = new Cache("cache", 0);
22
23     ram0 = new Ram("ram0", sc_time(10, SC_NS));
24     ram1 = new Ram("ram1", sc_time(10, SC_NS));
25     bridge_from_app = new Bridge("bridge_from_app", sc_time(50,
       SC_NS));
26     disp_c = new Disp_Controller("disp_c");
27
28     acc = new Accelerator("acc");
29     bus = new Interconnect("Interconnect");
30
31     /* Connecting the ports of the components */
32     bus->connect_socket<Ram>(ram0->socket, APPS1_R1_START_ADDR,
       APPS1_R1_END_ADDR);
33     bus->connect_socket<Ram>(ram1->socket, APPS2_R1_START_ADDR,
       APPS2_R1_END_ADDR);
34     bus->connect_socket<Bridge>(bridge_from_app->target_socket,
       APPS3_R1_START_ADDR, APPS3_R1_END_ADDR);
35     bus->add_address(APPS3_R2_START_ADDR, APPS3_R2_END_ADDR);
36     bus->add_address(APPS3_R3_START_ADDR, APPS3_R3_END_ADDR);
37     bus->connect_socket<Disp_Controller>(disp_c->target_socket,
       APPS4_R1_START_ADDR, APPS4_R1_END_ADDR);
38     bus->connect_socket<Accelerator>(acc->target_socket,
       APPS5_R1_START_ADDR, APPS5_R1_END_ADDR);
39
40     bus->connect_socket<Cache>(cache->memory_socket);
41     cpu->instr_bus.bind(cache->cpu_instr_socket);
42     cpu->exec->data_bus.bind(cache->cpu_data_socket);
43     irqmap->irq_line.bind(cpu->irq_line_socket);
44     bus->connect_socket<Bridge>(bridge_to_app->initiator_socket);
45     bus->connect_socket<Accelerator>(acc->data_bus);
46     bus->connect_socket<Disp_Controller>(disp_c->data_bus);
47 }
48
49 Top_App::~Top_App()
50 {
51     delete cpu;
52     delete irqmap;
53     delete bridge_to_app;
54
```

```
55      delete ram0;
56      delete ram1;
57      delete bridge_from_app;
58      delete disp_c;
59  }
```

Listing A.11: App domain top source file

```
1
2   /*!
3     \file Top_Shared.h
4     \brief Top level header file of the shared domain
5     \author Chinmayi Nadig
6     \date September 2020
7     Master's Thesis in Electronic Systems Design
8     Supervisors - Martin Olsson, Trond Ytterdal
9   */
10
11  #ifndef __TOP_SHARED_H__
12  #define __TOP_SHARED_H__
13
14  #include <iostream>
15  #include <fstream>
16
17  #define SC_INCLUDE_DYNAMIC_PROCESSES
18
19  #include "systemc"
20
21  #include "tlm.h"
22  #include <signal.h>
23  #include <unistd.h>
24  #include "tlm_utils/simple_initiator_socket.h"
25  #include "tlm_utils/simple_target_socket.h"
26  #include "CPU.h"
27  #include "Ram.h"
28  #include "Interconnect.h"
29  #include "Bridge.h"
30  #include "Log.h"
31  #include "Cache.h"
32  #include "Disp_Controller.h"
33  #include "Irqmapper.h"
34  #include "Accelerator.h"
35
36  using namespace sc_core;
37  using namespace sc_dt;
38  using namespace std;
39
40  // RAM 0 & 1
41  #define SHAREDS1_R1_START_ADDR  0x0A000000
42  #define SHAREDS1_R1_END_ADDR    0x0AFFFFFF
```

```
43 #define SHAREDS2_R1_START_ADDR  0x2A000000
44 #define SHAREDS2_R1_END_ADDR    0x2AFFFFFF
45
46 // BRIDGE TO APP
47 #define SHAREDS3_R1_START_ADDR  0x0B000000
48 #define SHAREDS3_R1_END_ADDR    0x0BFFFFFF
49 #define SHAREDS3_R2_START_ADDR  0x2B000000
50 #define SHAREDS3_R2_END_ADDR    0x2BFFFFFF
51
52 // BRIDGE TO APP1
53 #define SHAREDS4_R1_START_ADDR  0x07000000
54 #define SHAREDS4_R1_END_ADDR    0x07FFFFFF
55 #define SHAREDS4_R2_START_ADDR  0x27000000
56 #define SHAREDS4_R2_END_ADDR    0x27FFFFFF
57
58 // DISPLAY CONTROLLER
59 #define SHAREDS5_R1_START_ADDR  0x0C000000
60 #define SHAREDS5_R1_END_ADDR    0x0CFFFFFF
61
62 // DISPLAY CONTROLLER
63 #define SHAREDS6_R1_START_ADDR  0x0D000000
64 #define SHAREDS6_R1_END_ADDR    0x0DFFFFFF
65
66 /**
67  * @class Top_Shared
68  * @brief This class instantiates all necessary components
69  * of the shared domain and connects their ports
70  *
71  */
72 class Top_Shared : sc_module
73 {
74 public:
75     /* Declaring objects of the components */
76     CPU *cpu;
77     Irqmapper *irqmap;
78     Bridge *bridge_from_app;
79     Bridge *bridge_from_app1;
80     Cache *cache;
81
82     Bridge *bridge_to_app;
83     Bridge *bridge_to_app1;
84     Ram *ram0;
85     Ram *ram1;
86
87     Disp_Controller *disp_c;
88     Accelerator *acc;
89     Interconnect *bus;
90
91     /* Initialize program counter value */
```

```
92      int start_PC = SHAREDS1_R1_START_ADDR;
93
94      /**
95     * @brief Constructor
96     * @param name Module name
97     *
98     */
99      Top_Shared(sc_module_name name);
100
101     /**
102    * @brief Destructor
103    */
104     ~Top_Shared();
105 };
106 #endif
```

Listing A.12: Shared domain top header

```
1
2
3  /*!
4    \file Top_Shared.cpp
5    \brief Top level source file of the shared domain
6    \author Chinmayi Nadig
7    \date September 2020
8    Master's Thesis in Electronic Systems Design
9    Supervisors - Martin Olsson, Trond Ytterdal
10 */
11 #include "Top_Shared.h"
12
13 SC_HAS_PROCESS(Top_Shared);
14 Top_Shared::Top_Shared(sc_module_name name) : sc_module(name)
15 {
16     /* Instantiating the objects of the components */
17     bridge_from_app = new Bridge("bridge_from_app");
18     bridge_from_app1 = new Bridge("bridge_from_app1", sc_time(0,
       SC_NS));
19     cpu = new CPU("debug");
20     irqmap = new Irqmapper("irqmap");
21     cache = new Cache("cache", 0);
22
23     bridge_to_app = new Bridge("bridge_to_app");
24     bridge_to_app1 = new Bridge("bridge_to_app1", sc_time(0, SC_NS
       ));
25     ram0 = new Ram("ram0", sc_time(10, SC_NS));
26     ram1 = new Ram("ram1", sc_time(10, SC_NS));
27
28     disp_c = new Disp_Controller("disp_c");
29     acc = new Accelerator("acc");
30     bus = new Interconnect("Interconnect");
```

```
31
32      /* Connecting the ports of the components */
33      bus->connect_socket<Ram>(ram0->socket, SHAREDS1_R1_START_ADDR,
         SHAREDS1_R1_END_ADDR);
34      bus->connect_socket<Ram>(ram1->socket, SHAREDS2_R1_START_ADDR,
         SHAREDS2_R1_END_ADDR);
35      bus->connect_socket<Bridge>(bridge_to_app->target_socket,
        SHAREDS3_R1_START_ADDR, SHAREDS3_R1_END_ADDR);
36      bus->add_address(SHAREDS3_R2_START_ADDR, SHAREDS3_R2_END_ADDR)
        ;
37      bus->connect_socket<Bridge>(bridge_to_app1->target_socket,
        SHAREDS4_R1_START_ADDR, SHAREDS4_R1_END_ADDR);
38      bus->add_address(SHAREDS4_R2_START_ADDR, SHAREDS4_R2_END_ADDR)
        ;
39      bus->connect_socket<Disp_Controller>(disp_c->target_socket,
        SHAREDS5_R1_START_ADDR, SHAREDS5_R1_END_ADDR);
40      bus->connect_socket<Accelerator>(acc->target_socket,
        SHAREDS6_R1_START_ADDR, SHAREDS6_R1_END_ADDR);
41
42      bus->connect_socket<Bridge>(bridge_from_app->initiator_socket)
        ;
43      bus->connect_socket<Bridge>(bridge_from_app1->initiator_socket
        );
44      bus->connect_socket<Cache>(cache->memory_socket);
45      cpu->instr_bus.bind(cache->cpu_instr_socket);
46      cpu->exec->data_bus.bind(cache->cpu_data_socket);
47      irqmap->irq_line.bind(cpu->irq_line_socket);
48      bus->connect_socket<Accelerator>(acc->data_bus);
49      bus->connect_socket<Disp_Controller>(disp_c->data_bus);
50  }
51
52  Top_Shared::~Top_Shared()
53  {
54
55      delete cpu;
56      delete irqmap;
57      delete bridge_from_app;
58      delete bridge_from_app1;
59      delete cache;
60
61      delete bridge_to_app;
62      delete bridge_to_app1;
63      delete ram0;
64      delete ram1;
65  }
```

Listing A.13: Shared domain top source file

A.2.6 Overall Top file

```
/*!
  \file Simulator.cpp
  \brief Top level simulation entity
  \author Chinmayi Nadig
  \date September 2020
  Master's Thesis in Electronic Systems Design
  Supervisors - Martin Olsson, Trond Ytterdal
*/

#if !defined(MTI_SYSTEMC)
#pragma message("sccom not detected, compiling Simulator.cpp")
#pragma comment(lib, "systemc.h")

#define SC_INCLUDE_DYNAMIC_PROCESSES

#include <fstream>

#include "systemc"
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
#include <signal.h>
#include <unistd.h>
#include <iostream>
#include <string>
#include <sstream>
#include "Defines.h"

#include "Logger.h"
#include "CPU.h"
#include "Ram.h"
#include "Interconnect.h"
#include "Trace.h"
#include "Timer.h"
#include "Bridge.h"
#include "Top_App1.h"
#include "Top_Shared.h"
#include "Top_App.h"

using namespace sc_core;
using namespace sc_dt;
using namespace std;

int benchmark = 0;
int total_cpu = 0; //Number of CPUs input in the command line
string cpus;
string cpu[100];
string test;
fstream logfile;
```

```
50
51 /**
52  * @class Simulator
53  * This class instantiates all necessary domains, connects its
       ports and starts
54  * the simulation.
55  *
56  * @brief Top simulation entity
57  */
58 SC_MODULE(Simulator)
59 {
60   CPU *pointer_to_cpu;
61   int printk_addr;
62   int exit_addr;
63   int start_PC;
64   int cycle_counter = 0;
65
66   Top_Shared *shared;
67   Top_App *app;
68   Top_App1 *app1;
69
70   char cmd = 0;
71   void cpu_watch()
72   {
73     cout << "Entered cpu watch" << endl;
74     while (1)
75     {
76       pointer_to_cpu->single_step(benchmark);
77       sc_core::wait(10, SC_NS);
78       cycle_counter += 1;
79       sc_time cycle_time(10, SC_NS);
80
81       if (((pointer_to_cpu->register_bank->getPC()) == printk_addr
       ) && !(pointer_to_cpu->halted))
82       {
83         cout << hex << static_cast<unsigned char>(pointer_to_cpu->
       register_bank->getValue(10, false));
84       }
85       else if ((pointer_to_cpu->register_bank->getPC()) ==
       exit_addr)
86       {
87         if ((pointer_to_cpu->register_bank->getValue(10, false))
       == 0xCAFFE000)
88         {
89
90           cout << "-I TEST OK" << endl;
91           cout << "*****************************************"
       << endl;
92           pointer_to_cpu->register_bank->dump();
```

```
93          cout << "Instruction cycles: " << dec << cycle_counter
    << endl;
94          cout << "Total cycles: " << dec << (int)(pointer_to_cpu
    ->qk.get_current_time() / cycle_time) << endl;
95          //cout << "End time: " << sc_time_stamp() << endl;
96          cout << " End time: " << dec << pointer_to_cpu->qk.
    get_current_time() << endl;
97          pointer_to_cpu->perf->dump();
98
99          cout << "*******************************************"
    << endl;
100         cout << "Time taken to render the GUI in the frame
    buffer" << endl;
101         cout << "*******************************************"
    << endl;
102         cout << "Number of times: " << dec << pointer_to_cpu->
    render_num << endl;
103         cout << "Total CPU cycles taken: " << dec << (int)
    pointer_to_cpu->render_sumtotal << endl;
104
105         cout << "*******************************************"
    << endl;
106         cout << "Time taken to flush from DC to display" << endl
    ;
107         cout << "*******************************************"
    << endl;
108         cout << "Number of times: " << dec << pointer_to_cpu->
    dc_flush_num << endl;
109         cout << "Total CPU cycles taken: " << dec << (int)
    pointer_to_cpu->dc_flush_sumtotal << endl;
110
111         cout << "*******************************************"
    << endl;
112         cout << "Display Controller with DMA" << endl;
113         cout << "*******************************************"
    << endl;
114         cout << "Number of times: " << dec << pointer_to_cpu->
    dc_dma_num << endl;
115         cout << "Total CPU cycles saved: " << dec << (int)
    pointer_to_cpu->dc_dma_sumtotal << endl;
116
117         cout << "*******************************************"
    << endl;
118         cout << "Accelerator - blend operation " << endl;
119         cout << "*******************************************"
    << endl;
120         cout << "Number of times: " << dec << pointer_to_cpu->
    acc_blend_num << endl;
121         cout << "Total CPU cycles saved: " << dec << (int)
```

```
    pointer_to_cpu->acc_blend_sumtotal << endl;
122
123         cout << "*****************************************"
    << endl;
124         cout << "Accelerator - fill operation" << endl;
125         cout << "*****************************************"
    << endl;
126         cout << "Number of times: " << dec << pointer_to_cpu->
    acc_fill_num << endl;
127         cout << "Total CPU cycles saved: " << dec << (int)
    pointer_to_cpu->acc_fill_sumtotal << endl;
128
129         if (benchmark)
130         {
131
132           cout << "*****************************************
    " << endl;
133           cout << "LVGL Benchmarking - Math functions" << endl;
134           cout << "*****************************************
    " << endl;
135           cout << "lv_trigo_sin total cycles: " << dec << (int)
    pointer_to_cpu->trigo_sin_sumtotal << endl;
136           cout << "lv_trigo_sin number of times: " << dec <<
    pointer_to_cpu->trigo_sin_num << endl;
137
138           cout << "lv_bezier3 total cycles: " << dec << (int)
    pointer_to_cpu->bezier3_sumtotal << endl;
139           cout << "lv_bezier3 number of times: " << dec <<
    pointer_to_cpu->bezier3_num << endl;
140
141           cout << "lv_atan2_letter total cycles: " << dec << (
    int)pointer_to_cpu->atan2_sumtotal << endl;
142           cout << "lv_atan2 number of times: " << dec <<
    pointer_to_cpu->atan2_num << endl;
143
144           cout << "lv_sqrt total cycles: " << dec << (int)
    pointer_to_cpu->sqrt_sumtotal << endl;
145           cout << "lv_sqrt number of times: " << dec <<
    pointer_to_cpu->sqrt_num << endl;
146
147           cout << "*****************************************
    " << endl;
148           cout << "LVGL Benchmarking - BASIC drawing functions"
    << endl;
149           cout << "*****************************************
    " << endl;
150
151           cout << "lv_draw_px total cycles: " << dec << (int)
    pointer_to_cpu->draw_px_sumtotal << endl;
```

```
152             cout << "lv_draw_px number of times: " << dec <<
    pointer_to_cpu->draw_px_num << endl;
153
154             cout << "lv_draw_fill total cycles: " << dec << (int)
    pointer_to_cpu->draw_fill_sumtotal << endl;
155             cout << "lv_draw_fill number of times: " << dec <<
    pointer_to_cpu->draw_fill_num << endl;
156
157             cout << "lv_draw_letter total cycles: " << dec << (int
    )pointer_to_cpu->draw_letter_sumtotal << endl;
158             cout << "lv_draw_letter number of times: " << dec <<
    pointer_to_cpu->draw_letter_num << endl;
159
160             cout << "lv_draw_map total cycles: " << dec << (int)
    pointer_to_cpu->draw_map_sumtotal << endl;
161             cout << "lv_draw_map number of times: " << dec <<
    pointer_to_cpu->draw_map_num << endl;
162
163             cout << "*****************************************
    " << endl;
164             cout << "LVGL Benchmarking - ADVANCED drawing
    functions" << endl;
165             cout << "*****************************************
    " << endl;
166             cout << "lv_draw_arc total cycles: " << dec << (int)
    pointer_to_cpu->draw_arc_sumtotal << endl;
167             cout << "lv_draw_arc number of times: " << dec <<
    pointer_to_cpu->draw_arc_num << endl;
168
169             cout << "lv_draw_img total cycles: " << dec << (int)
    pointer_to_cpu->draw_img_sumtotal << endl;
170             cout << "lv_draw_img number of times: " << dec <<
    pointer_to_cpu->draw_img_num << endl;
171
172             cout << "lv_draw_label total cycles: " << dec << (int)
    pointer_to_cpu->draw_label_sumtotal << endl;
173             cout << "lv_draw_label number of times: " << dec <<
    pointer_to_cpu->draw_label_num << endl;
174
175             cout << "lv_draw_line total cycles: " << dec << (int)
    pointer_to_cpu->draw_line_sumtotal << endl;
176             cout << "lv_draw_line number of times: " << dec <<
    pointer_to_cpu->draw_line_num << endl;
177
178             cout << "lv_draw_rect total cycles: " << dec << (int)
    pointer_to_cpu->draw_rect_sumtotal << endl;
179             cout << "lv_draw_rect number of times: " << dec <<
    pointer_to_cpu->draw_rect_num << endl;
180
```

```
181             cout << "lv_draw_triangle total cycles: " << dec << (
    int)pointer_to_cpu->draw_triangle_sumtotal << endl;
182             cout << "lv_draw_triangle number of times: " << dec <<
     pointer_to_cpu->draw_triangle_num << endl;
183
184             cout << "****************************************
    " << endl;
185           }
186
187           //sc_stop();
188           break;
189         }
190         else if ((pointer_to_cpu->register_bank->getValue(10,
    false)) == 0xDEADD000)
191         {
192           cout << "-I TEST FAILED!" << endl;
193           break;
194           //SC_REPORT_ERROR("Exit function", "main returned 1");
195         }
196       }
197     }
198   }
199
200   SC_CTOR(Simulator)
201   {
202
203     /* Instantiating the domain objects */
204     app = new Top_App("app");
205     app1 = new Top_App1("app1");
206     shared = new Top_Shared("shared");
207
208     /* Connecting the top level domains via the bridges */
209     app->bridge_from_app->initiator_socket.bind(shared->
    bridge_from_app->target_socket);
210     shared->bridge_to_app->initiator_socket.bind(app->
    bridge_to_app->target_socket);
211     app1->bridge_from_app1->initiator_socket.bind(shared->
    bridge_from_app1->target_socket);
212     shared->bridge_to_app1->initiator_socket.bind(app1->
    bridge_to_app1->target_socket);
213
214     /* Selecting which CPU the program should run on based on the
    command line arguments*/
215     string filename;
216     for (int i = 0; i < total_cpu; i++)
217     {
218       if (cpu[i] == "app")
219       {
220         filename = "gap/tests/" + test + "/" + cpu[i] + "/" + test
```

```
    + ".ihex";
221         cout << filename << endl;
222         app->ram0->readHexFile(filename);
223         pointer_to_cpu = app->cpu;
224         start_PC = startPC_app;
225         printk_addr = printk_app;
226         exit_addr = exit_app;
227         pointer_to_cpu->register_bank->setValue(Registers::sp, 0
    x00000000);
228         pointer_to_cpu->register_bank->setPC(start_PC);
229         SC_THREAD(cpu_watch);
230       }
231
232       else if (cpu[i] == "app1")
233       {
234         filename = "gap/tests/" + test + "/" + cpu[i] + "/" + test
     + ".ihex";
235         cout << filename << endl;
236         app1->ram0->readHexFile(filename);
237         pointer_to_cpu = app1->cpu;
238         start_PC = startPC_app1;
239         printk_addr = printk_app1;
240         exit_addr = exit_app1;
241         pointer_to_cpu->register_bank->setValue(Registers::sp, 0
    x00000000);
242         pointer_to_cpu->register_bank->setPC(start_PC);
243         SC_THREAD(cpu_watch);
244       }
245
246       else if (cpu[i] == "shared")
247       {
248         filename = "gap/tests/" + test + "/" + cpu[i] + "/" + test
     + ".ihex";
249         cout << filename << endl;
250         shared->ram0->readHexFile(filename);
251         pointer_to_cpu = shared->cpu;
252         start_PC = startPC_shared;
253         printk_addr = printk_shared;
254         exit_addr = exit_shared;
255         pointer_to_cpu->register_bank->setValue(Registers::sp, 0
    x00000000);
256         pointer_to_cpu->register_bank->setPC(start_PC);
257         SC_THREAD(cpu_watch);
258       }
259
260       else
261         std::cout << "Entered an incorrect cpu name" << endl;
262     }
263   }
```

```
264
265   ~Simulator()
266   {
267     logfile.close();
268     cout << "Simulator destructor" << endl;
269     //delete shared; // Including these leads to additional CPU
      dumps being printed on terminal.
270     //delete app;
271     //delete app1;
272     delete pointer_to_cpu;
273   }
274 };
275
276 Simulator *top;
277
278 void intHandler(int dummy)
279 {
280   delete top;
281   sc_stop();
282   exit(-1);
283 }
284
285 /* Parsing and processing the command line arguments */
286 void process_arguments(int argc, char *argv[])
287 {
288
289   int c;
290   int debug_level;
291   Logger *log;
292
293   log = Logger::getInstance("Log_top.txt");
294   log->setLogLevel(Logger::ERROR);
295   while ((c = getopt(argc, argv, "D:T:C:b:?")) != -1)
296   {
297     switch (c)
298     {
299     case 'D':
300     { /* Select the level of logging */
301       debug_level = atoi(optarg);
302       switch (debug_level)
303       {
304       case 4:
305         log->setLogLevel(Logger::TRACE);
306         break;
307       case 3:
308         log->setLogLevel(Logger::INFO);
309         break;
310       case 2:
311         log->setLogLevel(Logger::WARNING);
```

```
312         break;
313       case 1:
314         log->setLogLevel(Logger::DEBUG);
315         break;
316       case 0:
317         log->setLogLevel(Logger::ERROR);
318         break;
319       default:
320         log->setLogLevel(Logger::INFO);
321       }
322       break;
323     }
324     case 'b':
325     {
326       /* Benchmark the LVGL drawing operations */
327       benchmark = atoi(optarg);
328       cout << "LVGL benchmarking mode activated" << endl;
329       break;
330     }
331     case 'T':
332     {
333       /* Select which test to run */
334       test = std::string(optarg);
335       cout << test << endl;
336       break;
337     }
338     case 'C':
339     {
340       /* Select which CPU to run the test on */
341       cpus = std::string(optarg);
342       cout << cpus << endl;
343       stringstream ss(cpus);
344       string cpus_token;
345       while (getline(ss, cpus_token, ','))
346       { //the comma seperated list of cpus is broken and
    individual cpu names are stored in cpu array
347         cpu[total_cpu] = cpus_token;
348         total_cpu++;
349       }
350       break;
351     }
352     case '?':
353       std::cout << "Call ./gaptlm -D<debuglevel> (0..4) -T <test>
    (printk/graphics/..) -C<CPUs seperated by commas> (app, app1,
    shared..)" << std::endl;
354       break;
355     }
356   }
357 }
```

```
358
359 int sc_main(int argc, char *argv[])
360 {
361
362   /* Capture Ctrl+C and finish the simulation */
363   signal(SIGINT, intHandler);
364
365   /* SystemC time resolution set to 1 ns*/
366   sc_set_time_resolution(1, SC_NS);
367
368   /* Parse and process program arguments.*/
369   process_arguments(argc, argv);
370
371   top = new Simulator("top");
372   cout << "Constructor successful!" << endl;
373   sc_start();
374   return 0;
375 }
376
377 #endif //MTI_SYSTEMC
```

Listing A.14: Overall top source file where all the domains are connected

Chinmayi Nadig

Master's Thesis 2020

NTNU
Norwegian University of
Science and Technology

NORDIC SEMICONDUCTOR