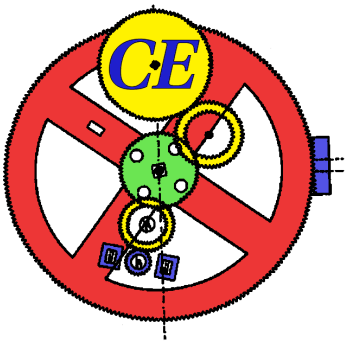

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2003

MSc THESIS

TR-DAMP:

Testing and redesigning the Delft Altera-based Multimedia Platform

W. Zwart

CE-MS-2003-06

Abstract

Currently it is envisioned that, for numerous industrial applications, a paradigm shift occurs from Application Specific Integrated Circuits (ASICs) towards reconfigurable hardware designs. Field Programmable Gate Arrays (FPGAs) constitute means for the implementation of reconfigurable processors and lately are integrated with General Purpose Processor (GPP) cores. Designing for such FPGA devices requires specific skills and methodologies, and is often supported by development kits. As seen from the Academic perspective, the existing development kits for FPGA devices suffer from two disadvantages, namely very high cost and insufficient flexibility, both making them unusable for educational and research purposes. The Delft Altera-based Multimedia Platform (DAMP) project is meant to produce a low-cost and flexible development board. DAMP's main target is to provide hardware and software support for the hardware/software co-design paradigm for multimedia applications. The Altera ARM-based Excalibur device with 100k or 400k gates around an ARM922T CPU is chosen as the main chip. This selection is due to the popularity of the ARM processor in a variety of embedded devices, some of them targeting the requirements on multimedia applications, e.g., 3G mobile phones and PDAs. In this thesis the testing methodology of the DAMP design is discussed. The results and the modifications that were made in order to obtain a functional prototype, are also presented. The outcome of the testing experiments was used to redesign the DAMP schematic design and the DAMP Printed Circuit Board (PCB) design. Furthermore, a number reference designs were developed in order to support future DAMP development.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Testing and redesigning the Delft Altera-based Multimedia Platform

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

W. Zwart
born in Gouda, Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Testing and redesigning the
Delft Altera-based Multimedia Platform

by W. Zwart

Abstract

Currently it is envisioned that, for numerous industrial applications, a paradigm shift occurs from Application Specific Integrated Circuits (ASICs) towards reconfigurable hardware designs. Field Programmable Gate Arrays (FPGAs) constitute means for the implementation of reconfigurable processors and lately are integrated with General Purpose Processor (GPP) cores. Designing for such FPGA devices requires specific skills and methodologies, and is often supported by development kits. As seen from the Academic perspective, the existing development kits for FPGA devices suffer from two disadvantages, namely very high cost and insufficient flexibility, both making them unusable for educational and research purposes. The Delft Altera-based Multimedia Platform (DAMP) project is meant to produce a low-cost and flexible development board. DAMP's main target is to provide hardware and software support for the hardware/software co-design paradigm for multimedia applications. The Altera ARM-based Excalibur device with 100k or 400k gates around an ARM922T CPU is chosen as the main chip. This selection is due to the popularity of the ARM processor in a variety of embedded devices, some of them targeting the requirements on multimedia applications, e.g., 3G mobile phones and PDAs. In this thesis the testing methodology of the DAMP design is discussed. The results and the modifications that were made in order to obtain a functional prototype, are also presented. The outcome of the testing experiments was used to redesign the DAMP schematic design and the DAMP Printed Circuit Board (PCB) design. Furthermore, a number reference designs were developed in order to support future DAMP development.

Laboratory : Computer Engineering

Codenumber : CE-MS-2003-06

Comittee Members :

Advisor:	Sorin Cotofana, CE, TU Delft
Advisor:	Georgi Gaydadjiev, CE, TU Delft
Chairperson:	Stephan Wong, CE, TU Delft
Member:	Stamatis Vassiliadis, CE, TU Delft

To my girlfriend Suzan for her love, support and endless patience.
And of course to my parents, who made it all possible.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I am deeply grateful to my advisors Dr. Sorin Cotofana and Ir. Georgi Gaydadjiev who were so helpful, patient and enthusiastic. Their patience, experience and encouragement are the valuable sources that have kept me on the right track. They have not only shown me the way to carry out the development work but more importantly, they also taught me how to work in a demanding environment.

Next, I want to thank Prof. Stamatis Vassiliadis for his encouraging and kind help during my study in the Computer Engineering group.

I would also want to thank all the other members in the Computer Engineering group, which I had a very good time with.

Delft, The Netherlands — Willem Zwart
October 31, 2003

1 Introduction

In the Industry, rapidly increasing design complexity and very tight time-to-market schedules are typical for many years by now. In the field of embedded systems design, a paradigm shift occurred from System-on-Chip towards Reconfigurable System-on-Chip (RSoC) designs. This shift requires new tools, skills, and methodologies. One way to help developers deal with these new challenges is by providing them with development kits. This practice is widely used even for designs based on simple micro-controllers, e.g., 8051-based systems, and provides two major advantages. First, by utilizing development kits selection of the most appropriate solution for the end product becomes feasible. Second, the utilization of working hardware together with example or reference designs can significantly speed up the development process and shorten the tools learning time. Since many of today's Computer Engineering students are the Industries embedded systems designers of tomorrow, they have to gain practical skills required by the real situation in the Industry. Therefore, it is important to include up-to-date development tools and state of the art hardware into the University educational programs.

As viewed from the Academic perspective, the existing development kits, such as the Altera EPXA10 development kit [1] and the Xilinx Virtex-II Pro [2] development kit, suffer from two main disadvantages: very high cost and insufficient interface options, e.g., lack of video out and audio in/out. Those two factors make them practically unusable for Academic and small Research/Industry groups. In order to address these disadvantages (with a limited budget in mind), it was decided to initiate the DAMP (Delft Altera-based Multimedia Platform) project which is meant to produce a low-cost multimedia enhanced platform built around an Altera Excalibur RSoC device. In this thesis the last stages of the DAMP project are described, namely testing a prototype, redesigning the schematics and PCB, and the development of reference designs, which are required to support future DAMP development.

This chapter is organized as follows. Section 1.1 gives background information concerning reconfigurable computing and system-on-chip design. Section 1.2 presents some information about the hardware/software co-design paradigm. Section 1.3 discusses the Altera Excalibur device. Section 1.4 presents the DAMP design goals and the utilized methodology. Section 1.5 discusses the framework of the thesis.

1.1 Reconfigurable System-on-Chip

The ongoing miniaturization process, the introduction of power efficient algorithms and computational structures, and the advances in battery technology has been boosting the portable devices market. Standard design practice for these portable devices is to

develop them around a single microchip that contains the electronic circuits and parts which can normally be found in a system. This microchip is referred to as a System-on-Chip (SoC). The key components of an SoC are a (micro)processor, memory and various interfaces to external devices.

Due to the increased performance and decreased cost, programmable devices are becoming standard components in many embedded systems. Today's Field Programmable Gate Arrays (FPGA's) provide over 10 million system gates composed of memory, datapath elements and reconfigurable logic gates. By using FPGA-based hardware in combination with a SoC design, a system can be reprogrammed to exhibit arbitrary hardware-based functionality. An SoC that includes reconfigurable logic is referred to as a Reconfigurable System-on-chip (RSoC).

The use of an RSoC design provides three major advantages when compared to Application Specific Integrated Circuit (ASIC) or General Purpose Processor (GPP) designs:

- First, the RSoC combines the speed of an ASIC with the versatility of a GPP, thus an RSoC can offer higher performance than a GPP and can be more flexible than an ASIC.

- Second, the use of RSoC designs reduces time to market when compared to traditional ASIC designs. This is because the real hardware is available immediately, while the ASIC has to be manufactured. In addition, bug fixes and design upgrades can be more extensive as significant portions of the hardware can be altered and not just the software.

- Third, the RSoC design provides the means for the utilization of run-time reconfigurability. Run-time reconfigurability implies that the behavior of application specific units can be changed under hardware or software control while maintaining the normal operation of the system.

There are two types of processor cores that can be used in an RSoC, namely a softcore processor implemented in reconfigurable logic, such as the Altera Nios processor [3], and a hardware-based processor like in the case of the Altera Excalibur device [4], which consists of an industry-standard ARM922T processor core and FPGA-based hardware. Another example of an RSoC with a hardware-based processor is the Xilinx Virtex-II Pro FPGA series [2] which integrate up to four IBM PowerPC 405 processors and FPGA-based hardware.

The ARM processor is very popular in many embedded devices, some of them targeting the 3G mobile telephony. An important part of the ARM processor is the Advanced Microcontroller Bus Architechure (AMBA) [5] bus, which is widely used by developers of SoCs. The AMBA bus is connecting the ARM processor with peripherals, such as a Liquid Crystal Display (LCD) controller and a Synchronous Dynamic Random Access Memory (SDRAM) controller. Two AMBA buses are present in the Altera Excalibur device. The Programmable Logic Device (PLD), which is the reconfigurable part of the Excalibur device, is one of the peripherals connected to the AMBA bus. Hence, it is possible to develop custom AMBA peripherals and to reconfigure them via a software program at run-time. Runtime reconfigurability and the presence of the AMBA bus,

give the Excalibur its unique properties. The Altera Excalibur device is therefore chosen as DAMP's backbone.

1.2 Hardware/Software Co-design

Many people tend to think of embedded systems in terms of end products, e.g., mobile telephones, PDAs. However, a system is an embedded system if it is a component of some larger system. Therefore SoCs and RSoCs are also embedded systems, which can be embedded in a larger environment, such as for example a television set.

DAMP supports the embedded systems development, and in fact already contains an embedded system, namely the Altera Excalibur RSoC. One of the main questions in designing embedded systems, is to decide what part of the application to map in hardware and what to map in software. This is called the hardware/software co-design paradigm. If an application is to be mapped on DAMP, this paradigm should be exercised in such a way that an optimal system design is produced.

When a given application has to be mapped on DAMP, the software can be run on the ARM processor and special hardware units can be programmed in the PLD. In this way, different implementations corresponding to the same application can be investigated, which helps finding the most appropriate implementation for a certain application.

1.3 Excalibur organization

The Excalibur device consists of two parts: a PLD and a stripe. The PLD is basically an FPGA fabric based on the Altera APEX 20KE technology [6]. The stripe contains an ARM processor and other peripherals, such as a Synchronous Dynamic Random Access Memory (SDRAM) interface, Universal Asynchronous Receiver/Transmitter (UART) interface, and an interrupt controller. Figure 1.1 presents the internal organization of the Excalibur device.

The ARM processor, the PLD and the on-chip peripherals are connected via two busses, which are based on the AMBA High-performance Bus (AHB) specification [5]. The first AMBA bus (`AHB1`) is the main bus and connects the processor to processor specific slaves, such as the interrupt controller and the watchdog timer. The processor is the only master on this bus. The internal Static Random Access Memory (SRAM) and the Dual Port Random Access Memory (DPRAM) are also connected to `AHB1`, which allows the processor fast memory access. Any transaction which is not intended for a peripheral on `AHB1` is routed to the `AHB1-2` bridge. This bridge is also one of the `AHB1` slaves, which gives the processor access to the AMBA bus `AHB2`.

AMBA bus `AHB2` is connected to all other peripherals and provides access to the PLD from the ARM processor. There are three potential bus masters on AHB2:

- **AHB1-2 bridge:** The `AHB1-2` bridge processes transactions that originate from the embedded processor on `AHB1` and whose destination is either on `AHB2` or in the PLD.
- **Configuration logic:** The Configuration logic configures the SRAM, DPRAM and the PLD.

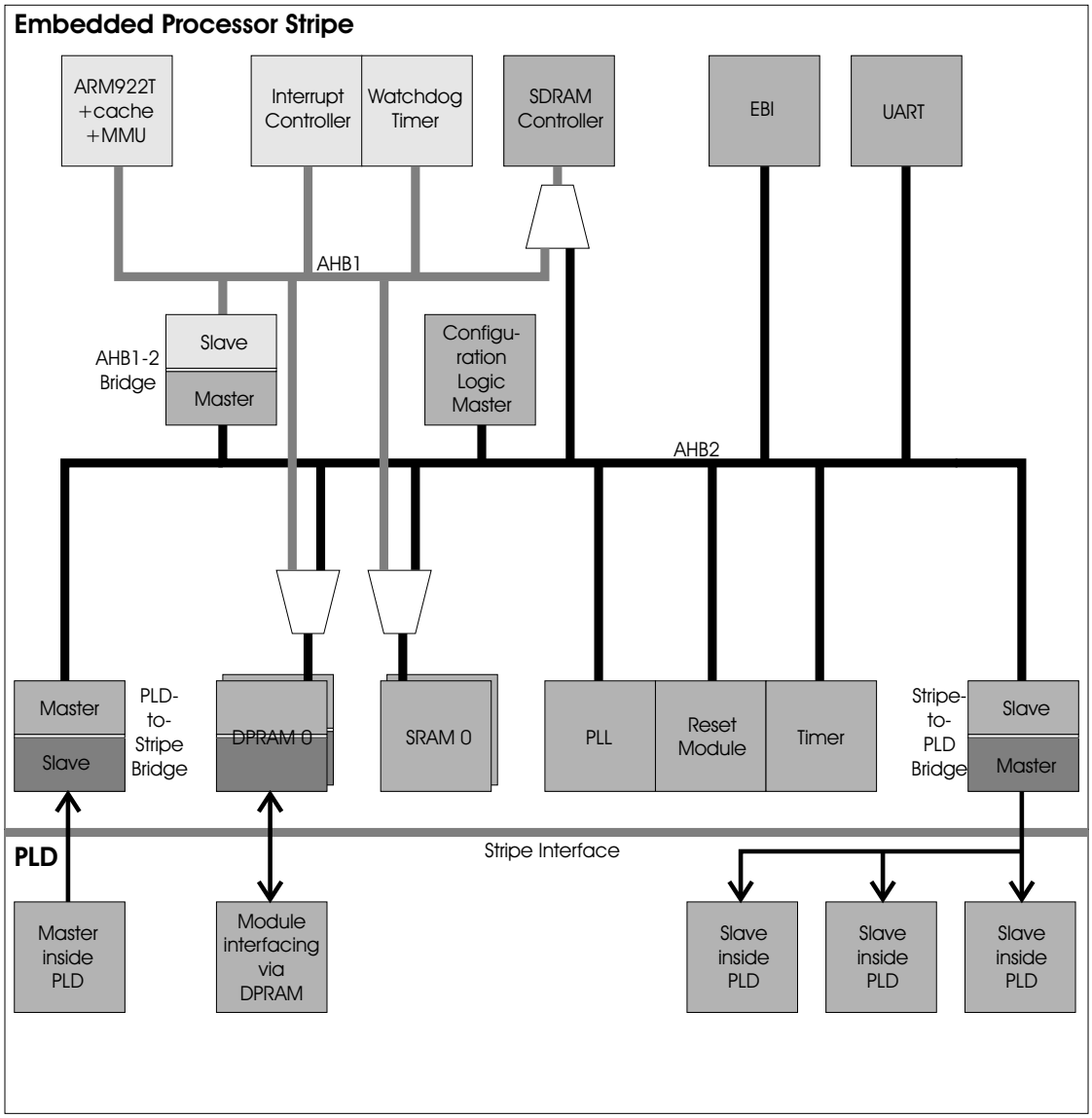

Figure 1.1: Internal organization of the Excalibur device.

- **PLD-to-Stripe bridge:** The PLD-to-Stripe bridge provides masters in the PLD access to slaves on `AHB2`.

All other peripherals, such as the UART and Stripe-to-PLD bridge, behave as slaves on the `AHB2`. Note that the Configuration Logic can operate in both modes: slave and master.

The stripe interface enables the communication between the stripe and the PLD of the Excalibur device and consists of the following interfaces: the DPRAM memory, the PLD-to-Stripe bridge and the Stripe-to-PLD bridge. The DPRAM memory can be used as an

shared memory interface with data access from both sides of the stripe interface. The PLD-to-Stripe and the Stripe-to-PLD bridges utilize the AHB bus standard [5]. As stated before, the PLD-to-Stripe bridge enables peripherals (masters) inside the PLD to access slave peripherals located on `AHB2`. An example of such a peripheral is a VGA engine, which reads data from a framebuffer located in the SDRAM. Access to the framebuffer is provided by the PLD-to-Stripe bridge, the `AHB2` bus and the SDRAM controller. The Stripe-to-PLD bridge enables the ARM processor to access slave peripherals located inside the PLD. Such a slave peripheral can be utilized by software and can perform computational intensive tasks in hardware, e.g., FFT.

In order to interface with external memory, the Excalibur device contains an Expansion Bus Interface (EBI) and an SDRAM interface. The SDRAM interface is capable of addressing 512Mbytes of SDRAM memory in 32-bit mode. The EBI interface supports various other memory types, e.g., Flash memory, SRAM, and memory mapped peripherals.

The Excalibur device family consists of the EPXA10, EPXA4 and EPXA1 device types, which all have different PLD sizes. Since the EPXA10 is expensive, when compared to the EPXA4 and EPXA1, it was decided that DAMP should be based on either the EPXA4 or the EPXA1. Table 1.1 shows a short overview of the main EXPAx capabilities. The ARM-Based Embedded Processor PLD's Hardware Reference Manual 2.0 [4] presents a complete overview of the capabilities of the EPXAx devices.

Feature	**EPXA1 device**	**EPXA4 device**
Max. system gates	263.000	1.052.000
Typical gates	100.000	400.000
Logic Elements	4.160	16.640
Embedded System Blocks	26	104
Maximum RAM bits	53,248	212.992
Maximum User I/O pins (PLD)	178	275
SRAM	32 kBytes	128 kBytes
DPRAM	16 kBytes	64 kBytes
Embedded Trace Module	No	ETM9
General Purpose I/O Port	4-bits	8-bits
Low-power PLL	Yes	No

Table 1.1: EPXAx capabilities

The EPXA4 and EPXA1 are available in various package versions. The EPXA4 is available in a 672-pin Fineline Ball Grid Array (FBGA) package and in a 1020-pin FBGA package. The EPXA1 is available in a 672-pin FBGA package and in a 484-pin FBGA package. The 672-pin versions of the EPXA4 and EPXA1 devices are pin-compatible. However, since the EPXA1 device contains less I/O pins than the EPXA4, there are more not-connected pins on the EPXA1 than on the EPXA4. It was decided to use the common 672-pin package, in order to support both devices with the same PCB design.

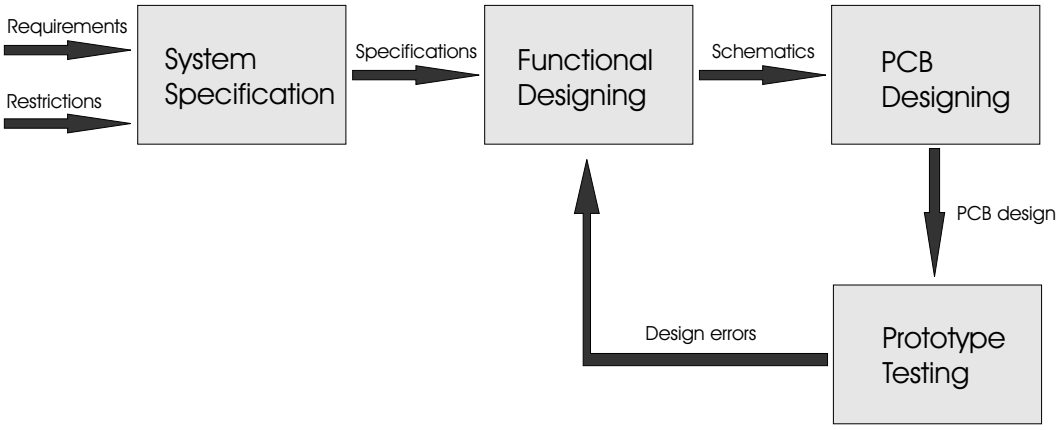

Figure 1.2: Overview of the DAMP Design Trajectory.

1.4 DAMP Design Trajectory

The main target of the DAMP project is to provide a low-cost development platform for the embedded systems specification and hardware-software co-design, with the main focus on (mobile) multimedia applications. In order to obtain the targeted result, a design trajectory (Figure 1.2) has to be followed. The DAMP Design Trajectory consists of the following steps:

1. A DAMP specification has to be defined. In order to acquire the DAMP specification, a thorough investigation of the requirements, restrictions, and of the design space is needed.

2. The DAMP hardware has to be designed. The design flow of the hardware design consists of the following sub-steps:

   - Design a functional representation of the system, i.e., develop a schematic design according to the DAMP specification.
   - Design a physical representation of the system by converting the schematic design into a Printed Circuit Board (PCB) design.

3. The DAMP design has to be verified, by testing a prototype. In order to do that, a testing methodology has to be developed and the test results have to be analyzed.

4. The DAMP hardware has to be redesigned according to the results of the verification step.

In order to support future DAMP developments, reference designs have to be developed. Such reference designs, together with a fully functional DAMP development board, can significantly speed up the development process and shorten the tools learning time.

1.5 Thesis Framework

The purpose of this thesis is to present an overview of the research and development work, concerning the DAMP project, done at the Computer Engineering Laboratory of the department of Electrical Engineering. This work is performed as part of a graduation project in the Computer Engineering Master of Science program at Delft University of Technology. The thesis organization is as follows:

Chapter 2 discusses the requirements, restrictions and the design space exploration used to acquire the DAMP specification.

Chapter 3 presents the testing methodology, which was developed to detect design errors. The resulting design modifications are also presented.

Chapter 4 describes the redesign process of the DAMP schematic and the DAMP PCB. The results of Chapter 3 are used as a guideline.

Chapter 5 discusses the reference designs, which are created in order to support future DAMP development.

Chapter 6 concludes the thesis, describes the main contributions and highlights future work directions.

2 DAMP Specification

Designing a complex multimedia platform for embedded systems development such as DAMP, cannot be done without a clear system specification. A thorough investigation of the requirements and restrictions together with a design space exploration are required, in order to acquire the DAMP specification. Acquiring the DAMP specification constitutes the subject of the present chapter.

This chapter is organized as follows. Section 2.1 discusses the requirements and restrictions that were taken into consideration. Section 2.2 presents the design space exploration process. Finally, in Section 2.3 an overview of the DAMP specification is presented, which is based on the decisions made during design space exploration.

2.1 Requirements and Restrictions

Before proceeding with the DAMP design space exploration, the main requirements and restrictions have to be considered. The requirements and restrictions form the base of the design space exploration.

Requirements

The DAMP design has to fulfill a number of requirements. These requirements are as follows:

1. **Multimedia:** DAMP is meant to be used as a platform for embedded systems development, mainly targeted on multimedia and mobile applications. DAMP should therefore support interfaces like video and audio, and should also provide communication options, such as UART or Ethernet. All these features require memory present on the development board, therefore DAMP should incorporate onboard memory.

2. **Excalibur compatibility:** DAMP should be compatible with all 672-pins versions of the Excalibur device family.

3. **Software support:** Currently only Altera provides design software that supports hardware and software development for the Excalibur device. Hence, DAMP should work gluelessly with the Quartus II development tools [7] and standard Altera download cables.

4. **Nios support:** DAMP is supposed to provide a natural Nios-to-Excalibur migration path for Nios customers extension boards.

5. **Testability:** DAMP should support a variety of testing capabilities. In order to accomplish this, some or all Excalibur pins should be connected to test pin-headers. DAMP should also support the debugging features of the Excalibur device. These debugging features can be addressed through standard Joint Test Action Group (JTAG [8]) or In-Circuit Emulator (ICE) interfaces.

Restrictions

The DAMP design process has one main restriction, namely that DAMP should be a low-cost development platform. The total cost of a DAMP PCB and the components should not exceed €1000. In order to keep the cost of the development board reasonable, the Printed Circuit Board (PCB) production cost should be minimal. Furthermore, the DAMP cost can be further decreased by reducing the component cost. The following factors influence PCB production cost:

- **PCB Stack-up:** The amount of copper used on each layer stack-up, the number of layers, the print thickness/size and layer base material.
- **PCB Traces:** The minimal/maximal trace width and number of traces used.
- **PCB Vias:** The amount, type and minimum hole size used for the vias.
- **PCB Surface finish:** The type of surface finish used for the top and bottom layer.
- **PCB Mask layers:** The optional use of a solder mask and paste mask.
- **PCB Silkscreen:** The optional use of silkscreen on one side or both sides of the PCB.

To reduce the component cost, circuits can be simplified. However, this decreases the number of services and the flexibility of these services. Some services can be taken over by units inside the PLD of the Excalibur device, which reduces the cost even more. However, DAMP is intended to be a platform for hardware/software co-design, which implies that the PLD utilization for such purpose should be limited. Ideally, the entire PLD should be available for the end user.

Because the cost of the Excalibur device is high, when compared with other components, the main focus should be on the reduction of the PCB cost. However, component cost and PLD utilization should still be taken in consideration.

2.2 Design Space Exploration

As stated before, DAMP is meant to be used as development platform for embedded systems, mainly targeted on multimedia and mobile applications. DAMP should contain an infrastructure, which is required by the Excalibur device and other devices in order to operate, namely power, clock and reset circuitry. Furthermore it should contain multimedia interfaces, such as video and audio. In order to communicate with other development boards, daughter cards or computers, various interfaces should also be part

of the DAMP design. To fulfill these requirements, a platform is proposed (Figure 2.1) which contains the following features:

- **Power, clock and reset circuit**
- **Connectors to access Excalibur's dedicated and I/O pins**
- **JTAG interface**
- **User I/O interface**
- **Flash memory**
- **SDRAM interface**
- **UART interface**
- **Video out**
- **High quality audio in/out** (stereo)
- **Ethernet support**
- **USB v2.0 support**
- **PS/2 interface**
- **IDE interface**
- **Optional Altera Apex FPGA** (socket)
- **Daughter card interface** (Nios daughter card interface compatible)

In the next subsections, different implementations of the proposed features are analyzed. A short evaluation of each possibility is presented in respect to cost and utilization of Excalibur resources when appropriate. Furthermore, for each and every feature, we present the implementation decision we've taken after the evaluation process.

Power, clock and reset circuit

The Excalibur device cannot operate in absence of proper power, clock and reset signals. Thus, circuits that provide these signals are to be implemented.

The Excalibur device requires two different voltage levels: 1.8V for the internal core and 3.3V for the I/O buffers, which connect the internal core to various I/O pins. Furthermore, it is likely that components are to be used which require voltage levels of 5V and 12V.

Related to the implementation of the power circuit, two options can be considered. First, an external Advanced Technology eXtended (ATX) power supply can be used and thus an ATX connector should be mounted on the development board. This gives the advantage that several voltage levels are present on the board and gives the flexibility of using standard widely available ATX power supplies. The second option is to use

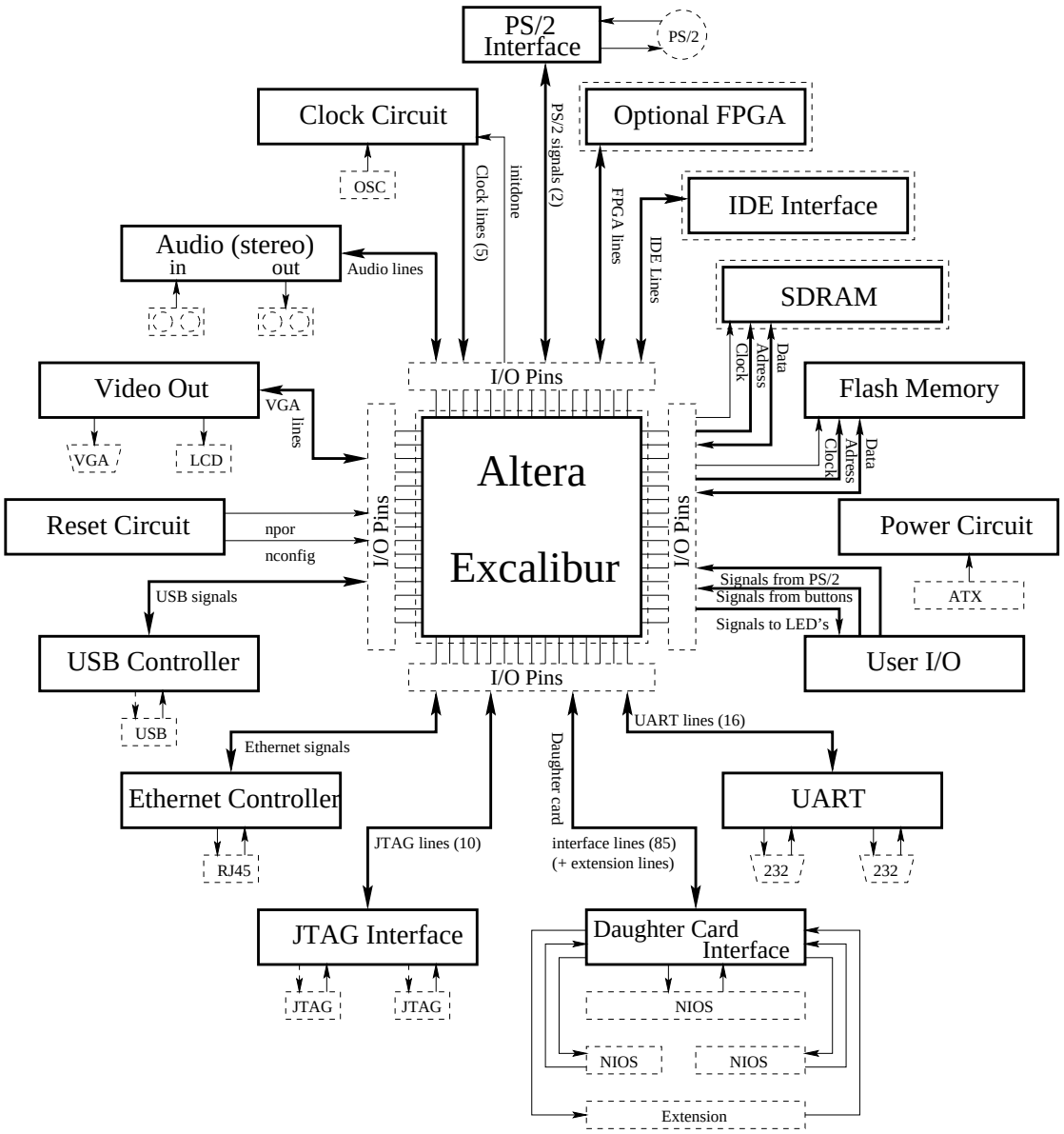

Figure 2.1: Proposed Features, Requirements and Signal-flow for DAMP board.

an external 5V or 9V power supply (not ATX) and several voltage regulators on the development board. This increases the complexity of the PCB design, however an external ATX power supply is not required.

Based on the previous discussion, the decision was made to implement the following. The Power Circuit should use an ATX power supply, which delivers most of the required voltages to the DAMP board. Since the ATX power supply doesn't provide

1.8V, an additional voltage regulator is required to generate this voltage on the board.

The Excalibur device features four clock inputs, which are connected to the Excalibur PLD part, and one reference clock input, which is connected to the stripe. This gives a total of five clock inputs. A clock circuit is required to provide the clock inputs of the Excalibur device with clock signals.

There are two solutions on how to implement the DAMP clock circuit. The first solution is to provide a single clock oscillator and a clock buffer. This reduces component cost and PCB space, however this solution cannot provide different clock frequencies at the Excalibur's clock inputs, hence all resources run with the same speed. The second option is to use a separate clock oscillator for each Excalibur clock input. This provides a high flexibility, however it increases the complexity of the design and the component cost. Nevertheless, by supplying various PLD units with different clock frequencies, one can achieve a better performance or a more efficient power usage. For example, a rather large computation takes more time than small computation and therefore requires to run at a lower clock frequency than the small computation. By increasing the frequency at which the small computation operates, one can achieve a higher performance. Decreasing the frequencies of both computations increases the efficiency of the power usage. Thus it becomes clear that the DAMP has to include more clock oscillators in order to be able to provide the means for power effective computation.

An additional feature for the clock circuit can be to provide the means for external clock inputs, in the form of an extra input pin. This requires the onboard clock oscillators to be disabled when this feature is utilized.

Based on the discussion about the above solutions, it was decided to use five separate clock oscillators. As stated before, four clock oscillators should be connected to the PLD of the Excalibur device and one should be connected to the stripe's clock input. Switches should also be incorporated to enable/disable the clock oscillators.

The Excalibur device requires an external power-on-reset signal, in order to perform a proper power-up sequence. The reset circuit should also contain a manual reset option, which functions independently of the power-on-reset signal. The external power-on-reset signal should be generated by a voltage monitoring component and should be connected to the power-on-reset input (`NPOR`) of the Excalibur device. In order to provide a manual reset option, a dedicated reset button should be used. This button is to be connected to the reset input (`NCONFIG`) of the Excalibur device.

Based on the previous arguments, the following was decided. The Reset Circuit should consist of a power-on-reset component, which generates the proper reset pulse to the Excalibur device after the required voltage levels are reached. Second a push-button has to be implemented to provide a manual reset option.

Connectors to access Excalibur's dedicated and I/O pins

In order to test the Excalibur device and its connected features, there should be an option to read, write and monitor all pin values of the Excalibur device. To accomplish this, pin headers should be implemented around the Excalibur device (Figure 2.1). The two possible options are:

1. Connect every ground, power, I/O and dedicated pin of the Excalibur device to a corresponding test pin.

2. Connect every I/O and dedicated pin of the Excalibur device to a corresponding test pin and connect the power and ground pins directly to the power distribution on the PCB.

The first option offers the highest flexibility, but requires a significant amount of test pins. Hence, this will increase the PCB space considerably. Furthermore, in order to connect all Excalibur pins to pin headers, additional signal layers are required and thus PCB production cost increases. The second option does not increase the cost, but then only the I/O and dedicated pins of the Excalibur device pins can be accessed, which reduces the testing capabilities.

Based on the discussion above, the following was decided. Only the dedicated and I/O pins of the Excalibur device are to be connected to test pins. To support this, twelve 2x20 pins headers and four 2x24 pins headers have to be mounted on the development board. The unconnected pins of the pin headers are to be connected to the power distribution on the PCB.

JTAG interface

The Excalibur device can be configured through the JTAG interface with the use of the Quartus II development tool. The Excalibur device provides JTAG Boundary Scan Test (BST) circuitry, that complies with the IEEE Std. 1149.1-1900 specification [4]. In addition to the internal test options, the JTAG BST circuitry provides access to external memory connected to the Excalibur device, e.g., Flash memory. The Excalibur device contains two JTAG Test Access Port (TAP) controllers. The Programmable Logic Device (PLD) TAP controller is used for boundary scan testing of the physical pins of Excalibur device and for downloading configuration data to the PLD. The TAP controller (ARM TAP) connected to the embedded ARM processor can be used for monitoring debug information from the embedded ARM processor. This includes processor registers information, stackpointer information and error detection capabilities.

The Excalibur device supports two JTAG configuration modes:

- **Serial JTAG programming mode:** In serial JTAG programming mode the PLD TAP controller and the ARM TAP controller are connected in single serial chain. Both controllers are accessible from one physical port and thus require only single connector.

- **Simultaneous JTAG programming mode:** In simultaneous JTAG programming mode the PLD TAP controller and the ARM TAP controller are connected to separate ports and thus require two connectors, which are presented in Figure 2.1. This mode makes it possible to simultaneously access both TAP controllers in realtime for debugging purposes.

Because one of the requirements is that the board should be compatible with Altera download cables, a MasterBlaster [9] compatible connector has to be used on the development board. The MasterBlaster should be connected to the Excalibur PLD TAP controller and utilizes JTAG boundary scanning. In order to support the JTAG simultaneous programming mode, a second 2x20 pins connector should be connected to the Excalibur device, which is directly connected to the ARM TAP controller. Furthermore it should still be possible to use JTAG serial programming mode through the use of the MasterBlaster connections. Thus two connectors are to be used on the development board.

User I/O interface

In order to support interaction with the user, an user interface is required. There are several options on how to implement this feature. One option is to use a touchscreen controller, which is advanced but costly. Another option, which is less complex but inexpensive, is to use a more standard approach that utilizes the following components:

- Push-buttons.
- Dipswitch bank.
- LEDs.
- 7-segment displays.

With simplicity and the low-cost restriction in mind, the following was decided. The User I/O implementation should consist of one 8-switch dipswitch bank, four push-buttons and two 7-segments displays. Furthermore, the User I/O should also include eight LEDs. All User I/O components have to be connected to the PLD I/O pins.

Flash memory

Flash memory devices have become the solution of choice in many applications, including mobile phones. The Altera Excalibur device provides a bidirectional Expansion Bus Interface (EBI), which is used as a bridge to external low-speed memory devices, such as Flash memory devices. Altera offers tools to program Flash memories through the JTAG interface. Furthermore the Excalibur device supports booting from external Flash memory.

The EBI of the Altera Excalibur device is supporting up to four external memory devices, each of them being up to 32 MBytes. The DAMP development board can also contain support for up to four devices, such that the end user can extend the amount of memory if desired. This provides maximum flexibility in Flash memory support, but

it increases design complexity and PCB cost. An alternative is to provide support for less Flash memory devices, which decreases the complexity and the cost. However, this also limits the flexibility.

The following was decided. Support of up to four Flash memory devices should be provided on the development board, in order to support a large storage space. To program the Flash devices with the Quartus II development tool, it is required that the Flash devices are to be compatible with the AMD command-set [10]. The Toshiba TC58FVB641FT-10 [11] NOR Flash device is to be used as a reference device, which has a capacity of 64Mbit and utilizes the AMD command-set.

SDRAM interface

The Excalibur device contains a certain amount of internal RAM in order to run software programs. The amount of memory inside depends on the Excalibur device type (Table 2.1). In many practical situations the on-chip memory present in either the EPXA1 or the EPXA4 devices is not sufficient. In order to support high-speed data transfers between a larger memory and the ARM processor or PLD, DAMP should support an SDRAM interface.

Device	SRAM Size	DPRAM Size
EPXA1	32 KBytes	16 KBytes
EPXA4	128 KBytes	64 KBytes

Table 2.1: Excalibur Internal RAM sizes.

There are two options available to implement an SDRAM controller:

1. Use a custom-made SDRAM controller implemented in the PLD.

2. Use the SDRAM controller integrated in the Excalibur device.

The first option gives maximum flexibility to the user in how to implement the SDRAM controller, such that different and faster implementations are feasible. The disadvantage here is that Excalibur I/O pins are required, when using the EPXA1. The EPXA4 device is able to use the pins connected to the integrated SDRAM controller as I/O pins, thus no extra pins are required.

The second option does not utilize the PLD. Furthermore the dedicated SDRAM pins from the Excalibur device can be used, thus no extra I/O pins are required. The integrated SDRAM controller supports Single Data Rate (SDR, 133 MHz) memory and Double Data Rate (DDR, 266 MHz) memory confirming to JEDEC [12] specifications. There are several options on how to implement SDRAM memory on the development board:

1. SDR memory devices mounted directly on the development board.

2. DDR memory devices mounted directly on the development board.

3. A 168-pin Dual Inline Memory Module (DIMM) socket mounted directly on the development board.

4. A combination of the above options: the use of memory and a DIMM socket on the development board.

The first two options have the advantage that there is always memory available onboard, however they increase the component cost of the development board. A major weakness for both is that the memory sizes are fixed and cannot be varied. The use of sockets solves this issue and gives the third option a significant advantage. The last option has all the advantages of the other options, but has also two disadvantages considering the low-cost restriction. First, it is the most expensive one in terms of component cost. Second, the PCB design complexity increases in terms of connecting the Excalibur to the SDRAM devices and matching SDRAM timing requirements.

Based on the previous discussion, the following was decided. An 168-pin DIMM socket is to be mounted on the DAMP development board in order to support SDRAM. This option provides a high flexibility, as different SDRAM DIMM modules can be utilized. The SDRAM controller only requires software configuration and initialization, which both are supported by the Quartus II development tool.

UART interface

DAMP should support a number of debugging and communication options. The Excalibur device has an integrated UART module, which allows serial communication with a terminal program running on a Personal Computer, that can be used to debug programs running on the embedded ARM processor for example.

The integrated UART module of the Excalibur device supports only one UART channel. More UART channels can be implemented by developing a PLD mapped UART controller. This can be used to interface with multiple devices that communicate via UART, such as GPS antennas and other development boards. However, extra Excalibur I/O pins and PLD resources have to be used.

The UART module is capable of functioning at the terminal end or at the modem end of the RS-232 link. Usually a null-modem cable is used for communication between two terminals, otherwise a normal serial cable is required. In case the UART is to act as a terminal device, a male connector is to be used on the DAMP development board, in order to support the null-modem cable option.

Based on the previous arguments, the following was decided. One RS-232 connector should be mounted on the development board and should be connected to the UART controller of the Excalibur device. A proper driver/receiver has to be used for voltage level conversion.

Video out

Video output is required for multimedia applications development. The following options for video out are considered (Figure 2.1):

1. A VGA connector on the development board, a simple resistor network for 8-bit DAC conversion and a simple 8-bit precision Red/Green/Blue (RGB) VGA controller unit mapped on the PLD of the Excalibur device.

2. A VGA connector on the development board, a 24-bit precision DAC (or multiple 8-bit precision DACs), and a 24-bit precision RGB VGA controller unit mapped on the PLD of the Excalibur device.

3. A third party LCD controller on the development board connected to the PLD.

An advantage of the first option is that only a simple VGA controller has to be implemented in the PLD of the Excalibur device. In addition only eight data signals (Red, Green and Blue) and two synchronization signals (`HSync` and `VSync`), and some resistor networks for simple DAC conversion are required. This solution provides 256 different colors, which is sufficient for most applications. Another advantage is that a standard VGA monitor can be used. A disadvantage here is that the VGA controller requires some PLD resources, which leaves less room for custom designs.

The main advantage of the second option is that it produces 24-bit precision colors on a standard PC monitor. However, it requires more I/O pins and occupies more space of the PLD of the Excalibur device, than the first option. This leaves even lesser room for custom designs. In addition, the component cost increases, because additional or more expensive external DACs are required.

The last option is intended for LCDs. This means that a standard PC monitor cannot be used. The use of an LCD controller increases the cost of the development board and requires a large amount of signals, when compared to the first option. The advantage here is that extra features of the LCD controller (e.g., integrated framebuffer) can be used. Furthermore only a small amount of PLD resources are required in order to communicate with the LCD controller.

The following was decided. For video support, a VGA connector is to be used on the development board. A hardware controller for 8-bit precision RGB VGA output has to be developed and programmed into the PLD. This solution only occupies a small amount of the PLD, when compared to the 24-bit precision RGB variant. Furthermore, it also reduces the cost, when compared to the use of a complete LCD controller on the DAMP development board.

High quality audio in/out

High quality audio input and output is required in order to develop multimedia applications. Audio support can be obtained via one of the following options:

1. Separate Analog to Digital Converters (ADCs) and Digital to Analog Converters (DACs) implemented on the development board.

2. ADCs and DACs implemented on optional daughter extension card.

3. Audio Coder-Decoder (CODEC) implemented on the development board.

4. Audio CODEC implemented on optional daughter extension card.

The advantage of the first two approaches is their simplicity. The audio can be processed by the ARM922T processor, if a simple driver is mapped on the PLD, or it can directly be processed by a dedicated engine, which is mapped on the PLD. This is, the PLD mapped unit can be used as a simple Digital Signal Processor (DSP). However most ADCs and DACs don't support more than 8-bit precision, while it is preferable to support state of the art audio quality (24-bit precision with 96kHz sampling rate). The disadvantage of using 24-bit DACs/ADCs is that they require 96 (stereo in/out) PLD device pins. The second option decreases the cost of the development board, as the audio daughter card becomes an optional extension to the DAMP mainboard, but requires a significant amount of daughter card interface pins. An advantage of the last two options is that audio CODECs often feature DSP functionality, thus higher performance can be achieved. An additional advantage is that a limited number of interface signals is required, when compared to the use of DACs and ADCs, because the audio is encoded into a serial bitstream. A disadvantage of the use of a CODEC, is that the Excalibur device has to contain a driver which takes care of the communication protocol between the driver unit and the CODEC. Such driver has to be implemented in the PLD of the Excalibur device, which decreases the amount of PLD resources remaining for custom designs. However, the driver occupies less PLD space than a complete DSP would as with the first two options. The fourth option is similar to the third one but additionally decreases the cost of the development board and requires five daughter card interface pins.

Based on the previous arguments, the following was decided. A Texas Instruments TLC320AD77C CODEC [13] is to be included on the DAMP development board, because of its low cost and high-quality (up to 24-bit precision with a sample rate of 96kHz). Furthermore only eight Excalibur I/O pins are required for data communication and device configuration. The CODEC is able to communicate via the following communication protocol standards:

- 16-bit, 20-bit and 24-bit MSB first, right/left justified.
- 16-bit, 20-bit and 24-bit I^2S.
- 16-bit MSB first, left justified/left justified.
- 16-bit DSP frame.

A detailed overview of the various standards can be found in the data sheet [13] of the TLC320AD77C. A hardware driver unit mapped on the PLD has to be developed to interface with the CODEC.

Ethernet support

In addition to all low-speed external communication options, DAMP should support an external interface which allow the transfer of large amounts of data at high speeds

(e.g., streaming video). Second, DAMP should support remote configuration and remote debugging of the Excalibur device. To provide these features industry standard Ethernet support is considered.

The Ethernet controller usually consists of a physical layer (PHY) and a protocol layer. The physical layer consists of a Manchester encoder/decoder and a peripheral that drives/receives 10BASE-T signals. The protocol layer processes all Ethernet frame data transactions (`TX` and `RX`). There are four different options to be considered in respect to the Ethernet controller:

1. Use a third party PHY controller on the development board, implement the protocol layer unit and map it on the PLD of the Excalibur device.
2. Use a third party chip that contains both layers on the development board.
3. Use a third party chip that contains both layers on an optional daughter card.
4. Combination of the first and third option, i.e., use a third party PHY chip on the development board and use a third party chip that contains both layers on a daughter card.

The first option is a low-cost solution and offers high flexibility. The user is free to create its own network interface or network controller for example. A disadvantage is that the controller implementation requires a significant amount of PLD space, which leaves less room for other functionality to the end user. The second and third option do not offer the flexibility of the first one, but occupy only a limited amount of PLD resources. This is for a driver unit mapped on the PLD of the Excalibur device. An advantage of the third option is that it decreases the cost of the development board, but it occupies a significant amount of daughter card interface pins. The fourth option is the most flexible, and leaves the final decision to the end user. It has the disadvantages of the first or the third option, as it requires a significant amount of PLD resources and it occupies the daughter card interface pins.

It was decided to use a Cirrus Logic CS8900A [14] controller, incorporating both Ethernet layers, because of its low cost. In order to support remote (via network) PLD reconfiguration, the Ethernet controller is to be connected to the EBI of the Excalibur device with the use of glue logic, thus a dedicated driver implemented in PLD logic is not required.

USB v2.0 support

USB, like Ethernet, provides a way of communicating with the development board at high speed and can be split up into two distinct layers: a physical layer and a protocol layer. Furthermore, USB controllers are available in two different types, host controllers and device controllers.

There are several options for the development board:

1. Use a third party USB v.2.0 host/device controller chip on the development board.

2. Use a third party USB v.2.0 transceiver chip on the development board and implement the protocol layer of the USB controller in the PLD of the Excalibur device.
3. Use a third party USB v.2.0 host/device controller chip on a daughter card.

The first two options require a considerable amount of Excalibur I/O pins. The first option occupies some PLD logic for a driver, which leaves less PLD resources for custom designs. The second option requires that the protocol layer is implemented as PLD logic in the Excalibur device, which decreases the amount of PLD resources even more. The second option however, is the most flexible solution of the two, since custom USB controllers can be implemented in the PLD of the Excalibur device. The third option leaves it to the designer to implement a controller on the daughter card (with or without the use of PLD logic). The advantage here is that the cost of the development board decreases. The disadvantage of this option is that it occupies the daughter card interface.

The decision was made not to implement a USB interface, but to leave it as a feature for a daughter card. This is due to the fact that the requirement of having a high-speed communication interface is already being fulfilled by the Ethernet controller.

PS/2 interface

Nowadays all computers can be controlled with the use of a PS/2 device (e.g., mouse or keyboard). DAMP should provide a PS/2 interface to the Excalibur device. Support for this interface (Figure 2.1) can be given by placing a PS/2 connector on the development board and by implementing the PS/2 controller in PLD logic on the Excalibur. Another option is to place a controller on the development board, which controls the PS/2 clock and data signals, however the cost of an extra component is large when compared to the limited amount of PLD resources required for the first option.

It was decided to implement the following. One PS/2 connector is to be mounted on the development board in order to support a mouse or PC keyboard. A PS/2 driver has to be designed and programmed into the PLD.

IDE interface

Nowadays most computers require devices which can store large amounts of data, e.g., CD-ROM players/recorders and harddisks. Most of these devices can also be used to store a file system, thus provide the means for an operating system stored on this device. In order to support these features, an IDE interface has to be implemented on the DAMP board.

This interface can be implemented by introducing a 40-pins header on the development board and by implementing an PLD mapped IDE controller unit. Another option for the IDE interface is the use of an external IDE controller on the development board, however this reduces the flexibility and increases component cost.

Based on the previous discussion, it was decided to provide support to IDE compatible devices. A 2x20 pins header is to be mounted on the development board and is

to be connected to Excalibur I/O pins. Furthermore, a PLD mapped IDE controller is to be developed. The IDE interface can only be used with the EPXA4 device, because the EPXA1 device features less I/O pins and thus some IDE interface connections are connected to pins of the EPXA1 device which are unused.

Optional Altera Apex FPGA

A very flexible way of adding extra functionality to the DAMP is to use an extra FPGA device. Using this FPGA device offers more room for user specific circuits, because both the PLD of the Excalibur device and the external FPGA can be reconfigured at run-time. There are several options:

1. Altera Apex FPGA mounted directly on the development board.

2. Altera Apex FPGA mounted on a daughter card.

3. FPGA socket for an Altera Apex FPGA mounted on the development board.

The first option increases the flexibility of the development board. However, it also increases the cost. The cost of the development board can be reduced by the second option, but this option increases the utilization of daughter card interface pins. The last option lets the user decide whether or not to use an extra FPGA device, so the cost of the development board increases only by a small amount.

The optional Altera Apex FPGA is not to be implemented on the DAMP development board, but is left as a feature for a daughter card. In this way the overall cost of the DAMP board is reduced. If more FPGA space is required, the first solution is to use the EPXA4 device, instead of the EPXA1. If the FPGA space of the EPXA4 is still not sufficient, the FPGA daughter card solution should be considered.

Daughter card interface

As stated in Section 2.1, one of the requirements is that the DAMP development board should be compatible with the daughter card interface of the Altera Nios development kit [3]. In addition to the Nios daughter card interface compatible part, an additional interface can be defined to support more complex daughter cards.

It was decided to implement three headers with pin-counts 14, 20 and 40, to ensure compatibility with the Nios daughter card interface. Furthermore, a single oscillator should be connected to the Nios daughter card interface, which provides the clock signal to the daughter card. In order to provide additional support to the developer of daughter cards, an extra DAMP extension header is to be implemented next to the Nios daughter card interface compatible part. This DAMP extension header consists of 40 extra pins, which connect to I/O pins of the PLD.

2.3 DAMP Specification

This section summarizes the DAMP specification. DAMP is based on the Altera Excalibur device, which is housed in a 672-pins FPGA package. Hence, DAMP is compatible with both the EPXA1 and EPXA4 type Excalibur devices. Furthermore, it was decided that DAMP should incorporate the following:

- **Power circuit:** An external ATX Power Supply and an 1.8V voltage regulator.
- **Clock circuit:** Five separate clock oscillators with enable/disable function (jumpers).
- **Reset circuit:** A reset button for a manual reset and a power-on-reset component.
- **Connectors to access Excalibur's dedicated and I/O pins**.
- **JTAG interface:** Two connectors to support serial JTAG programming and simultaneous JTAG programming.
- **User I/O interface:** Four push-buttons, one 8-switch dipswitch bank, eight LEDS and two 7-segment displays.
- **UART interface:** One line-convertor and RS-232 connector.
- **PS/2 interface:** One PS/2 connector.
- **Flash memory:** Support of up to four AMD compatible NOR Flash devices.
- **SDRAM interface:** 168-pins DIMM socket connected to the SDRAM controller of the Excalibur device.
- **High quality audio in/out (stereo):** TLC320AD77C Audio CODEC connected to the PLD.
- **Video out:** Support of up to 256 colors through a VGA connector and a simple DAC.
- **Ethernet support:** Cirus Logic CS8900A Ethernet controller connected to the EBI of the Excalibur device.
- **Nios daughter card interface:** Nios daughter card interface compatible part and an additional DAMP extension header.
- **IDE interface:** IDE header connected to the PLD of the Excalibur device.

Testing methodology, results and modifications

3

The last step of the DAMP Design Trajectory, is testing a prototype in order to verify its functionality and to improve the DAMP design. Hence, each DAMP feature should be tested thoroughly. In order to test the DAMP features correctly and efficiently, a testing methodology was developed.

This chapter is organized as follows. Section 3.1 presents a short overview of the DAMP testing process. The testing methodology, results and modifications of the various DAMP features are discussed in Section 3.2 through Section 3.14. Section 3.15 concludes this chapter by summarizing the results.

3.1 Introduction

Testing the Power, Clock, Reset and Excalibur configuration selector circuits before mounting the Excalibur device on the PCB is preferable, because these circuits constitute the PCB infrastructure. If the infrastructure does not function properly, the Excalibur device and all the other devices cannot function properly as well.

However, the Excalibur device is housed in an Fineline Ball Grid Array (FBGA) package, which was mounted on the DAMP PCB. Hence it was not possible to test the Power, Clock, Reset and Excalibur configuration selector circuits before the Excalibur device was mounted. This implies that the mounted Excalibur device formed the starting-point of the testing methodology.

One of the steps in verifying the DAMP prototype is a functional verification. During a functional verification, speed aspects are not considered. This implies that the Excalibur device with the lowest speed grade can be used, which reduces the component cost significantly. The DAMP design is supposed to be compatible with both the EPXA4 and the EPXA1. The PLD of the EPXA1 device has less logical gates (100,000) than the PLD of the EPXA4 device (400,000), which implies that the EPXA1 has a lower number of I/O pins than the EPXA4. However, they are both housed in the same 672-pin FBGA package, which results in a larger amount of unconnected pins on the EPXA1, compared to the EPXA4. This implies that the DAMP extension of the Daughter card interface and the IDE interface are partially or not connected when using the EPXA1, thus they cannot be tested. However, the other part of the Daughter card interface, the Nios daughter card interface compatible part, is connected entirely even when the EPXA1 is utilized. Furthermore, the EPXA1 can be used to test all the other features.

Based on the previous discussion, it was decided to use the Excalibur EPXA1 device with the lowest speed grade on the DAMP prototype. This reduced the cost of the prototype significantly, but implied that some parts could not be tested.

The Power, Clock, Reset and Excalibur configuration selector circuits are required by the Excalibur device, in order to function properly. This implies that these circuits

should be tested first. The JTAG interface comes second, because this interface is used to program and debug the Excalibur device and to program Flash memory devices.

The User I/O provides simple interaction with the user. Since interaction may be required by other tests, the User I/O should be tested as third.

When the Altera Quartus II development tool is used to compile the VHDL and C/C++ program files, Quartus links a bootloader object file to the other files. This process generates a bootloader, which can be programmed into the Flash memory. On start-up the Excalibur device reads and executes the bootloader program from Flash memory. The bootloader then programs the PLD, configures the stripe and starts the software program. Using the Flash memory offers a large storage space for user programs and provides the possibility to initialize the SDRAM controller during the boot process. These features can be used to test some other features, therefore the Flash memory should be tested as fourth.

The Excalibur device features on-chip SRAM: 32KB on the EPXA1 and 128KB on the EPXA4. Most applications however, require a larger memory space. The applications that are used to test the other features might use a larger memory space than the internal SRAM offers, therefore SDRAM should be tested as fifth.

Some applications require interaction with the user. Using the User I/O is not always sufficient. This can be solved by using a terminal program and a serial connection to DAMP via the UART interface. Because the applications that are used to test the various features might require interaction, the UART interface should be implemented as sixth.

DAMP is intended to be used a multimedia development platform. Hence both VGA and Audio should be tested, but the order in which VGA and Audio are tested is of no importance. The order in which the PS/2 and Daughter card interface are tested is also not relevant.

The order in which the various features were tested, can be summarized as follows:

1. Power circuit.
2. Clock circuit
3. Reset circuit.
4. Excalibur configuration selector circuits.
5. JTAG interface.
6. User I/O.
7. Flash memory.
8. SDRAM.
9. UART interface.
10. VGA.
11. Audio.

12. PS/2
13. Daughter card interface (Nios compatible part only).

These 13 steps provide a guideline, which was used to test the various parts and services of the DAMP design. However, some parts of the DAMP design were not tested. These parts are:

1. DAMP Extension of the Daughter card interface
2. IDE interface
3. Ethernet facility

Furthermore, a speed verification was not performed. Thus future work should incorporate a speed verification and a test of the untested parts of DAMP.

3.2 Power circuit

The Power circuit mainly consists of passive components, which are all located at the bottom of the DAMP PCB. Furthermore, it consists of an ATX power supply connector, a LED, a switch and the PT6520 voltage regulator. In Figure 3.1 the location of the Power circuit components is presented.

According to the ATX specifications [15], one of the requirements of an ATX power supply is that a 2.5W load has to be attached to the power supply in order to start up. This requirement is met by using the WH25 high power resistor [16].

If the ATX power supply starts up, the `POWER OK` LED should light up. The signal that drives the `POWER OK` LED is generated by the ATX power supply and indicates that the 5V and 3.3V voltage levels are within the tolerated threshold margins. This start up process should be tested first, before performing all other measurements.

The Power circuit supplies four different voltages: 12V, 5V, 3.3V and 1.8V. The 12V, 5V and 3.3V voltages are supplied directly by the external ATX power supply. The 1.8V voltage is supplied by the PT6520 voltage regulator. The PT6520 voltage regulator requires a supply voltage of 3.3V. The voltage levels at the outputs of the ATX power supply connector and the in- and outputs of the PT6520 voltage regulator should be measured second.

Measuring the voltages near the ATX connector and near the PT6520 voltage regulator does not guarantee that the entire Power circuit is correct. Fabrication flaws can be the cause of errors. Measuring the voltage levels and ground plane connections of predefined test points in the four corners of the DAMP PCB, ensures an even power distribution of the power plane. If the alignment is correct, it can be assumed that the Power circuit contains no fabrication flaws. The predefined test points are located near the WH25 high power resistor, near the UART connector, near the Daughter card interface and near the PS/2 connector.

Based on the previous discussion, it was decided to test the Power circuit in the following order:

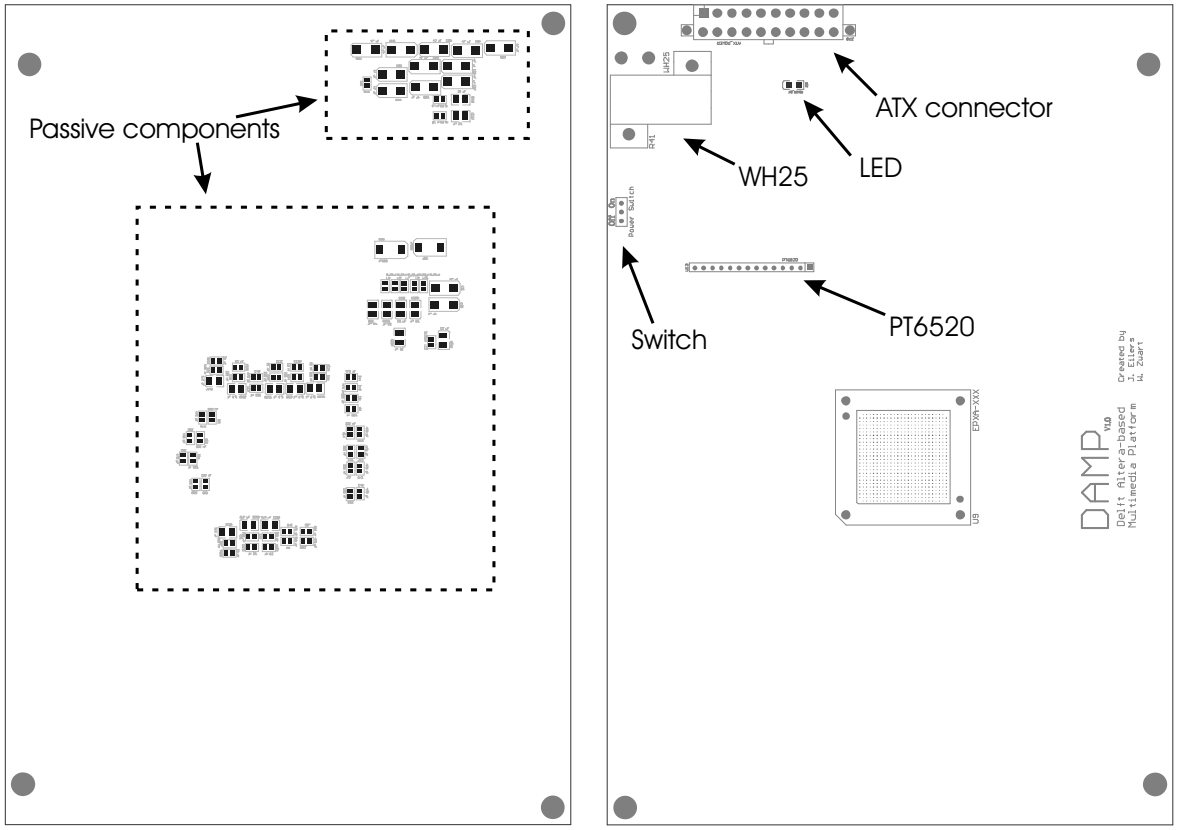

Figure 3.1: Location of the Power circuit components.

1. ATX power supply startup test, with the WH25 high power resistor attached.
2. Checking the `POWER OK` LED.
3. Measurements of the voltage levels and ground plane connections near the ATX connector:
   - 12V voltage levels.
   - 5V voltage levels.
   - 3.3V voltage levels.
   - Ground plane connections.
4. Measurements of the voltage levels and ground plane connections near the PT6520 voltage regulator:
   - 3.3V input voltage levels.
   - 1.8V output voltage levels.
   - Ground plane connections.

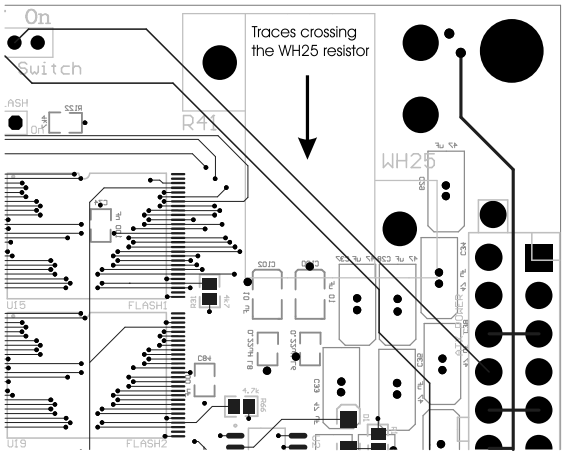

Figure 3.2: Footprint of the WH25 high power resistor on the DAMP PCB.

5. Measurement of the voltage levels and ground plane connections near the WH25 power resistor:
   - 5V voltage level of the WH25 power resistor.
   - Ground plane connection of the WH25 power resistor.
6. Measurement of the voltage levels and ground plane connections near the UART connector:
   - 3.3V voltage level of resistor `R2`.
   - Ground plane connection of pin 5 of the UART connector.
7. Measurement of the voltage levels and ground plane connections near the Daughter card interface:
   - 3.3V voltage level of pin 4 of clock oscillator `U6`.
   - Ground plane connection of pin 2 of clock oscillator `U6`
8. Measurement of the voltage levels and ground plane connections near the PS/2 connector:
   - 5V voltage level of pin 4 of the PS/2 connector.
   - Ground plane connection of pin 3 of the PS/2 connector.

During the construction of the Power circuit, it became clear that traces are routed through the footprint of the WH25 high power resistor. Furthermore, vias are inside the footprint. Figure 3.2 presents the footprint of the WH25 resistor on the DAMP PCB. Because these traces and vias can cause a short circuit with the WH25, it was decided to leave the WH25 unmounted. However, wires were used to connect the WH25 to the PCB.

The ATX power supply specifications [15] clearly states the restrictions and requirements which the manufacturer should follow. The results of the first test proved that most ATX power supplies on the market do not comply with these specifications:

- ATX power supplies indeed require a load in order to start up.
- The behavior of the `POWER OK` LED depends on the model of the ATX power supply. If the load of the WH25 power resistor is not sufficient, the LED blinks. If it is sufficient, the `POWER OK` LED burns constantly.

The other results of the followed testing methodology were:

- The voltage levels and ground plane connections near the ATX power connector are correct.
- The voltage levels and ground plane connections near the PT6520 voltage regulator are correct.
- The voltage levels and ground plane connections of the predefined points in the four corners of the PCB are correct.
- The silkscreen that indicates the `On` position of the power switch indicates the `Off` position and visa versa.

3.3 Clock circuit

The Clock circuit consists of five clock oscillators and their enable/disable jumpers. Furthermore the Clock circuit consists of a 1kΩ resistor and a connector for an external clock input. The resistor is connected to the `CLKLK_ENA` pin and enables the PLL's of the Excalibur device. One of the oscillators is connected to the dedicated clock input of the stripe of the Excalibur device. The other four clock oscillators are connected to the dedicated clock inputs of the PLD of the Excalibur device. A socket was used, such that the 25MHz clock oscillators can be easily interchanged with clock oscillators with a different frequency. The location of the Clock circuit components is presented in Figure 3.3.

Testing the Clock circuit was straightforward. The clock oscillators can be enabled or disabled by changing the setting of the jumpers that are part of the Clock circuit. The output of each clock oscillator was measured, with the clock oscillator in its enabled and disabled state. The clock signals of the other components are not part of the Clock circuit. Most clock signals however, are generated by the Excalibur device. Because the clock frequencies required by these components are relatively low, clock skew due to propagation delay is not an issue. The SDRAM interface is an exception.

The results of the followed testing methodology were:

- All clock oscillators function correctly in their enabled state. The output pins of the clock oscillators are high impedance connections in the disabled state of the clock oscillators.

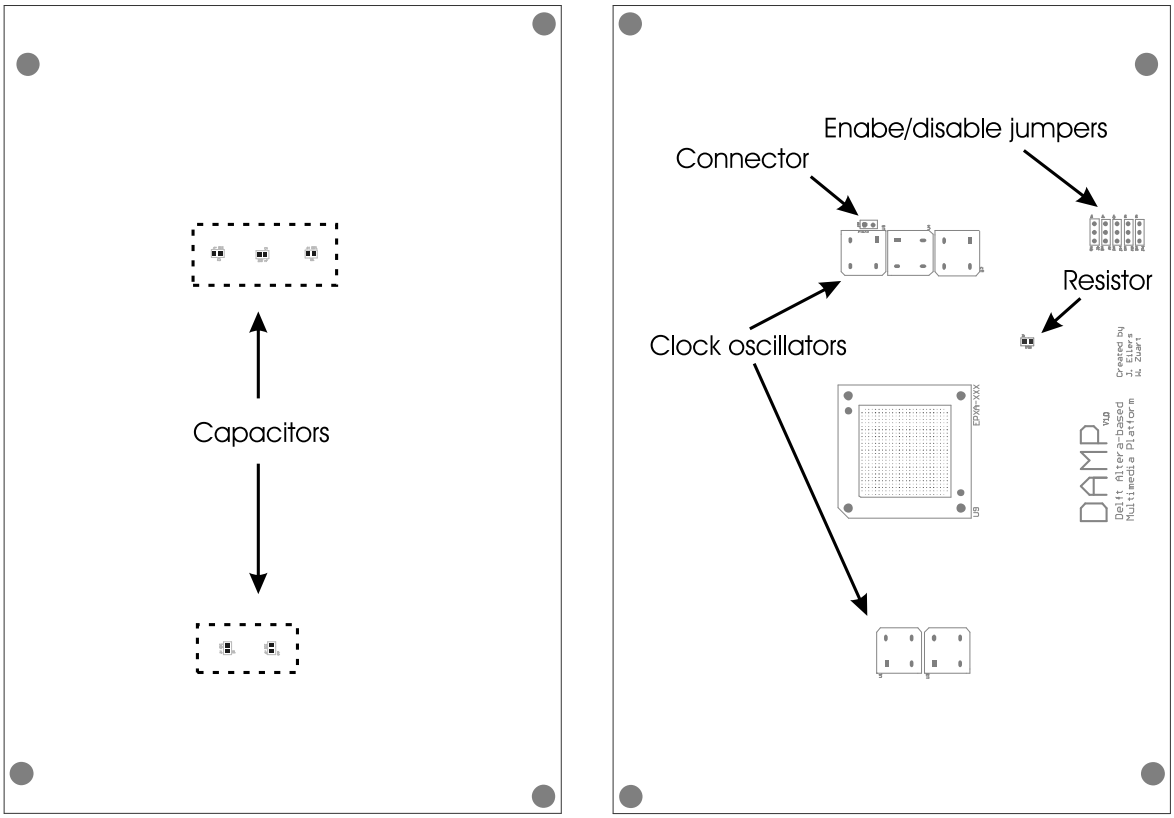

Figure 3.3: Location of the Clock circuit components.

- The silkscreen that indicates the `On` position of the jumpers indicates the `Off` position an visa versa.

3.4 Reset circuit

The Reset circuit only contains a button, an LM3722 power-on-reset device, a diode and some passive components. Figure 3.4 presents the location of these components.

There are two mechanisms in the Reset circuit that can generate a reset signal: the push-button and the LM3722 power-on-reset device. In order to test the Reset circuit, these mechanisms were tested.

The first mechanism was tested by measuring the output of the LM3722 when the ATX power supply started up. The LM3722 drives the `npor` pin of the Excalibur device. The reset signal on this pin should be low until all voltage levels are within the tolerated margins. Additionally, once these conditions are met, five additional clock cycles must elapse before the reset signal becomes high. The other mechanism is tested by measuring the output of the push-button after pressing it and observing the behavior of the Excalibur device.

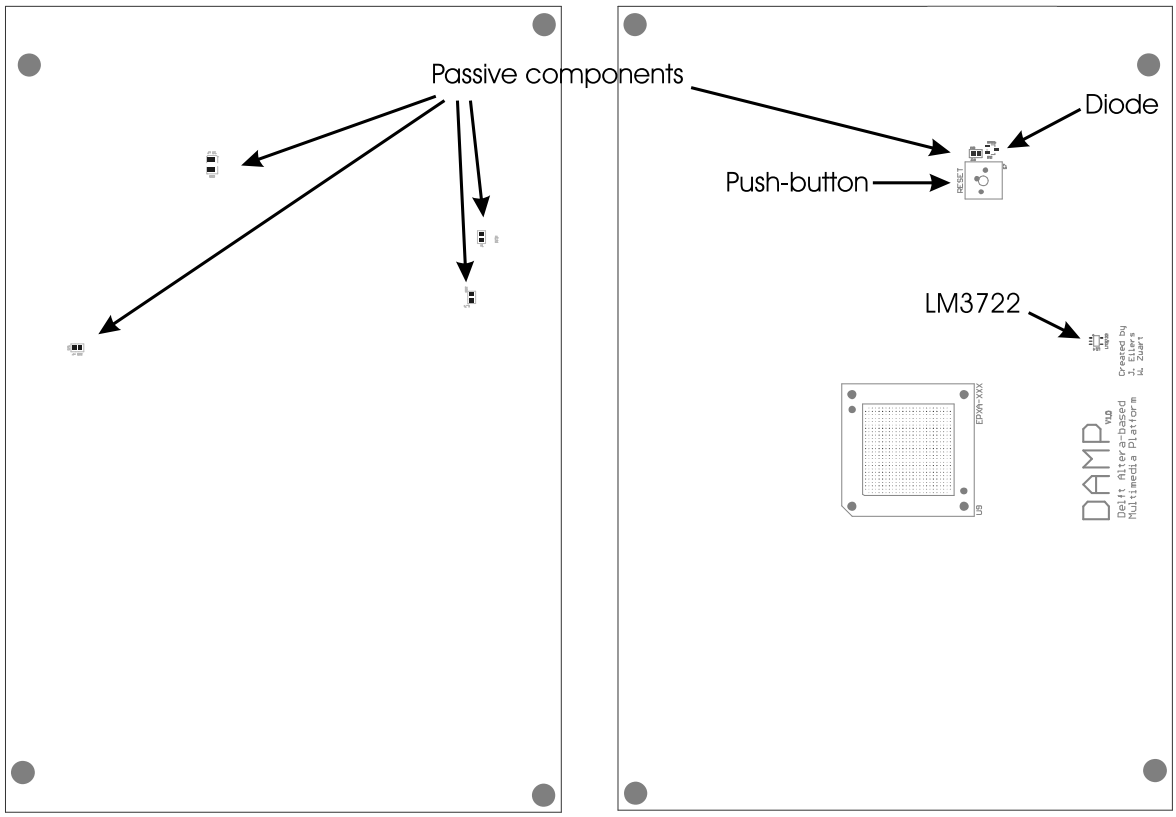

Figure 3.4: Location of the Reset circuit components.

The results of the followed testing methodology were:

- The LM3722 generated a reset signal on power up. The duration of the reset pulse was 190 ms, which is more than the required five clock cycles.

- The output of the reset push-button was correct and no jitter was detected, which indicated that the debounce circuit was behaving correct. The Excalibur device is reset by pressing the reset button.

During the construction of the Reset circuit, it became clear that capacitor C98, which is used in the debounce circuit, has a different value (10μF) than the capacitors used in the debounce circuit of the push-buttons of the User I/O. Because of this, a large current flows through the reset button when pressed. This could damage the button. Hence, capacitor C98 should be replaced by a capacitor with a smaller value. The value of this capacitor can have the same value (100nF) as the capacitors used in the debounce circuit of the push-buttons in the User I/O circuit.

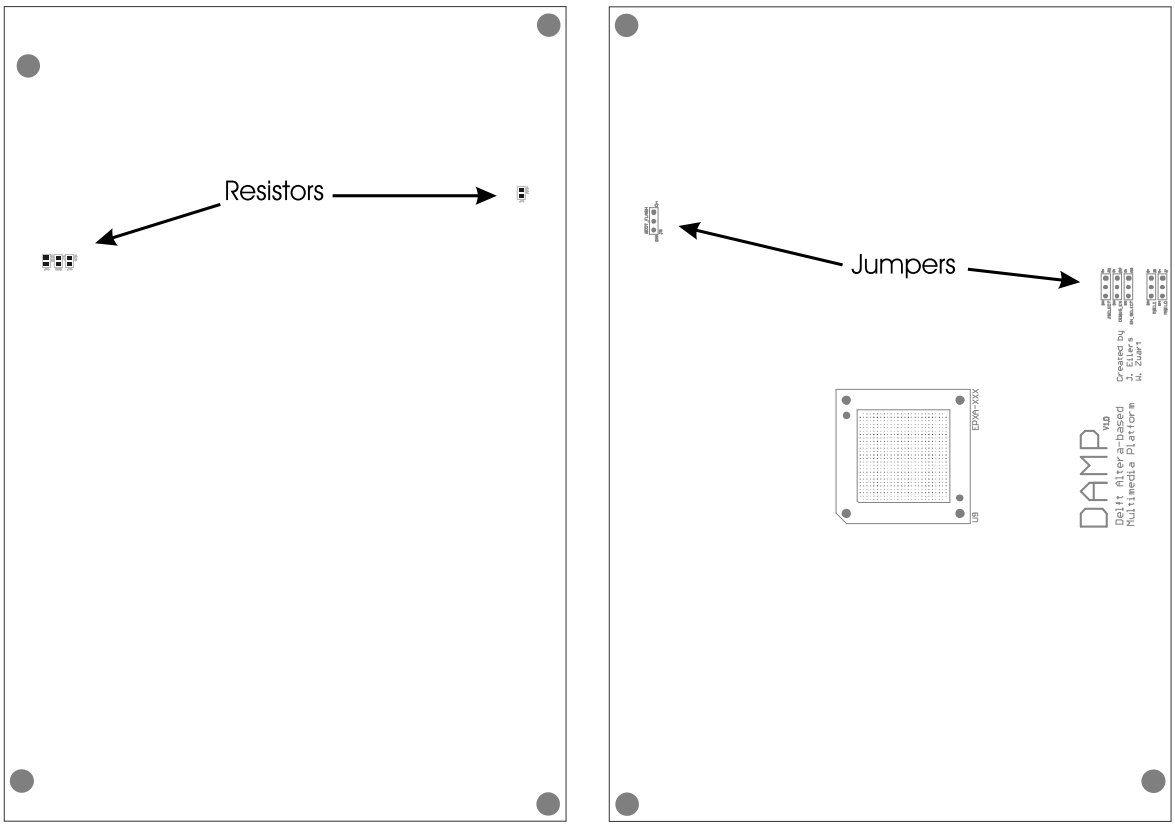

Figure 3.5: Location of the Excalibur configuration selector circuit components.

3.5 Excalibur configuration selector circuit

Figure 3.5 presents the location of the Excalibur configuration selector circuit components. The circuit only consists of jumpers and a few resistors.

The jumpers that are part of the Excalibur configuration selector circuit, enable the various configuration modes used for booting the Excalibur device. Furthermore, a jumper is used to select either the Serial JTAG programming mode or the Simultaneous JTAG programming mode. The `EN_SELECT` jumper can provide a ground connection or a 3.3V voltage level to the corresponding Excalibur pin. This pin is reserved for future use and should be connected to ground. The pins of the Excalibur configuration selector circuit are either connected to the ground plane, connected directly to the 3.3V voltage level or connected to the 3.3V voltage level via a pull-up resistor. The pins that are connected via a pull-up resistor were tested to ensure that the Excalibur configuration selector circuit can properly setup the programming mode.

The results of the followed testing methodology indicated that the voltage levels of the pins on the `On` side (pull-up resistor side) of the `BOOT_FLASH`, `JSELECT`, `DEBUG_EN` and the `EN_SELECT` jumpers were correct.

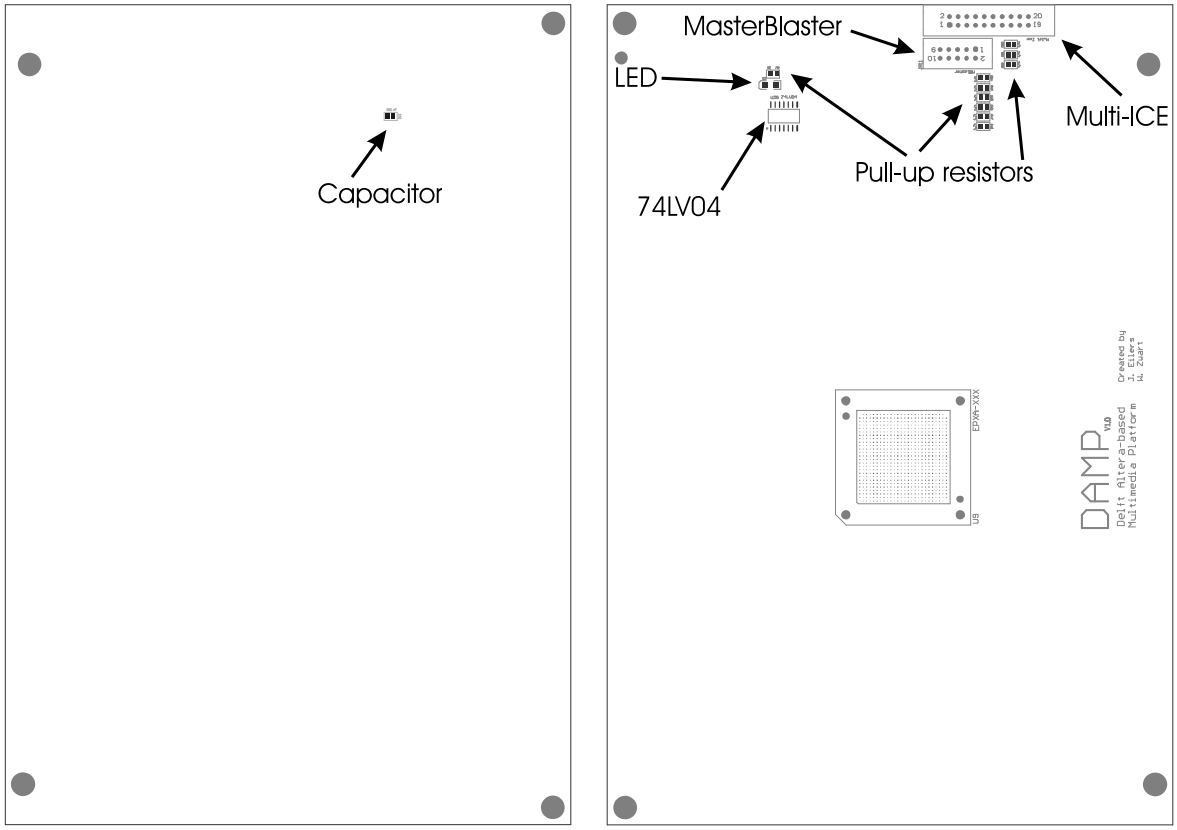

Figure 3.6: Location of the JTAG interface components.

3.6 JTAG interface

The JTAG circuit consists of a few pull-up resistors, a capacitor and an invertor, which inverts the `INIT_DONE` signal and drives a LED. Furthermore, the JTAG circuit contains a MasterBlaster and a Multi-ICE connector. The location of the components is presented in Figure 3.6.

Even though the JTAG interface contains both a Multi-ICE and a MasterBlaster connector, only the MasterBlaster connector was tested. The reason for this is that in order to test the Multi-ICE connector, extra third-party hardware had to be acquired. Because of the limited budget, no Multi-ICE hardware was acquired.

The Excalibur device contains two parts: a Programmable Logic Device and a stripe. The PLD contains FPGA based hardware. The stripe of the Excalibur device contains the ARM processor and some common interfaces, e.g., UART of SDRAM, SRAM, Dual port SRAM (DPRAM). Furthermore the stripe contains Configuration logic, which communicates between a configuration source and the PLD of the Excalibur device. The Configuration logic also initializes the internal registers and SRAM. DAMP uses the Altera ByteBlaster cable as a configuration source, which has the same connector as the MasterBlaster.

In order to test the JTAG interface the Altera ByteBlaster cable was connected to DAMP and the Altera Quartus II development tool was used to perform the following actions:

1. The `Auto Detect` function of the programmer was used to probe the JTAG chain.
2. The PLD of the Excalibur device was programmed with an SRAM Object File (SOF).

The SOF file of the first User I/O testbench (see Section 3.7) was used to program the Excalibur device. The Quartus II development tool indicates whether programming was successful or not. After programming the Excalibur device, it initializes itself. During this initialization phase, the `INIT_DONE` signal is low which deactivates the `INIT_DONE` LED. If the initialization was executed successfully, the `INIT_DONE` signal activates the `INIT_DONE` LED. This behavior was also tested.

The results of the followed testing methodology were:

- The Quartus II development tool recognized the two devices in the JTAG chain. These devices are the configuration logic master and the ARM processor of the EPXA1 Excalibur device.
- The User I/O SOF file was successfully programmed into the Excalibur device. A screenshot of this is presented in Figure 3.7.
- The behavior of the `INIT_DONE` LED was as expected.

3.7 User I/O

The User I/O section of DAMP consists of four push-buttons, a 74LV14 hex invertor, a dipswitch bank, eight LEDs and two 7-segment displays. Figure 3.8 presents the location of the User I/O components.

In order to test the User I/O, a testbench had to be designed, which can be programmed in the PLD of the Excalibur device. This testbench can then be used to do a steady-state analysis of the following:

- Push-buttons.
- LEDs.
- 7-segment displays.
- Dipswitch bank.

Two testbenches were implemented. The first testbench only uses the User I/O circuit and the PLD of the Excalibur device. The second testbench also consists of a program that runs on the ARM processor in the stripe of the Excalibur device.

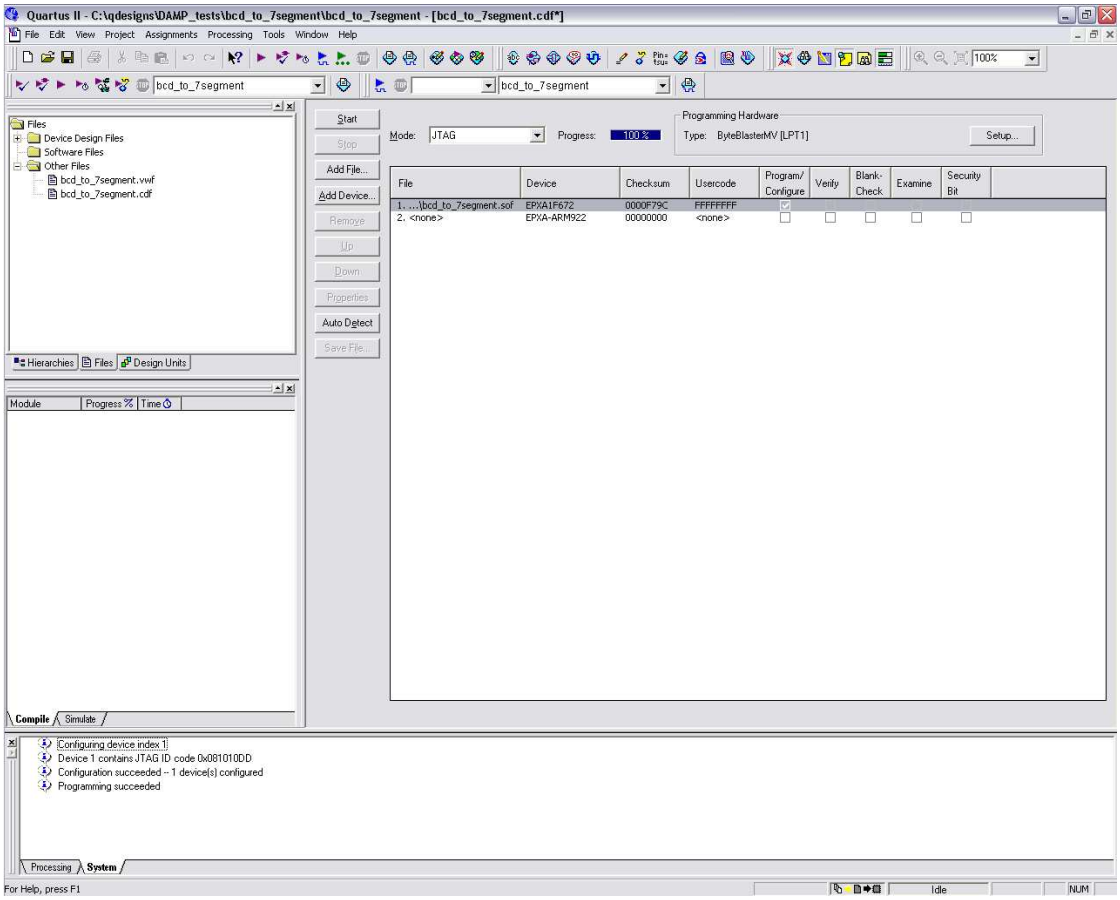

Figure 3.7: Screenshot of the Quartus II development tool, which has programmed the Excalibur device via JTAG.

The first testbench displays a number on the 7-segment displays and is able to drive the LEDs. The dipswitch bank is used as an input for the binary number to 7-segment display conversion. This conversion only uses four switches of the dipswitch bank, which means that numbers between 0 and 15 can be displayed. The other four switches and the four push-buttons are used to drive the eight LEDs. Note that both 7-segment displays display the same number and that they display the least significant digit of the decimal number. The dots of the 7-segment displays indicate the most significant digit, which is one or zero. Figure 3.9 presents the schematic of the first testbench. The file listing of the `decoder.vhd` file can be found in Appendix A.

The second testbench is actually a program written in assembly, which writes certain values to a predefined address of the DPRAM of the Excalibur device. A simple device, which is located in the PLD of the Excalibur device, is connected to the internal DPRAM interface of the Excalibur device and the LEDs of the User I/O. This device reads the value from the predefined address of the DPRAM and uses it to drive the LEDs. The predefined address which is used to interface between the software

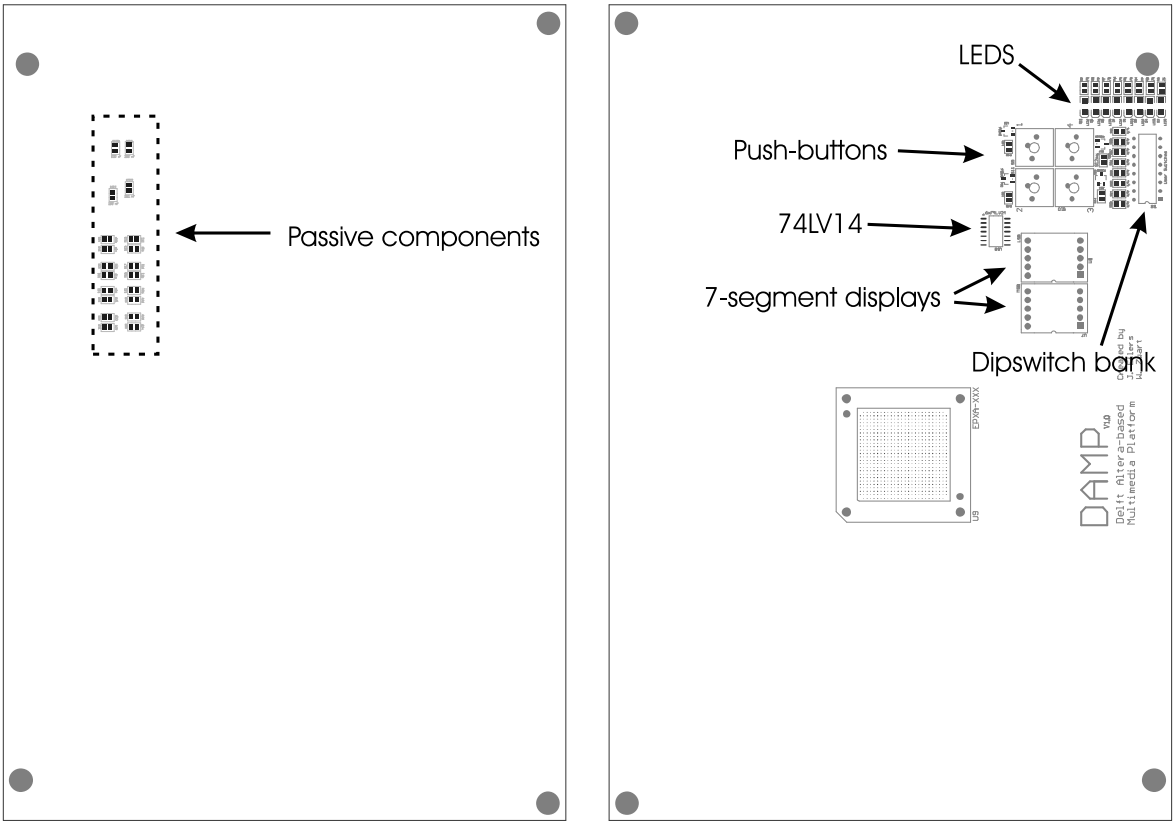

Figure 3.8: Location of the User I/O components.

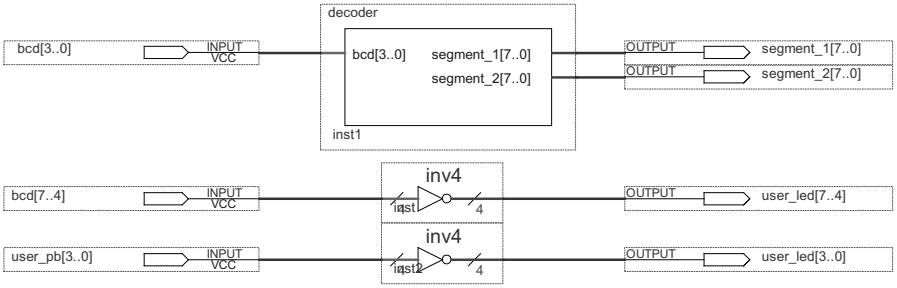

Figure 3.9: Schematic of the first User I/O testbench.

and the hardware, can be any arbitrary value, but it was decided to use address zero. The testbench merely consists of connections to I/O pins and some internal ground connections. The address bus of the DPRAM interface is connected to ground, which results in the selection of address zero. The write enable and input data bus are also connected to ground. The output data bus is connected to the LEDs. Figure 3.10 presents the second testbench. The assembly program file (arm_pld.s) can be found in

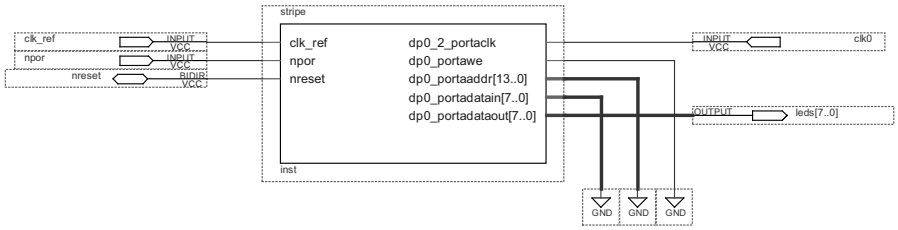

Figure 3.10: Schematic of the second User I/O testbench.

Appendix A.

During the construction of the User I/O circuit, it became clear that the 74LV14 hex invertor has no 100nF decoupling capacitor attached to its voltage supply, which implies that transitions inside the invertor are not filtered and can thus be seen on the 3.3V voltage level. No modification was performed, however the design flaw was corrected by redesigning the schematics and PCB, which is discussed in Chapter 4. The results of the followed testing methodology were:

1. Results of the first testbench:

   - The push-buttons functioned correctly. The silkscreen that indicated button 2 actually indicated button 3, and visa versa.
   - The LEDs functioned correctly.
   - The 7-segment displays functioned correctly.
   - The dipswitch bank functioned correctly, but the enabled position (`On`) of the dipswitch bank actually was the disabled position.

2. Results of the second testbench:

   - The second testbench functioned correctly, which indicated that the program running on the ARM processor was writing data to the correct location and that the design within PLD of the Excalibur device was reading the data from this location.

3.8 Flash memory

The Flash memory circuit consists of up to four NOR Flash memory devices, some pull-up resistors and buffer capacitors, and a jumper that selects the bus width. Figure 3.11 presents the location of the components.

Even though DAMP offers supports for up to four Flash memory devices, only one Flash memory device was used on the DAMP prototype in order to test the Flash memory circuit. A Flash programming tool is required in order to program the Flash memory device. Altera supplies such a tool with the Quartus II development tool. The

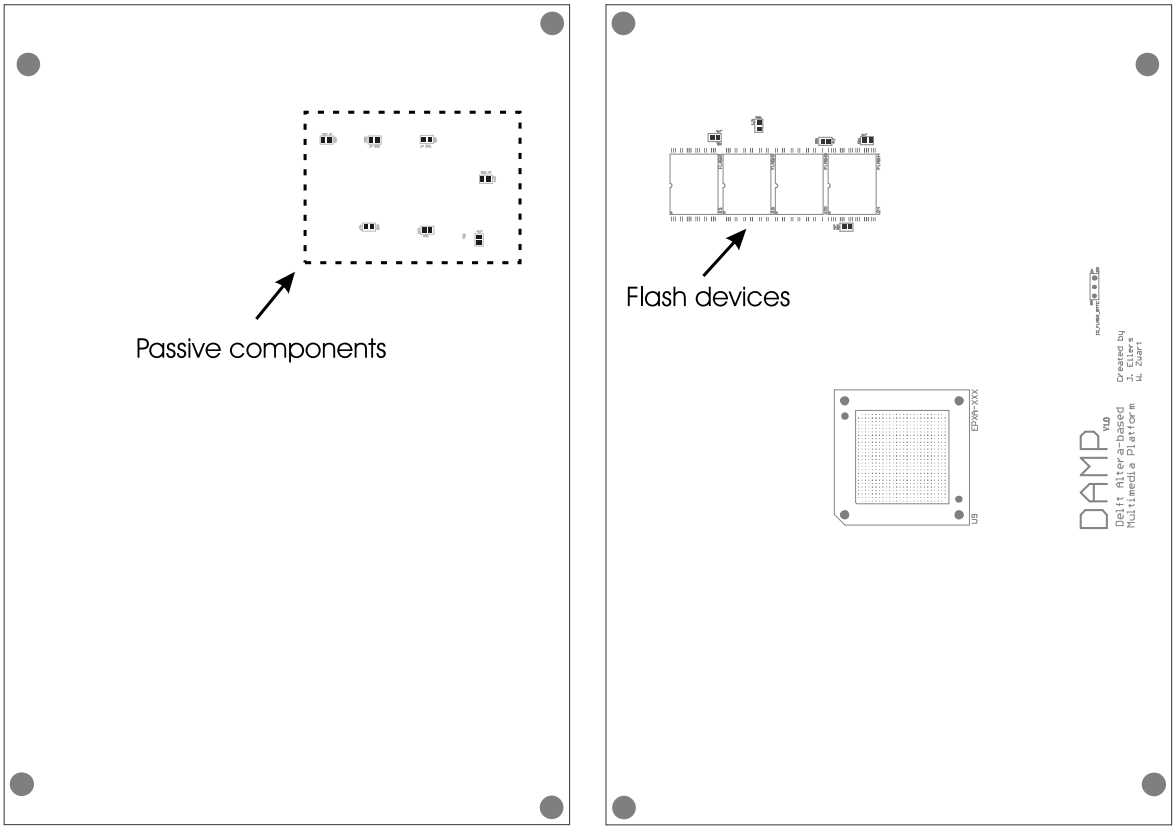

Figure 3.11: Location of the Flash memory circuit components.

`exc_flash_programmer` tool can be used to program, read and erase the Flash memory devices that are connected to the Excalibur device.

Testing the Flash memory circuit was done by reading from the Flash memory device, because then every data and address line that is connected to the Flash memory device is used. If one of the data or address lines fails, the `exc_flash_programmer` tool notifies the address on which the error occurs. In order to test the Flash memory circuit, the Flash memory device must contain predefined data. This was achieved by erasing the Flash memory device prior to the reading process.

Finally, booting from Flash memory was tested. The Quartus II tool can be configured to generate a Flash programmable bootloader or an SOF file when compiling the VHDL and C/C++ files of the testbench. If the Excalibur device starts up and the `BOOT_FLASH` jumper is set correctly, the Excalibur reads and loads the bootloader data from Flash memory. This bootloader will then configure the stripe and program the PLD of the Excalibur device. Since a bootloader was required, Quartus II was configured accordingly. In order to test the booting from the Flash memory, the second User I/O testbench was compiled with the Quartus II tool, which was configured to generate a bootloader. This bootloader was programmed into Flash memory and was used to test the boot process.

After successfully assembling the Flash memory circuit, the `exc_flash_programmer` tool was used in order to erase the device. However, this resulted in an error message, which noted that the device could not be recognized. Investigating the schematic and PCB design initially gave no result. An assumption was made that compatibility issues might create the problem. In order to solve the compatibility issues, it was decided to use the same AMD Flash devices that Altera uses on their EPXA1 development board. After mounting the AMD Am29DL322D Flash devices the `exc_flash_programmer` generated the same error.

After another examination of Altera application note 142 [17], about using the EBI of the Excalibur device with Flash devices, and the DAMP schematics, it became clear that the Flash device was incorrectly connected to the EBI of the Excalibur device. The address lines of the EBI should be shifted up one position with respect to the address lines of the Flash memory devices, such that the EBI address line `EBI_A1` is connected to the Flash memory address line `A0`.

Because of this design flaw, the Flash memory footprints on DAMP were useless. However Flash memory support was still desirable. Therefore the decision was made to use a separate PCB that contains the appropriate footprint and breakout header connections, which are connected to this footprint. A Flash memory device was mounted on this PCB and wires were used to connect the breakout header connections to the DAMP PCB.

The results after applying the modification were:

- The `exc_flash_programmer` was able to erase the Flash memory device successfully.
- The data was read from the Flash memory device successfully.
- The contents of the data was correct.
- The testbench was successfully programmed into the Flash memory.
- The Excalibur device was able to boot from Flash memory with the `BOOT_FLASH` jumper enabled (`On`).

3.9 SDRAM

DAMP supports 168-pins SDRAM DIMM modules. The SDRAM circuit therefore consists of a DIMM socket. Furthermore the SDRAM circuit consists of a CY2305 clock buffer, some resistors and a capacitor. The location of the components is presented in Figure 3.12.

A testbench was designed in order to test the SDRAM interface. Every data and address line had to be tested, thus the testbench writes the write addresses to the appropriate memory locations and verifies the contents of the memory locations afterwards. Figure 3.14 presents the schematic of the testbench. The file listings can be found in Appendix E.

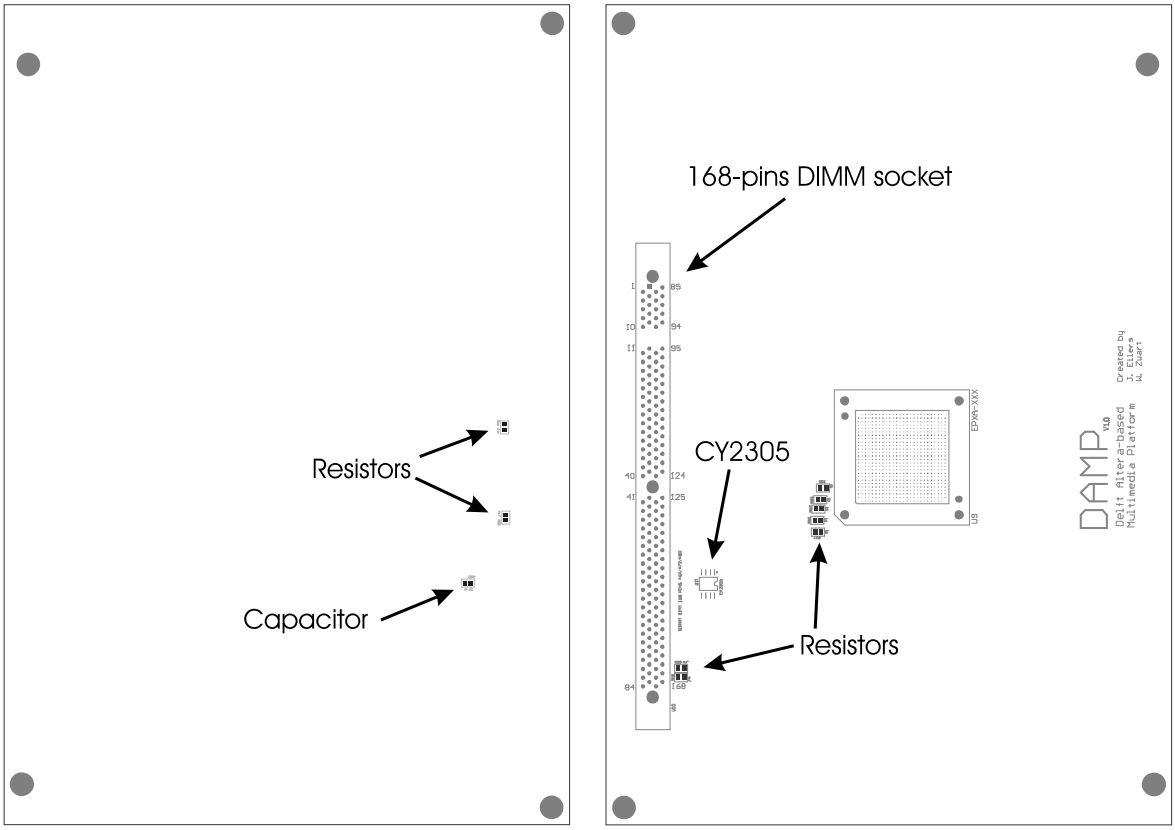

Figure 3.12: Location of the SDRAM circuit components.

The testbench generates output on the 7-segment displays. Initially the 7-segment displays display two small circles. If the test completes successfully, the circles rotate clockwise. If the test is not successful, an error mark appears on the 7-segment displays. Figure 3.13 presents the behavior of the 7-segment displays during the test.

Note that the SDRAM controller of the Excalibur device can be used in 16-bit or 32-bit mode. The 16-bit mode uses the lower 16 lines of the data bus and the 32-bit mode uses all lines of the data bus. Altering the settings of the stripe and modifying the width of the `sdramdq`, `sdramdqs` and `sdramdqrn` signals in the schematic of the testbench, generates a design which uses the 16-bit or 32-bit mode. The SDRAM testbench was used in 16-bit and 32-bit mode.

The results of the followed testing methodology were as follows: the testbench functioned correctly in 16-bit mode. However, the testbench did not function in 32-bit mode. Even though all connections were correct and different timing settings were used, the 32-bit mode was not functioning. Future work should incorporate a more thorough test of the 32-bit SDRAM mode.

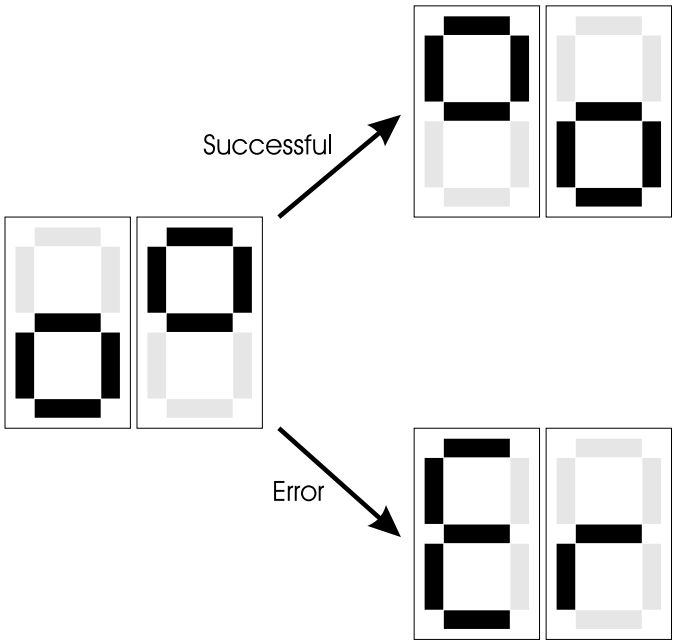

Figure 3.13: Behavior of the 7-segment displays during the memory test.

3.10 UART interface

The UART interface consists of a MAX3241 voltage level convertor, a 9-pins female SUB-D connector and a few capacitors. Figure 3.15 presents the location of the UART interface components.

In order to test the UART interface, a testbench was designed. Since almost every book concerning programming starts with the classic `Hello World!` program, this was also be the base of the UART testbench. The schematic of the testbench, which was created in the Quartus II development tool, is presented in Figure 3.16. The program generates the following output on the UART:

```
**************************************************
**                                              **
** Hello DAMP Team!                             **
**                                              **
** DAMP is alive and kicking....                **
**                                              **
**************************************************
```

Appendix B presents the listings of the C and assembly files. The file `armc_startup.s` starts up the ARM C environment. This is required in order to run programs written in C. The `uartcomm.c` file contains a routine that initializes the UART, such that it uses a baudrate of 38400, 8 data bits, 1 stop bit and no parity. The other files are required to provide communication via the UART and to handle interrupts and exceptions. The

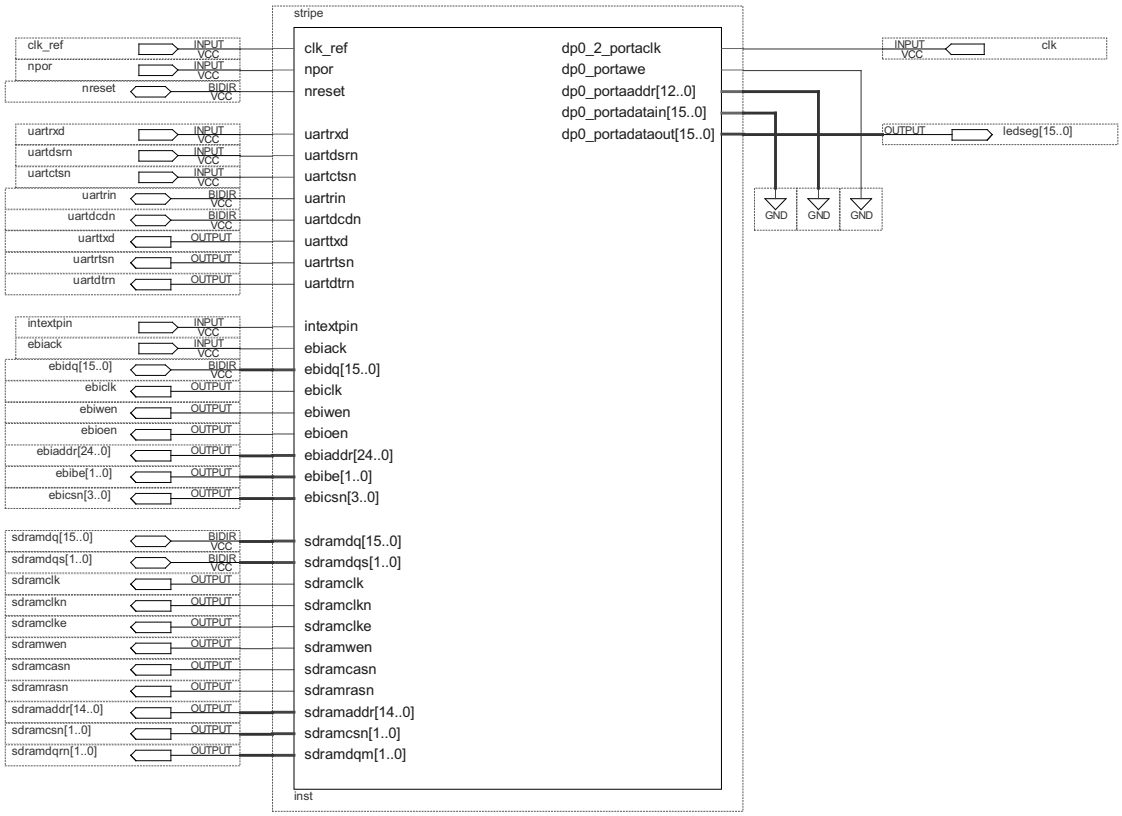

Figure 3.14: Schematic of the 16-bit SDRAM testbench.

Quartus II development software was used to compile the design and the software. The resulting bootloader was be programmed into Flash memory, so the Excalibur device could boot the program.

During the construction of the UART interface circuit, it turned out that the footprint of the MAX3241 was too small for the device. This was solved by bending the pins of the device. Furthermore the female 9-pins SUB-D connector had to be mounted on the bottom of the PCB in order to fit.

The testbench was successfully programmed into Flash memory. However, incorrect characters were displayed by the terminal program. Investigating this issue made clear that a null-modem cable should be used, which has a female connector on each end. Because a female 9-pins SUB-D connector was mounted on the DAMP PCB, a standard null-modem cable cannot be used. Using a male 9-pins SUB-D connector on the DAMP PCB would have solved the null-modem cable problem and this connector could have been mounted on the top of the DAMP PCB.

To solve this issue, a custom cable was made in order to connect the output of the UART interface to the serial port of a computer. After applying this modification, the text was displayed correctly by the terminal program. Figure 3.17 presents a screenshot

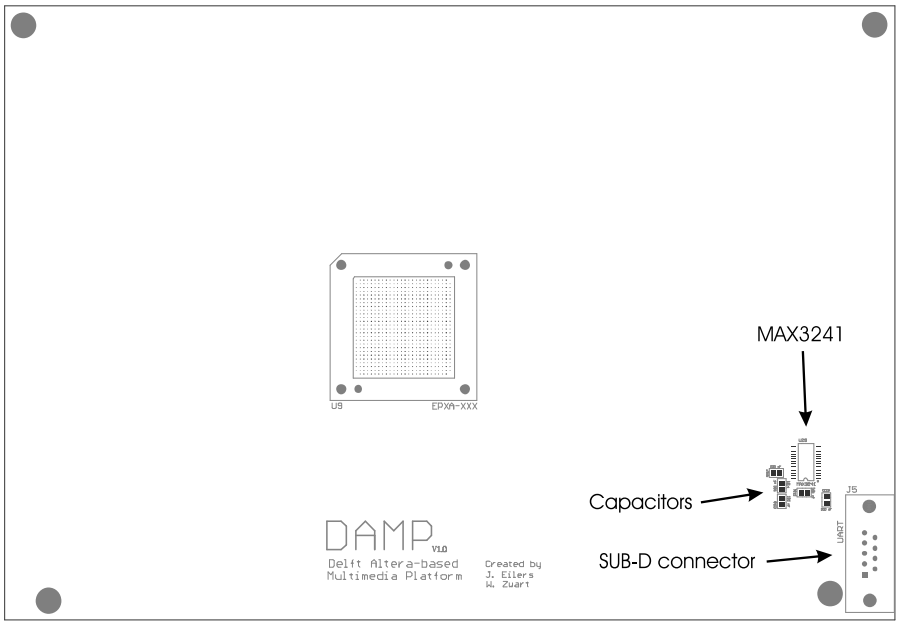

Figure 3.15: Location of the UART interface components.

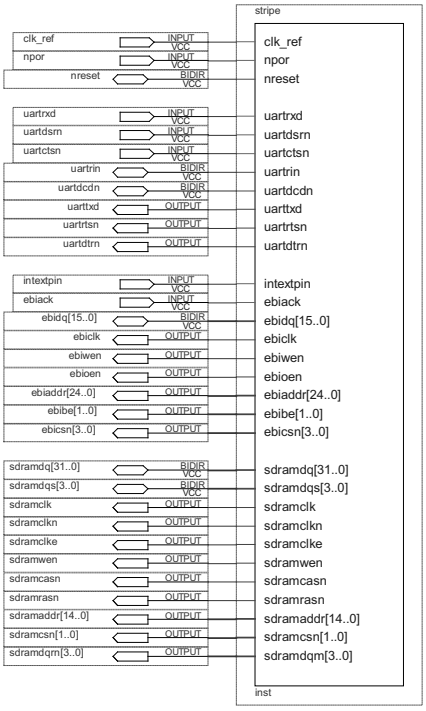

Figure 3.16: Schematic of the UART testbench.

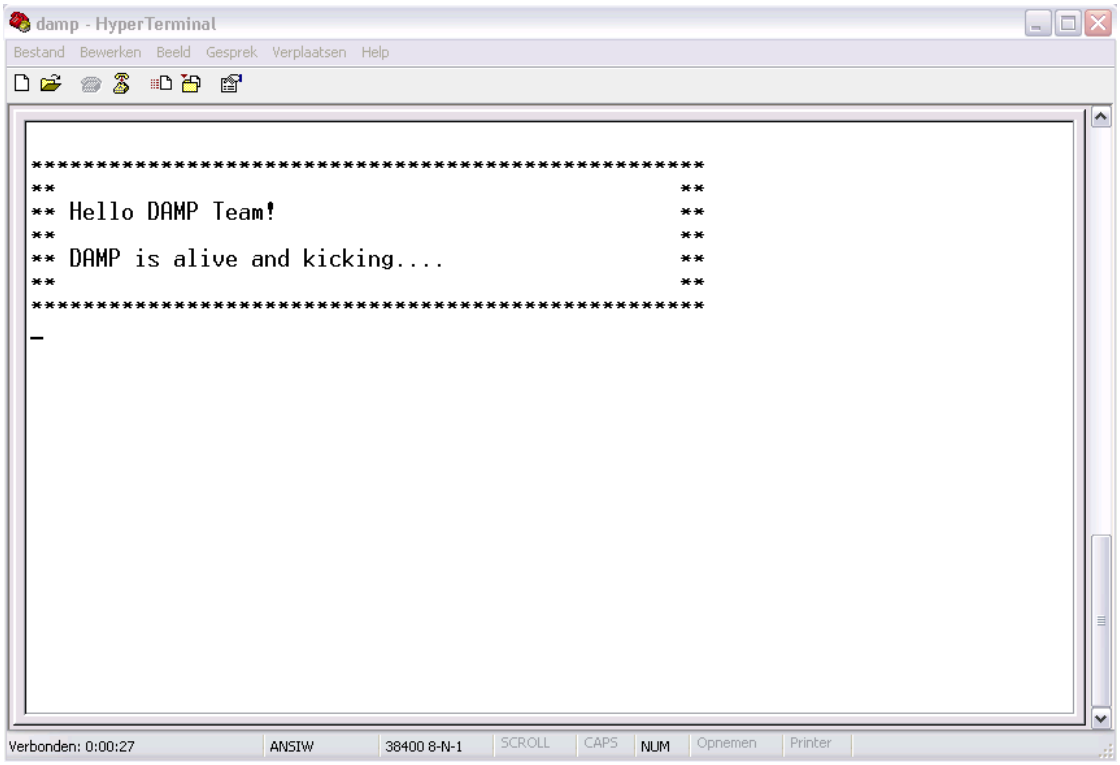

Figure 3.17: Screenshot of the UART testbench output.

of the output of the UART testbench, displayed by a terminal program.

3.11 VGA

The VGA circuit basically consists of three simple DACs, which are made out of resistors. The red and green color signals are created by a 3-bit DAC and the blue color signal by a 2-bit DAC. The output voltage level of the outputs of the Excalibur device can be 3.3V, 2.5V or 1.8V. The level of these outputs, in DAMP's case, is equal to the voltage level of the Excalibur I/O power supply, which is 3.3V. The resistor network was designed for a 5V input, such that a voltage level convertor is needed. The VGA circuit contains two 7407 buffers, which function as a voltage level convertor. Figure 3.18 presents the location of the VGA circuit components.

In order to test the VGA circuit, a testbench was designed that generates a color pattern. The output can be displayed on a VGA monitor. In order to do that, the testbench generates the VGA synchronization signals HSYNC and VSYNC, and the color signals red, green and blue.

Negative pulses on the HSYNC signal indicate the end of a line and the start of a new line. Negative pulses on the VSYNC signal indicate the end of a frame and the start of a new frame. Figure 3.19 presents the relation between the HSYNC and the VSYNC

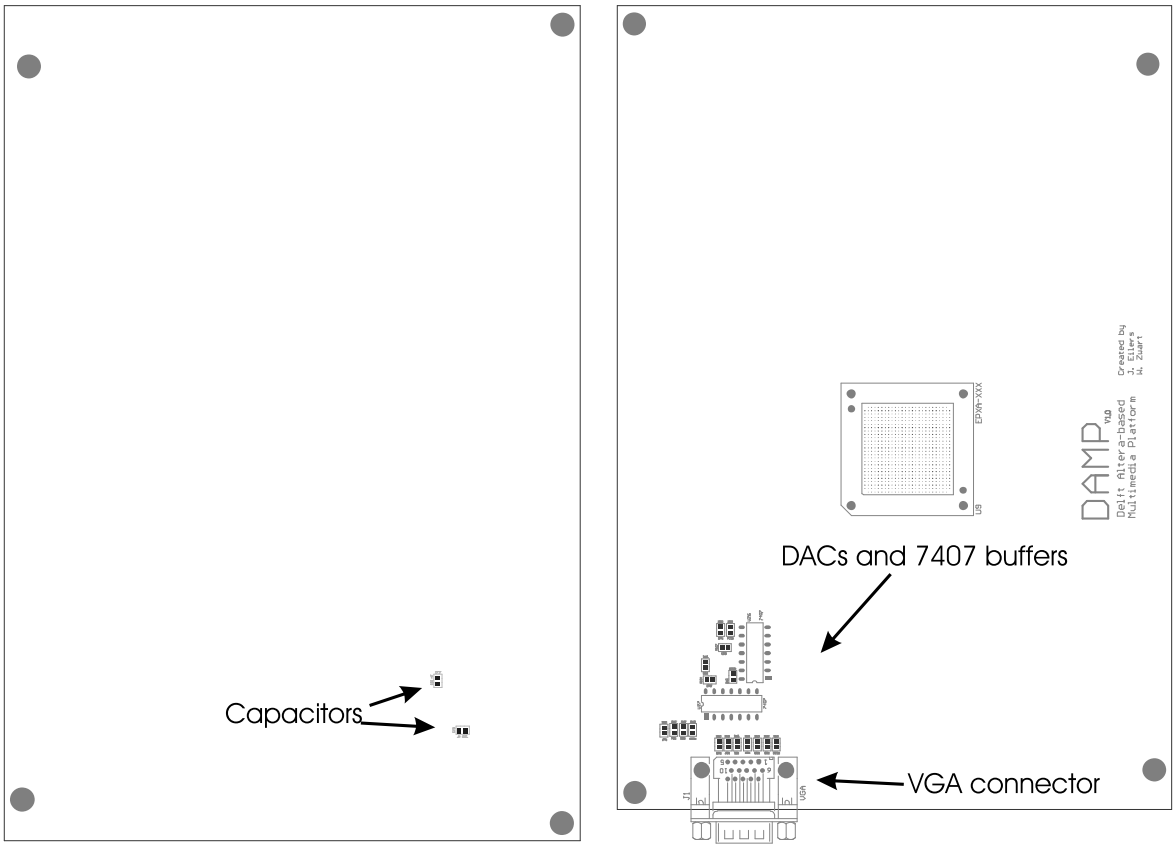

Figure 3.18: Location of the VGA circuit components.

signals. The variables within his figure have the following meaning:

Thsync The duration of the horizontal synchronization pulse, measured in pixels. A pixel is generated on each pulse of the clock oscillator that drives the VGA signal generator.

Thgdel The duration of the time between the end of the horizontal synchronization pulse and the start of the horizontal gate, measured in pixels. This is usually called the back porch.

Thgate The duration of the visible area of a video line, which is typically 640 pixels. This is usually called the active time.

Thsdel The duration of the time between the end of the visible area of a video line, and the start of the horizontal synchronization pulse, measured in pixels. This is usually called the front porch.

Thlen The duration of a complete video line, measured in pixels.

Tvsync The duration of the vertical synchronization pulse, measured in lines.

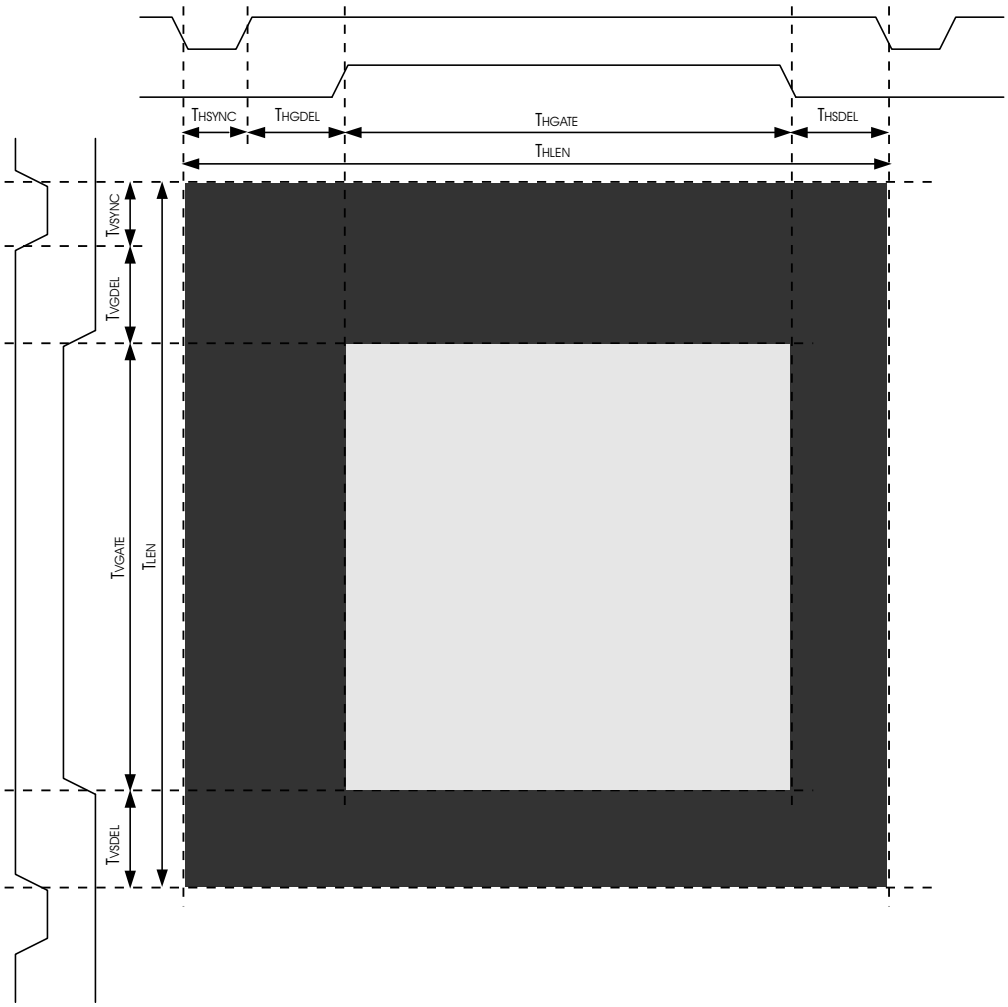

Figure 3.19: Video frame timing overview.

Tvgdel The duration of the time between the end of the vertical synchronization pulse and the start of the vertical gate, measured in lines. This is usually called the back porch.

Tvsdel The duration of the time between the end of the visible area of a video frame, and the start of the vertical synchronization pulse, measured in lines. This is usually called the front porch.

Tvgate The duration of the visible area of a video frame, which is typically 480 lines. This is usually called the active time.

Tvlen The duration of a complete video frame, measured in lines.

The value of these variables depends on the VGA mode that is used. Table 3.1 presents the horizontal timing information of two VGA modes that generate 640x480 video frames.

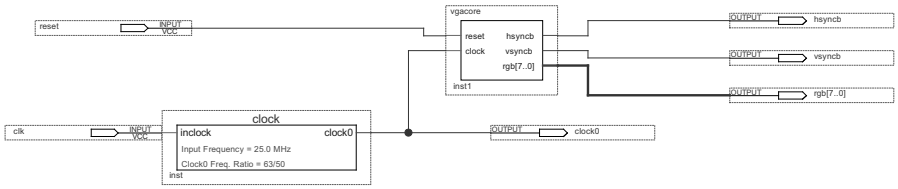

Figure 3.20: Schematic of the VGA testbench.

Table 3.2 presents the vertical timing information.

Refresh rate	Pixel clock	Sync pulse		Back porch	Active time	Front porch	Line total
	MHz	usec	pix	pix	pix	pix	pix
60 Hz	25.175	3.81	96	45	646	13	800
72 Hz	31.5	1.27	40	125	646	21	832

Table 3.1: Horizontal timing information.

Refresh rate	Line width	Sync pulse		Back porch		Active time		Front porch		Frame total	
	usec	usec	lin	usec	lin	usec	lin	usec	lin	usec	lin
60 Hz	31.78	63	2	953	30	15382	482	285	9	16683	525
72 Hz	26.41	79	3	686	26	12782	484	184	7	13735	520

Table 3.2: Vertical timing information.

The timing information of the 72 Hz mode was used, because the required pixel clock frequency of 31.5 MHz can be easily created by using a PLL in the PLD of the Excalibur device. This can be accomplish this by setting the multiplier of the PLL to 63, which multiplies the input frequency of 25 MHz by 63, and setting the divider of the PLL to 50, which divides the result by 50. Figure 3.20 presents the schematic of the VGA testbench. In Appendix C one can find the VHDL description of `vga_core`. Note that the output of the PLL is connected to a dedicated clock pin of the Excalibur device to provide an additional test for the PLL in the PLD of the Excalibur device.

During the VGA tests it became clear that the color signal `RGB[3]` could not be connected to the physical pin `J2`, which is connected to a 7407 buffer. This is because the pin `J2` is a not-connected pin on the EPXA1 Excalibur device. Using the EPXA4 would have solved this problem, but DAMP should support VGA with both the EPXA1 and EPXA4.

The design was modified by soldering a wire on two pins of the breakout header, which is connected to the Excalibur pins, such that `J2` is connected to `J3`.

After the PLD of the Excalibur device was programmed with the VGA testbench, nothing appeared on the VGA monitor. Measuring the outputs of the I/O pins of the Excalibur device, at the inputs of the 7407 buffers, made clear that the testbench was not generating signals. However, the simulation results of the VGA testbench indicated that the design should function correctly.

Since the VGA testbench requires a clock in order to function, measuring the output of the PLL in the PLD of the Excalibur device is a logical step. As described earlier, the output of the PLL is connected to a dedicated clock output pin. Probing the dedicated clock output pin made clear that the PLL was not functioning.

The PLLs are internal circuits that use dedicated clock input lines. The number of PLLs present inside the Excalibur device depends on the device type: the EPXA1 has two PLLs and the EPXA4 has four. In the EPXA1 one of the PLLs can be connected to a dedicated clock output pin. The EPXA4 has two dedicated clock output pins. The PLL circuits also have their own dedicated power supply pins. These should be connected to the same voltage level as the stripe of the Excalibur device, which is 1.8V. The dedicated clock output pins also have their own dedicated power supply pins. These should be connected to 3.3V.

Examining the DAMP schematic, made clear that two of these power supply pins (`VCC_CKLK2` and `VCC_CKLK4`), that were supposed to be connected to 1.8V, were actually connected to a voltage level of 3.3V. It also became clear that the power supply pins (`VCC_CKOUT1` and `VCC_CKOUT2`) of the two dedicated clock output pins were connected to 1.8V instead of the required 3.3V.

The design was modified by desoldering the chip inductors `L8` and `L10` at the side of the 3.3V voltage level connection, and soldering two wires from the chip inductors to a connection on the PCB with a 1.8V voltage level. The design flaw at the power supply pins of the dedicated clock output pins was corrected by desoldering the chip inductors `L11` and `L12` at the side of the 1.8V voltage level connections, and soldering two wires from the chip inductors to a connection on the PCB with the 3.3V voltage level.

After applying these modifications, the PLD of the Excalibur device was programmed again with the VGA testbench. Probing the dedicated clock output pin made clear that the PLL was functioning. However, there was still no result visible on the VGA monitor. Measuring the outputs of the I/O pins of the Excalibur device at the inputs of the 7407 buffers, indicated that the testbench was generating signals. But there was no activity on the outputs of the buffers.

After re-examining the schematics, it became clear that a design flaw exists at the open-collector outputs of the 7407 buffers [18]. The open-collector outputs require a pull-up resistor in order to pull up the output voltage level. However, attaching a pull-up resistor on the output affects the behavior of the resistor ladder network, which is attached to the outputs of the buffers and is used to generate the three color signals red, green and blue.

This problem was solved by using a 7414 hex-invertor, which is pin-compatible with the 7407 and has no open-collector outputs. Using the 7414 hex-invertor, automatically implied the use of invertors in the VGA testbench. The modified testbench is presented in Figure 3.21.

These modifications seemed successful, because a color pattern was visible on the VGA monitor. However, the displayed color pattern was not correct. Investigating the DAMP schematic turned out that the labels, which denote the color signal and are used in the schematic, are incorrect. The results were as follows:

- The `VGA_R3` PCB signal is connected to pin `J5`, but should be connected to `L2`.

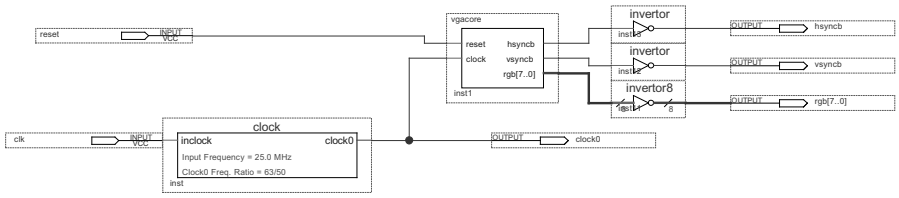

Figure 3.21: Schematic of the modified VGA testbench.

- The `VGA_R2` PCB signal is connected to pin `H4`, but should be connected to `J3`.
- The `VGA_R1` PCB signal is connected to pin `L2`, but should be connected to `K2`.
- The `VGA_G3` PCB signal is connected to pin `H5`, but should be connected to `H4`.
- The `VGA_G2` PCB signal is connected to pin `J3`, but should be connected to `J5`.
- The `VGA_G1` PCB signal is connected to pin `K2`, but should be connected to `H5`.
- The `VGA_B2` PCB signal is connected to pin `H3`, which is correct.
- The `VGA_B1` PCB signal is connected to pin `E3`, which is correct.

Modifying these design flaws was done by modifying the pin assignments of the VGA testbench. After applying this modification, the PLD of the Excalibur device was programmed again, which resulted in a correct color pattern on the VGA monitor.

3.12 Audio

The Audio circuit contains a Texas Instruments TLC320AD77C audio CODEC, three Texas Instruments TLV2362 OPAMPs, a CINCH multi-connector and a few passive components. Figure 3.22 presents the location of the Audio circuit components.

In order to test the Audio circuit a testbench was designed. The CODEC is continuously sampling data, which is transmitted by the `SDOUT` pin of the CODEC. This data can be used as an output, by utilizing a loopback device. The testbench therefore contains a loopback device, which uses the input data from the `SDOUT` pin of the CODEC as an output to drive the `SDIN` pin of the CODEC. A simple schematic of the loopback device is presented in Figure 3.23. A side effect of the testbench is that the left and right input channels are reversed at the output.

As stated before, the TLC320AD77C audio CODEC supports various communication standards. The 16-bit I^2S standard was used, because of the wide spread use of the standard. The testbench generates three clock signals: `MCLK`, `SCLK` and `LRCLK`. These clock signals have to meet the following requirements, which can be found in the data sheet [13] of the TLC320AD77C:

- The frequency of `LRCLK` must be equal to the sample frequency.
- The frequency of `SCLK` must be 64 times the sample frequency.

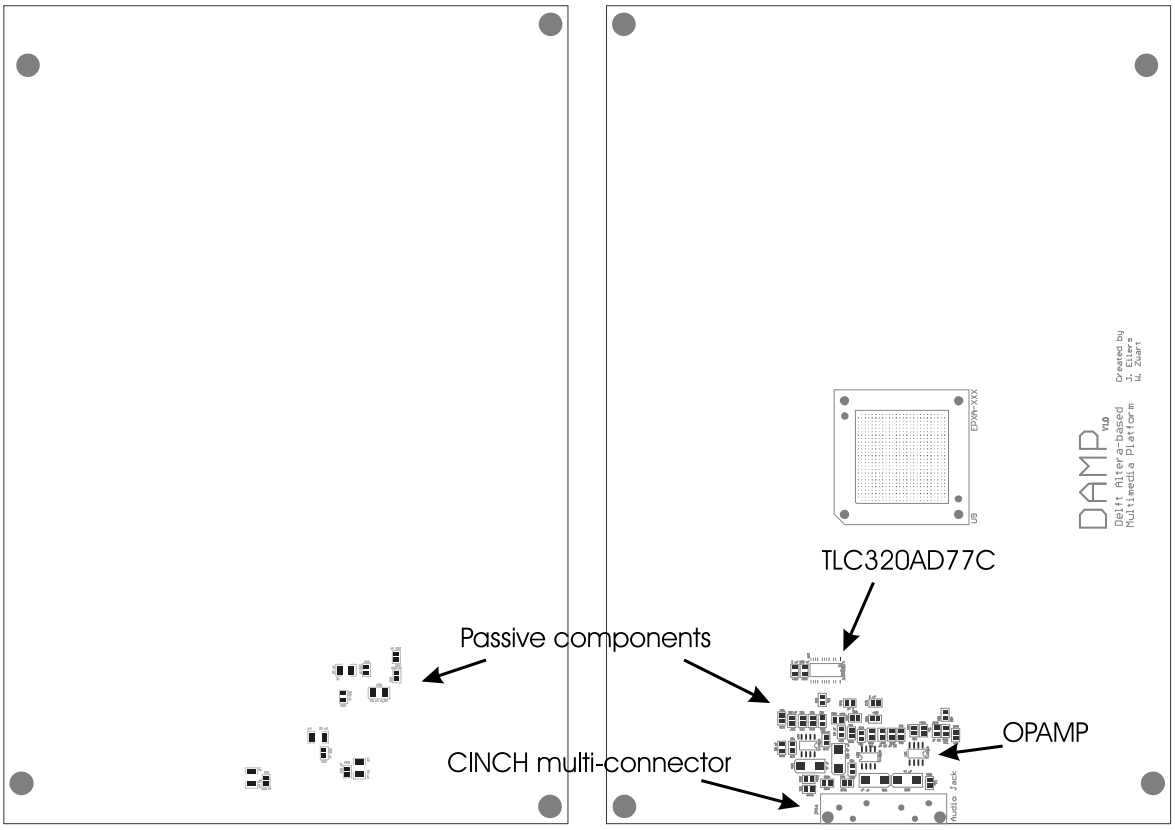

Figure 3.22: Location of the Audio circuit components.

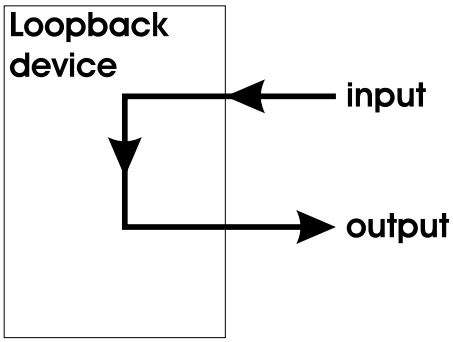

Figure 3.23: Simple schematic of a loopback device.

- The frequency of the `MCLK` is four times the `SCLK` frequency, thus 256 times the sample frequency.
- The edges of the `SCLK` and the `MCLK` signals must be at least 3ns apart.

The testbench uses a sample frequency of approximately 32 kHz. The preferred

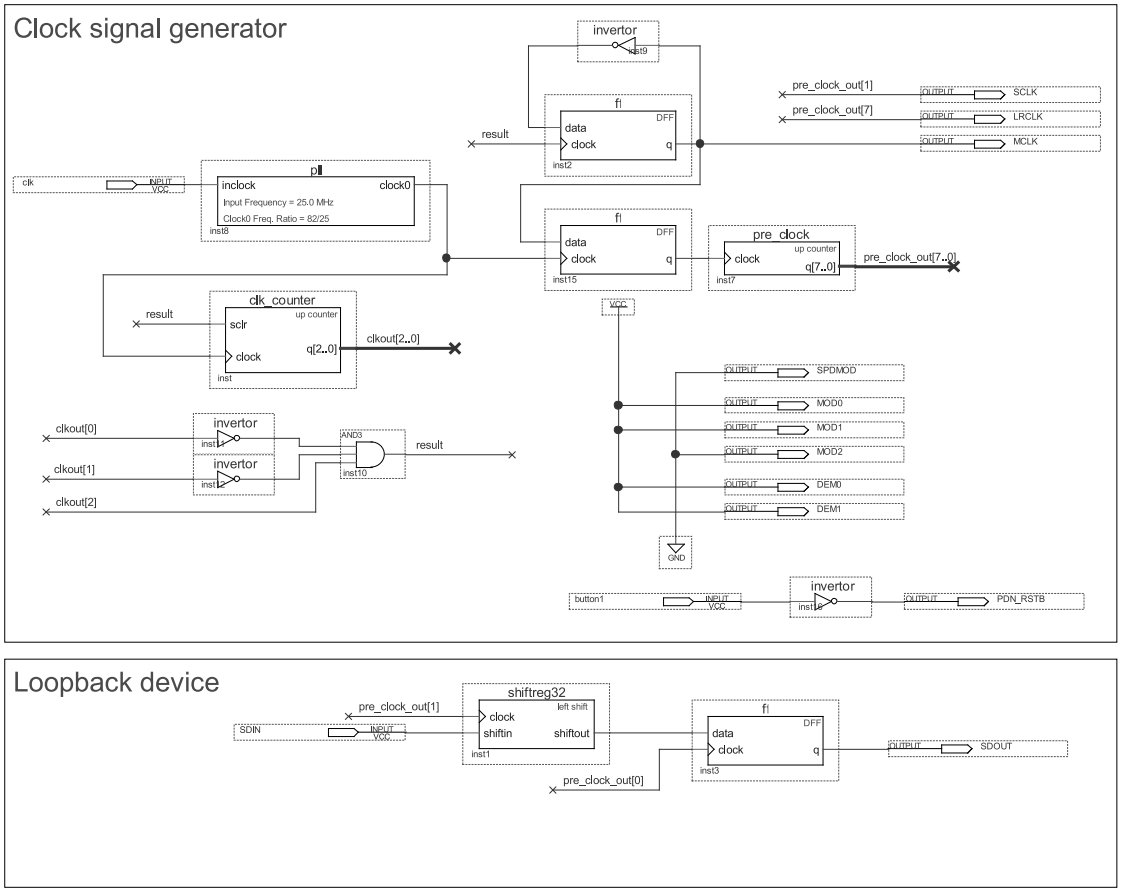

Figure 3.24: Schematic of the Audio testbench.

frequency of `MCLK` should then be 8.192 MHz, but since a 25MHz clock oscillator is used, this frequency cannot be generated by a PLL because of the limited divider and multiplier settings of the PLL. However, the PLL can generate a frequency of 82MHz, which is divided by 10 in order to obtain a frequency of 8.2 MHz. This results in a sample frequency of 32.031kHz. Because the edges of `SCLK` and `MCLK` should be at least 3ns apart, a flipflop was inserted. This flipflop delays the `SCLK` and `LRCLK` with approximately 12ns with respect to the `MCLK` signal. Figure 3.24 presents the schematic of the Audio testbench.

During the construction of the Audio circuit, it became clear that the footprint of the TLC320AD77C Audio CODEC was to small. This was corrected by bending the pins of the CODEC. Furthermore, the footprint of the CINCH multi-connector was mirrored. The CINCH multi-connector was therefore mounted on the bottom of the PCB.

After programming the PLD of the Excalibur device with the Audio testbench, the Audio circuits input was connected with the audio output of a laptop. The Audio circuits output was connected to a speaker set with build in amplifier. However, the Audio circuit

did not function.

Measuring at the input of the Audio circuit, indicated that there was a signal present on the input. However, measuring the analog input pins of the audio CODEC indicated that the input circuit was incorrect. Re-examining the data sheet of the TLC320AD77C audio CODEC [13] and the schematic, made clear that the input circuit was incorrect. The following design flaws were found:

- The 2.5V reference voltage at pin 5 of Opamp U21, should be 1.65V.
- A connection was missing between pin 7 of Opamp U21 and resistor R81.
- A connection was missing between pin 7 of Opamp U23 and resistor R83.
- Opamp U23 was not connected to power, because the connection between the inductor L4 and the capacitor C92 was not connected to the 5V voltage level.
- Capacitors C75 and C76 had a value of 1μF, but this value should be 1nF.

After modifying the prototype, the analog input pins of the audio CODEC were probed. This indicated that the input circuit was functioning correctly. However, there was no activity on the analog output pins of the CODEC. Measuring the `LRCLK`, `SDCLK` and `MCLK` clock signals, made clear that the timing requirements of the clock signals were met.

The Audio circuit has to be tested more thoroughly, but this is left as future work.

3.13 PS/2

The PS/2 circuit only consists of a PS/2 compatible mini DIN connector, which is connected to the ground plane and to the 5V voltage level of the power plane. The location of the PS/2 connector is presented in Figure 3.25.

In order to test the PS/2 connector, a testbench was implemented that displays the hexadecimal code send by a keyboard when pressing a key. The hexadecimal code appears on the 7-segment displays. Figure 3.26 presents the schematic of the testbench. The listings of the VHDL files of the testbench can be found in Appendix D.

The results of the test was as follows. The PS/2 testbench functioned correctly, which indicated that the PCB traces are correct. However, the PS/2 standard is based on a 5V voltage level and the I/O of the Excalibur device is based on a 3.3V voltage level. This difference in voltage levels can damage the Excalibur device, because the I/O pins of the Excalibur device are not 5V tolerant. Measuring the clock and data pins of the PS/2 connector made clear that a voltage level of 5V was present.

There is a solution to this problem. According to a publication [19] from Altera, the device can be made 5V tolerant by using a special I/O mode: 3.3V PCI. Doing this enables the PCI clamping diode present on each I/O pin of the Excalibur device. This results in a voltage drop on the input, such that the voltage is 4.1V or less. To limit a large current draw from the 5V device, a resistor should be inserted between the device and the I/O pin of the Excalibur device. This resistor should be small enough for a fast signal rise time and large enough so it does not damage any device. In an example,

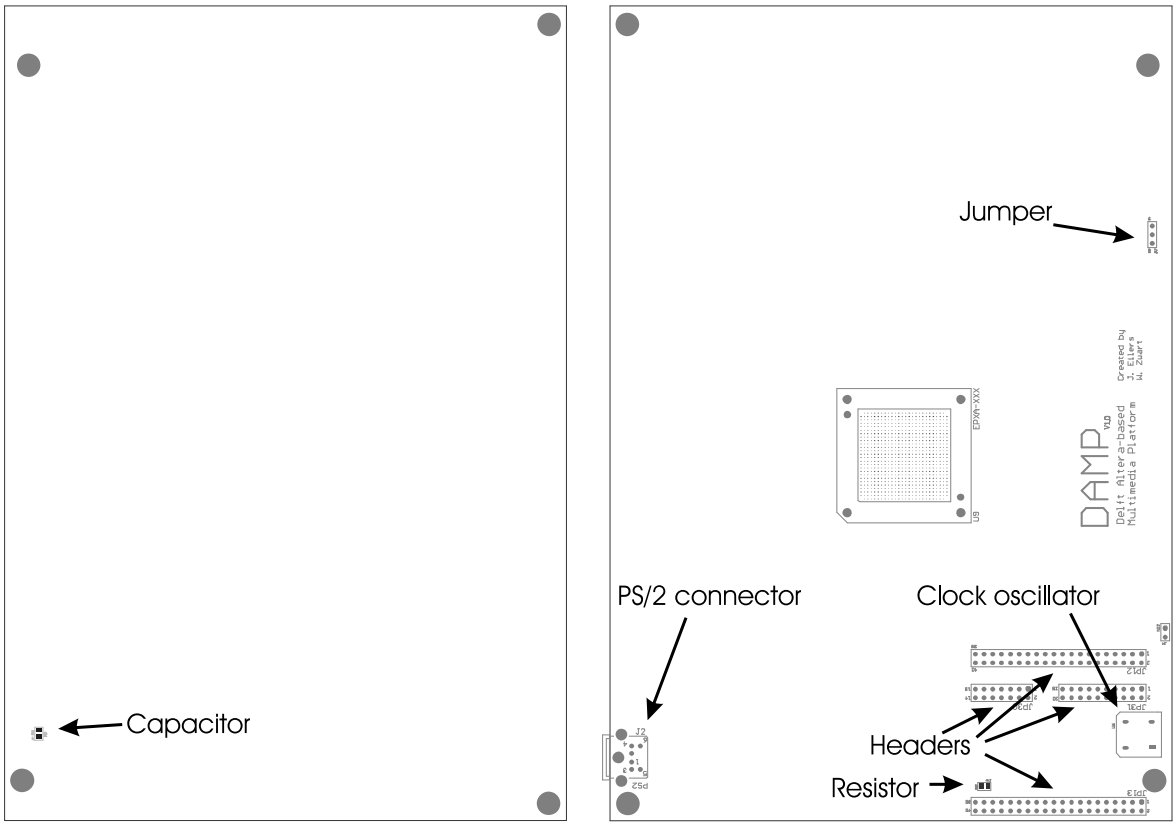

Figure 3.25: Location of the PS/2 connector and the Daughter card interface components.

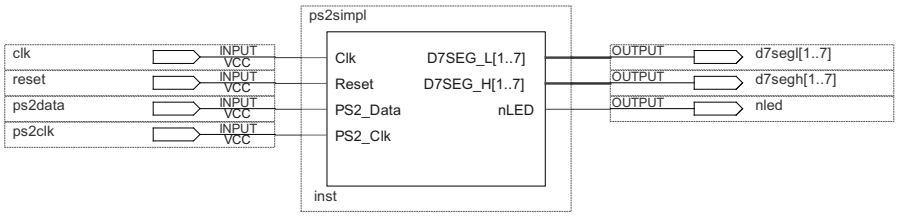

Figure 3.26: Schematic of the PS/2 testbench.

Altera indicates that a 164Ω resistor can be used and that the example employs worst-case conditions. To be certain of not damaging any device, a resistor of 300Ω was soldered inside a keyboard. Because the PS/2 clock frequency has a maximum of 16kHz, there should be no problem with slow signal rise times.

After applying this modification, a voltage of 3.79V was measured on the clock and data pins of the PS/2 connector. The PS/2 testbench still functioned correctly.

3.14 Daughter card interface

The Daughter card interface consists of four breakout headers, a jumper, a clock oscillator and two passive components. The location of these components is presented in Figure 3.25.

The Daughter card interface contains three headers which are compatible with the Nios daughter card interface. Furthermore, the Daughter card interface contains an additional header which is the DAMP extension header. As stated before, the DAMP extension header is not connected to the Excalibur device used on the current prototype. This is because the EPXA1 device is used, which has fewer I/O pins available than the EPXA4. Thus, only the Nios daughter card interface compatible part of the DAMP Daughter card interface could be tested.

In order to test the remaining three headers of Daughter card interface, a testbench was developed. The Daughter card interface contains three dedicated clock pins: a clock oscillator output, a pin which is connected to a PLL output pin of the Excalibur device, and a clock input directed to the Excalibur device. The clock oscillator, which drives the clock oscillator output pin, can be enabled and disabled by setting the `J17` jumper in the `On` or `Off` position. This behavior was verified by probing the clock oscillator pin. The PLL output pin of the Excalibur device was attached to a PLL in the testbench in order to test the PLL output pin. The clock input pin was treated as an I/O pin.

The Daughter card interface also contains a reset pin, which can be used to reset the Excalibur device externally. This behavior was analyzed by connecting the reset pin `MR` to the ground plane and a 3.3V voltage level.

All other pins are I/O pins or not connected pins. Testing the I/O pins can be done in numerous ways. A simple 2-bit counter was used in the testbench, which feeds the odd and even rows of the Daughter card interface. The odd rows are connected to output bit 0 of the counter and are thus driven by a signal with a frequency of 25MHz. The even rows are connected to output bit 1 of the counter and are thus driven by a signal with a frequency of 12.5MHz. Figure 3.27 presents the testbench that tests both the PLL output of the Excalibur device and the I/O pins.

One of the last design steps, during the design process of the Daughter card interface testbench, was assigning the pins of the testbench design to the physical I/O pins of the Excalibur device. It became clear that the signals `nios40_27` and `nios40_32` could not be connected to physical pins `Y7` and `Y9` respectively, which are the pins that are connected to pins 27 and 32 of breakout header `JP13`. This is because the pins `Y7` and `Y9` are not connected on the EPXA1 Excalibur device.

This problem was solved by soldering wires between pins of the connector to the Excalibur pins. The wires were soldered between pin `Y7` and `AE16` and between `Y9` and `AC21`.

After applying this modification, the PLD of the Excalibur device was programmed with the Daughter card interface testbench. The results of the followed testing methodology are as follows:

- Connecting the pin `MR` to ground did result in a reset of the Excalibur device.

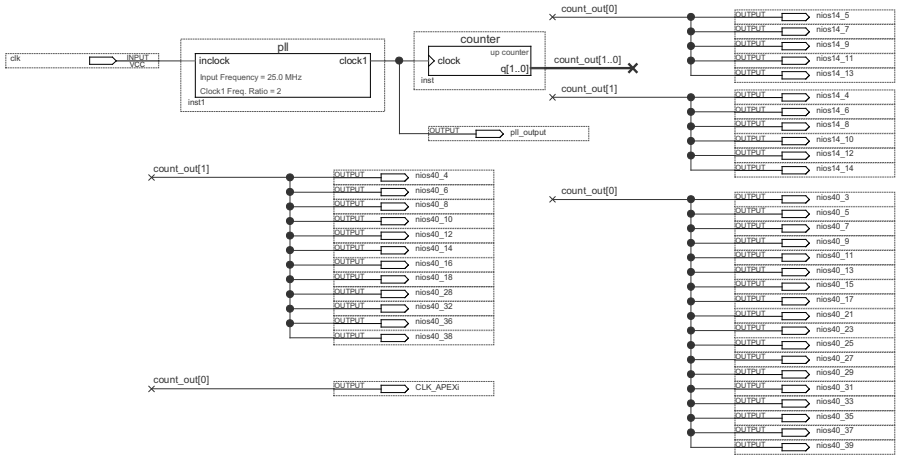

Figure 3.27: Schematic of the Daughter card interface testbench.

- The clock oscillator functioned correctly in its enabled state and was inactive in its disabled state.
- The silkscreen that indicates the enabled (`On`) position of the clock jumper, actually indicated the disabled position.
- The dedicated clock output pin of the Daughter card interface was functioning correctly.
- Probing the I/O pins of the Daughter card interface gave the expected result.

3.15 Summary

The design flaws which have been found during the testing process can be summarized as follows:

1. **Power circuit:**
   - `VCC_CKLK2` should be connected to a voltage level of 1.8V.
   - `VCC_CKLK4` should be connected to a voltage level of 1.8V.
   - `VCC_CKOUT1` should be connected to a voltage level of 3.3V.
   - `VCC_CKOUT1` should be connected to a voltage level of 3.3V.
   - All vias, which are located inside the footprint of the WH25 high power resistor, should be relocated.
   - All traces, which are routed through the footprint of the WH25 high power resistor, should be rerouted.
   - The silkscreen that indicates the `Off` and `On` position of the power button, should be corrected.

2. **Clock circuit:**
   - The silkscreen that indicates the `Off` and `On` position of the enable/disable jumpers, should be corrected.
3. **Reset circuit:**
   - The value of capacitor C98, which is located in the debounce circuit, should be 100nF.
4. **User I/O:**
   - A buffer capacitor should be added to the power connection of the 74LV14 hex invertor.
   - The connections of the dipswitch bank should be altered, such that the `On` indication is correct.
   - The silkscreen of the push-buttons should be altered, such that the push-button numbering is correct.
5. **Flash memory:**
   - Flash addresses 0 to 23 should be connected to EBI addresses 1 to 24.
6. **UART interface:**
   - The footprint of the MAX3241 should be resized.
7. **VGA:**
   - The 7407 open-collector hex buffer should be replaced by a 7414 hex invertor.
   - The `VGA_R3` PCB signal should be connected to `L2`.
   - The `VGA_R2` PCB signal should be connected to `J3`.
   - The `VGA_R1` PCB signal should be connected to `K2`.
   - The `VGA_G3` PCB signal should be connected to `H4`.
   - The `VGA_G2` PCB signal should be connected to `J5`.
   - The `VGA_G1` PCB signal should be connected to `H5`.
8. **PS/2:**
   - Resistors should be added to the data line and clock line of the PS/2 interface.
9. **Daughter card interface:**
   - Pin `nios40_27` should be connected to `AE16`.
   - Pin `nios40_32` should be connected to `AC21`.

Since the Audio feature was not tested more thorough, it does not appear in the summary. This implies that the Audio circuit was not redesigned.

4 Schematic and PCB redesign

When designing a complex multimedia platform such as DAMP, it is unavoidable to have design mistakes, despite the effort to achieve a first-time-right design. In order to correct the design flaws found in DAMP, some of the parts had to be redesigned. This chapter presents the redesign process of these parts.

4.1 Introduction

The DAMP Design trajectory (Chapter 1, Figure 1.2) basically consists of an iterative loop. Chapter 3 describes the last step of this loop. The design flaws, summarized in Section 3.15, are used as a guideline in order to redesign the features that contain the design flaws. The redesign process starts with the following steps:

- A redesign of the DAMP schematic and DAMP PCB, in order to correct functional design flaws.
- A redesign of the DAMP PCB, in order to correct PCB design flaws. Note that PCB design flaws have no relation with the functional design flaws.

The last step of the iterative loop of the DAMP Design Trajectory consists of testing another prototype. Future work should incorporate this.

4.2 Redesign

In order to correct the design flaws, the related parts need redesign. The following sections describe all redesign issues in more detail.

Power circuit

The following design corrections were performed, in order to correct the Power circuit:

- **Functional redesign:** The Excalibur PLL power pins `VCC_CKLK2` and `VCC_CKLK4` were connected to net-label 3.3V, and are now connected to 1.8V (Figure 4.1). Pin `VCC_CKOUT1` and pin `VCC_CKOUT2` were connected to net-label 1.8V, and are now connected to 3.3V. The Power circuit now provides the correct voltage levels to the Excalibur power supply pins, capacitors and inductors.
- **PCB redesign:** The Excalibur PLLs power supply capacitors (`C100`, `C102`, `C105`, `C106`) and inductors (`L6`, `L8`, `L11`, `L12`) were relocated on the PCB, because they should be placed as close to the appropriate power supply as possible. The power

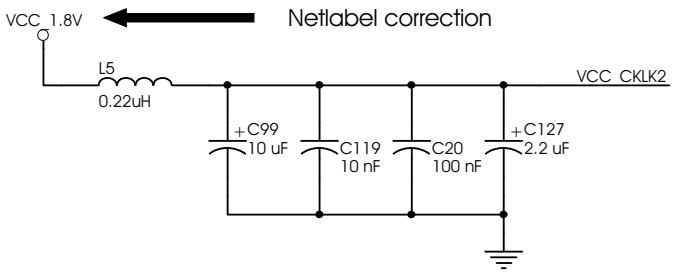

Figure 4.1: Example of a net-label correction of a PLL power supply pin.

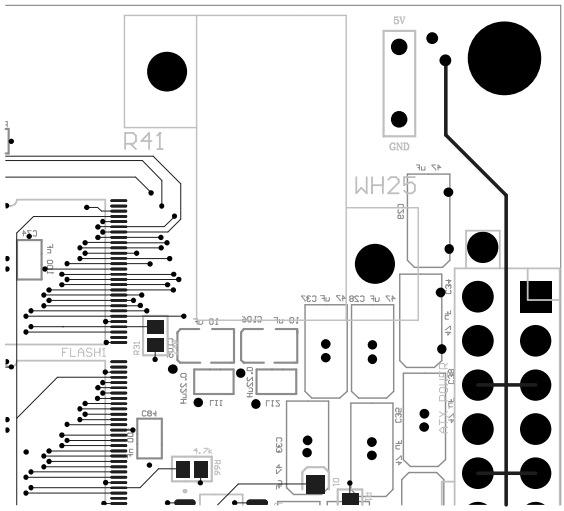

Figure 4.2: Footprint of the WH25 high power resistor without traces or vias.

supply in this case is either the ATX power supply and ATX connector (3.3V), or the PT6520 voltage regulator (1.8V). The filtering capacitors (100nF, 10nF, and 2.2μF) do not require relocation, since they are intended for filtering the high-frequency signals originating from the PLLs of the Excalibur device. All via's inside the footprint of the WH25 high power resistor were relocated, in order to avoid electrical connections to the WH25 high power resistor. Furthermore, all traces on the top layer, which are crossing the footprint of the WH25 high power resistor (Figure 4.2), were rerouted on other layers. The `On` and `Off` silkscreen marks of the power switch were corrected.

Clock circuit

The silkscreen text indicating the `Off` and `On` positions of the enable/disable jumpers of the clock oscillators was corrected, in order to provide a valid indication of the clock oscillator setting.

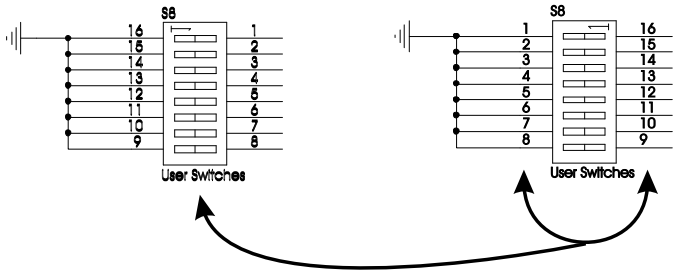

Figure 4.3: Redesign of the dipswitch bank connections.

Reset circuit

The following design corrections were performed, in order to correct the Reset circuit:

- **Functional redesign:** As explained in Chapter 3, the value of capacitor `C98` in the debounce circuit of the reset push-button should be 100nF. The capacitor value was changed in the schematic design.
- **PCB redesign:** The footprint of capacitor `C98` was resized to a standard 0805 footprint, in order to be able support the mounting of a 100nF capacitor.

User I/O

The following design corrections were performed for the User I/O circuit:

- **Functional redesign:** A 100nF capacitor was added between the power supply of the 74LV14 hex invertor and ground. The connections of the dipswitch bank were altered, in order to provide a valid display of the `On` indication. This is presented in Figure 4.3.
- **PCB redesign:** The 100nF capacitor of the 74LV14 hex invertor was added on the PCB (capacitor `C135`) close to the 74LV14 hex invertor. The changes in the dipswitch bank connections were also adjusted in the PCB design.

Flash memory

The following design corrections were performed for Flash memory:

- **Functional redesign:** In order to support 16-bit Flash devices, capable of being programmed by the Altera flash programmer, the Excalibur EBI address-lines `EBI_A1` to `EBI_A24` were connected at Flash address-lines `A0` to `A23`. This was done by adjusting all net-labels of the first three Flash devices in the schematic design. The highest Excalibur EBI address-line (`EBI_A24`) is used as a chip-select, in order to select either Flash device four or the Ethernet controller. Hence, Flash device four can have a maximum size of 16Mbytes. Other Flash devices still support up to 32Mbytes.

- **PCB redesign:** The address-lines between the footprint of Flash device one and the Excalibur EBI interface were reconnected. The other three Flash devices have their address-lines directly connected to the first Flash device, thus no further rerouting was necessary.

UART interface

The SMD pads of the PCB footprint used for the MAX3241 device were not large enough to mount the component on the PCB. In order to correct this the land pads on the top copper layer of the PCB were enlarged, such that the footprint provides sufficient space for soldering the MAX3241 device.

VGA

The following design corrections were performed for VGA:

- **Functional redesign:** As stated in Section 3.15, the 7407 hex buffers should be replaced by 7414 hex invertors. Hence, the component labels of the hex buffers (7407) were replaced by labels with the correct indication (7414). Furthermore, the labels of the VGA-signals between the 7414 hex-buffers and Excalibur I/O pins were altered to ensure a valid connection between the Excalibur and the DACs. The connection of signal `VGA_R2` to the Excalibur device was altered, such that it connects to Excalibur I/O pin `J3` instead of pin `J2`, because the `J2` pin is a not-connected pin on the EPXA1.
- **PCB redesign:** The silkscreen of the 7407 hex buffer was adjusted, such that it indicates the right component type. In order to match the altered VGA-signals in the schematics, an appropriate rerouting of the VGA traces was performed.

PS/2

In order to support 5V tolerance, two resistors were placed between PS/2 pins (`PS2_data` and `PS2_clk`) and the corresponding Excalibur I/O pins. The PCB was redesigned by placing two additional SMD footprints with size 0805 on the PCB, to accommodate the resistors, and by rerouting the appropriate traces.

Daughter card interface

Two signals of the Daughter card interface (`NIOS40_27` and `NIOS40_32`) were connected to pins which are unused in the EPXA1 (EPXA4 does support these pins as I/O pins). The two Nios signals were reconnected to respectively Excalibur I/O pins `AE16` and `AC21`.

5 DAMP reference designs

In order to support future DAMP development, reference designs are required. Three reference designs were created in order to satisfy this requirement:

1. ***VGA Slideshow:*** *This reference design contains a Direct Memory Access (DMA) controller, which fetches data from SDRAM and outputs this data on a VGA monitor. A software program, which runs on the ARM processor, is used to initialize the DMA controller.*

2. ***DAMP Gamepack:*** *This reference design is a software program, which can reconfigure the PLD of the Excalibur device at run-time. The DAMP Gamepack contains four classic games.*

3. ***DAMP Linux:*** *This Linux version offers a starting point for further Linux development.*

This chapter is organized as follows. Section 5.1 presents the VGA Slideshow reference design. The DAMP Gamepack reference design is discussed in Section 5.2. The third reference design (DAMP Linux) is presented in Section 5.3.

5.1 VGA Slideshow

The VGA Slideshow reference design provides a reference for the development of applications which require both hardware and software. The hardware is written entirely in VHDL. It basically consists of a DMA controller, which reads data from a framebuffer via the AMBA bus of the Excalibur device. Furthermore, it contains additional hardware to process the data and send the appropriate signals to the VGA monitor. The framebuffer is located in the SDRAM. Figure 5.1 presents the global hardware structure of the VGA Slideshow and the hierarchy of the VHDL files.

The top entity `video_system.vhd` contains all components and connections. The `video_dma.vhd` contains the DMA controller (`video_dma_controller.vhd`) and the VGA driver (`vga_driver.vhd`), which generates the appropriate VGA signals. Furthermore, it consists of a line buffer (`line_buffer.vhd`) and a component which contains registers (`slave_interface.vhd`). These registers are required in order to setup, control and monitor the DMA controller and the VGA driver. The `slave_decoder.vhd`, `default_slave.vhd` and `response_and_data_mux.vhd` files function as an interface between the registers and and the AMBA bus.

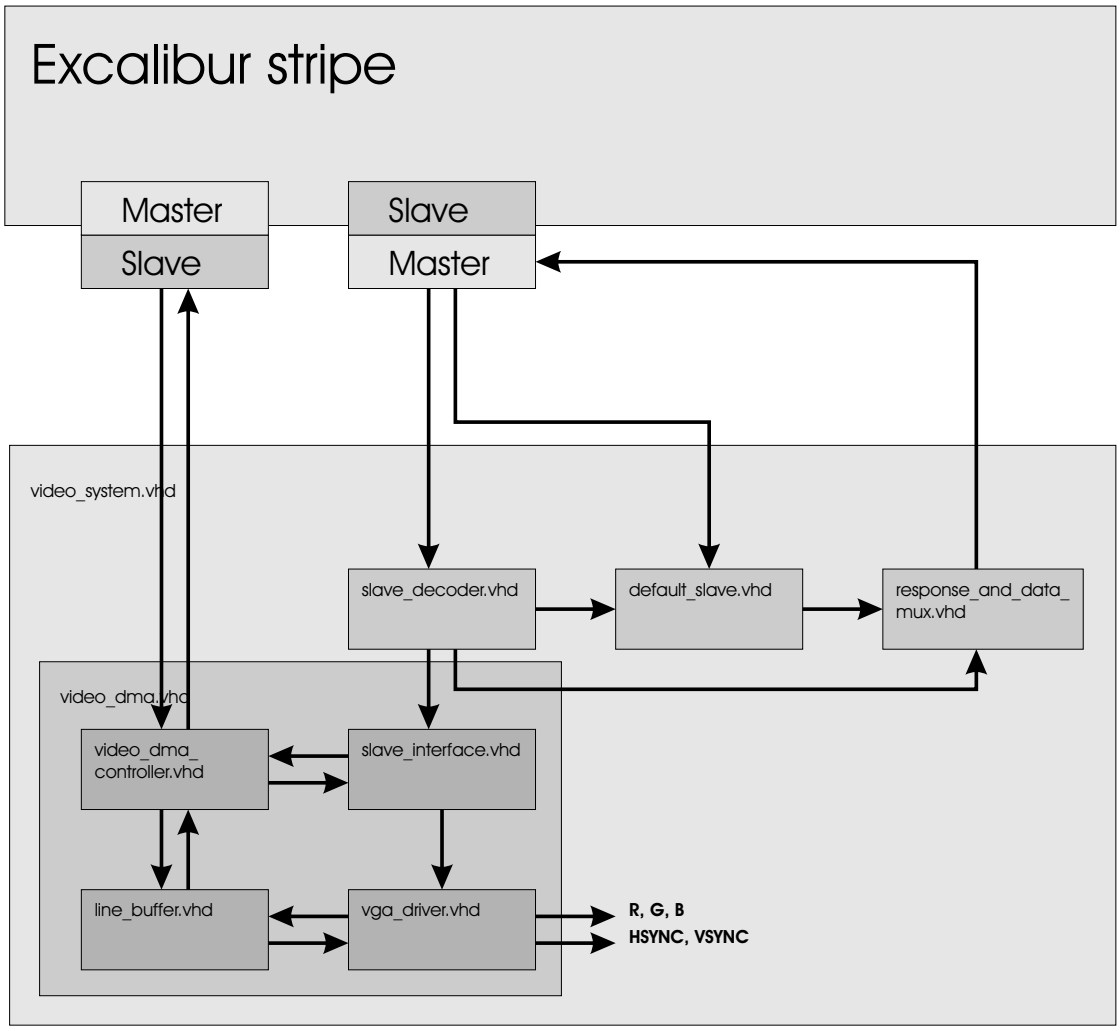

Figure 5.1: Hardware structure of the VGA Slideshow and the file hierarchy of the VHDL files.

Functional description

The organization of the Excalibur device is presented in Chapter 1. A schematic representation of this organization is presented in Figure 1.1. Given this organization, the DMA controller (master) is connected to the slave interface of the PLD-to-stripe bridge. The master initializes the bridge to receive data from the framebuffer, which is located in SDRAM. In order to initialize the PLD-to-stripe bridge, the master has to set signals which indicate the data size and burst type. The burst type can be one of the following:

- Single transfer.
- Incrementing burst of unspecified length.
- 4-beat wrapping burst.

- 4-beat incrementing burst.
- 8-beat wrapping burst.
- 8-beat incrementing burst.
- 16-beat wrapping burst.
- 16-beat incrementing burst.

A beat is a transfer of data packets, thus a 4-beat wrapping burst is a transfer of 4 packets. Incrementing bursts access sequential locations, such that the address of each transfer in the burst is just an increment of the previous address. Wrapping burst also access sequential locations, but are wrapped at 16-byte boundaries. For example, if the start address of a 4-beat wrapping burst with a data size of 4 bytes is 0x34, four transfers occur on addresses 0x34, 0x38, 0x3C and 0x30. The DMA controller uses the incrementing burst of unspecified length in order to fetch an entire video line in one burst. This increases the performance of the hardware design, because access to the AMBA bus is negotiated once. The video line is then stored in the line buffer (`line_buffer.vhd`). The VGA driver reads the pixel data from the line buffer and generates the appropriate VGA signals. The driver always transmits 640x480 frames with a refresh rate of 60Hz. However, different framebuffer sizes can be used with this driver. Information about the framebuffer size and the base address is transmitted by the software into `slave_interface.vhd`. As stated before, the `slave_interface.vhd` contains the registers in which this information is stored. The registers of `slave_interface.vhd` are presented in Table 5.1.

Register name	Address offset	Description
`BUFFER_ADDRESS`	0x00	This register contains the 32-bit base address of the framebuffer.
`IMAGE_DIMENSIONS`	0x04	Bits 15 : 0 contain the total number of lines. Bits 31 : 16 contain the number of pixels in each line.
`CONTROL`	0x08	Bit 0 is used to enable the DMA controller: 1 = enable, 0 = disable.
`CURRENT_ADDRESS`	0x0C	This register contains the 32-bit address from which the DMA controller reads data.
`STATUS`	0x10	Bit 1 of this register contains the status of the horizontal blanking signal, bit 2 contains the status of the vertical blanking signal.

Table 5.1: Register information of the VGA Slideshows.

The address offset is the offset with respect to the Stripe-to-PLD bridge address, which is 0x80000000 in the VGA Slideshow. The `slave_decoder.vhd`, `default_slave.vhd` and `response_and_data_mux.vhd` files are functioning as an interface between the Stripe-to-PLD bridge and the `slave_interface.vhd` register bank, such that the software can read and write the contents of the register bank.

The software initializes the hardware by writing the base address of the framebuffer to the `BUFFER_ADDRESS` register, writing the framebuffer size into the `IMAGE_DIMENSIONS` register and writing a 1 into bit 0 of the `CONTROL` register. After the DMA controller has transmitted the last pixel of the frame to the line buffer, an interrupt is generated.

This interrupt signal is connected to the stripe, such that the software running on the ARM processor is able to handle this interrupt. The interrupt handler of the software contains a counter, which is used to count the number of interrupts (frames). If the counter has reached a value of '100', the interrupt handler changes the base address of the framebuffer by writing the new base address into register `BUFFER_ADDRESS`. After that, the software continues with interrupt counting. This is done two times (with couter values '200' and '300'), such that three pictures are displayed on the VGA monitor. The interrupt handler then resets the counter and writes changes the base address of the framebuffer to its initial value.

All VHDL files can be found in Appendix F. The software uses the same files as the UART testbench, which is described in Chapter 3. However, the `main.c` and `irq.c` files are different, so they can also be found in Appendix F. In order to fill the SDRAM memory with images, a Perl script was used which converts BMP images into Intel HEX format files. The next three paragraphs presents the script.

BMP2HEX Perl script

A `BMP2HEX` Perl script, which is supplied by Altera, was modified in order to produce a Intel HEX format file which contains 8-bit pixels instead of 16-bit pixels. The Perl script uses a 24-bit BMP image with a resolution of 640x480 as an input, strips the least significant bits of the pixels and uses the remaining bits to produce a Intel HEX file which contains 8-bit pixels (640x480).

The generated Intel HEX format file contains a base address, which indicates the memory location where the pixels of the image start. This base address can be altered by modifying the Perl script. The line `my $temp = 48 + $_[0];` in the `hex_address_line` subroutine can be used to alter the base address by changing the number `48` into a multiple of 16: 16 indicates a base address of 0x100000, 32 indicates a base address of 0x200000, and so on.

After converting the BMP images, the Intel HEX files can be linked to the bootloader. If the bootloader starts on power-up, it copies the images from Flash memory to SDRAM memory. The Perl script is presented in Appendix F.

Restrictions

The DMA controller of the VGA Slideshow uses locked transactions to fetch data from the SDRAM. A locked transaction locks the AMBA bus to provide a master exclusive access to the bus. Since the VGA Slideshow requires a lot of memory transfers, the AMBA bus is locked for a significant amount of time. This leaves less bandwidth to the processor, which implies that updating the framebuffer from software can cause problems. However, the VGA Slideshow was tested on an Excalibur device with the lowest speed grade. Using a faster device will increase the bandwidth, because the AMBA bus on a faster device can run on higher frequencies.

5.2 DAMP Gamepack

The DAMP Gamepack is intended to be a reference design for run-time reconfiguration of the PLD of the Excalibur device. The PLD can be reconfigured from a software program with one of four classic games: Minesweeper, Pong, Snake and Riverrun. The software program is based on the Germs monitor, which is generated by the Altera SOPC Builder software [21]. The Germs monitor can be used to read and write from memory and is able to display the memory contents on a terminal program via the UART interface. The games are implemented entirely in VHDL, thus the Germs monitor does not interact with these games. Four instructions were added to the Germs monitor, in order to reconfigure the PLD with the four VHDL games.

For PLD reconfiguration, the software program includes the Serial Binary Image (SBI) files of the four games. This is done in an assembly file (`sbi_data.s`) by using the `INCBIN` statement. To indicate the *start* and *end* addresses of the SBI file, additional labels were added. In assembly this is represented as follows:

```
start_1_sbi

    INCBIN sbi/minesweeper.sbi

end_1_sbi
```

The file `germs_monitor.c` is the main program, from which the instructions are issued that reconfigure the PLD of the Excalibur device. These instructions call the `configure_pld` function located in the `config_logic.c` file, which configures the PLD of the Excalibur device. Figure 5.2 presents the flowchart of `configure_pld` function. This function starts by sending a value to the `CONFIG_UNLOCK` register to unlock the Configuration logic, which is located in the stripe. Then the `configure_pld` function checks if the Configuration logic is unlocked, by checking the value of the `LK` (locked) bit of register `CONFIG_CONTROL`. If it is locked, the function returns an error code. If it is unlocked, the `configure_pld` function sets the clock divider of the Configuration logic, which requires an operating frequency of 16MHz or less. Then the `IDCODE` is checked, which is located in the SBI file. This code indicates the device type and should be equal to the code of Excalibur EPXA1 device. If the `IDCODE` is not correct, the `configure_pld` function should not configure the PLD with this SBI file. Hence, the function returns an error code. If it is correct, the `configure_pld` function enables the Configuration logic. The Configuration logic is enabled by setting the `CO` bit of register `CONFIG_CONTROL`. The initialization phase is ended when the B (busy) bit of register `CONFIG_CONTROL` is cleared.

After the initialization phase, the Configuration logic is ready to receive data from the SBI file. An SBI file contains a byte stream, which should be send to register `CONFIG_DATA`. The `configure_pld` function starts a timer and then sends four bytes to register `CONFIG_DATA`. The Configuration logic then starts configuring the PLD of the Excalibur device. The `configure_pld` function waits until the busy bit of register `CONFIG_CONTROL` is cleared and then reads out the timer to check if a timeout occurred. If a timeout occurs, the function returns an error code. If a timeout doesn't occur,

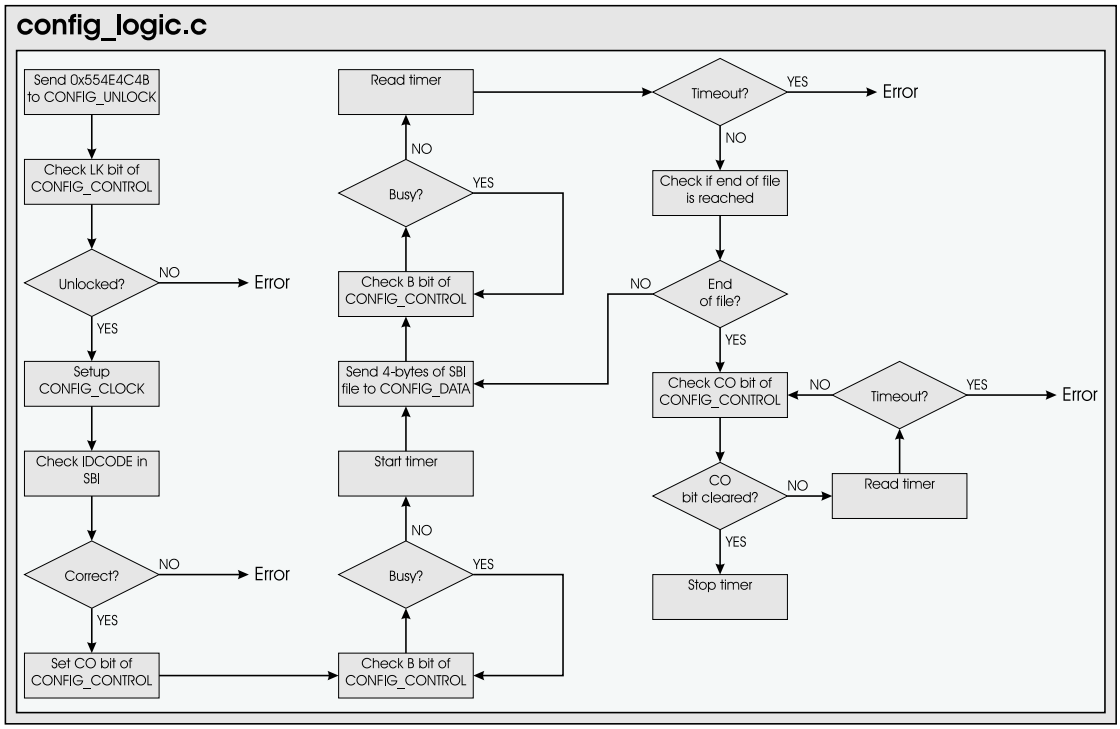

Figure 5.2: Flowchart of the configuration process.

the `configure_pld` function checks if the end of the SBI file is reached. If not, the `configure_pld` function sends the next four bytes to register `CONFIG_DATA`. The Configuration logic then continues the configuration of the PLD. The `configure_pld` function waits until the busy bit of register `CONFIG_CONTROL` is cleared and then reads out the timer again to check if a timeout occurred. This loop is executed until the end of the SBI file has been reached. If the end of the SBI file is reached, the `configure_pld` function waits until the `CO` bit of register `CONFIG_CONTROL` is cleared or until a timeout occurs. If a timeout occurs, the function returns an error code. Else, the function stops the timer and returns the ok code.

As stated before, the software is based on the Germs monitor, which is generated by the Altera SOPC Builder software. Hence, only the modified `germs_monitor.c` file and the other application specific files are presented in Appendix G.

5.3 DAMP Linux

The operating requirements of an embedded application are often simple enough, that there is no need for an operating system. However, some of today's embedded applications are quite complex, requiring software to manage a large number of tasks and hardware devices simultaneously. Linux is becoming a popular operating system for embedded systems. Additionally it is open source, thus it can be ported to new platforms

with relative low effort. In order to port Linux to the DAMP board, the following steps should be performed:

1. A bootloader has to be created for DAMP, or has to be ported from a existing bootloader to DAMP. The bootloader performs the initialization of the Excalibur device and additionally boots the Linux kernel.

2. The Linux kernel has to be ported to DAMP, together with drivers for DAMP devices, such as Flash and UART.

3. Linux requires a root filesystem, i.e., a filesystem used for storage of data and proper booting of Linux. In addition both the bootloader and Linux kernel should be stored on Flash memory for stand-alone operation of DAMP.

In order to compile both the bootloader and Linux kernel, a cross-compiler tool is required. The compiler used for DAMP bootloader and Linux kernel is a precompiled toolchain from Armlinux [22]. The Linux distribution used for compiling the bootloader and Linux kernel is Redhat Linux 8.0 [23] which contains the standard GCC compiler toolchain used in combination with the cross-compiler.

DAMP bootloader

To boot the Linux kernel, a bootloader is required. It is possible to develop a custom bootloader for DAMP, however some advanced bootloaders are already present, like BLOB [24] and ARMBoot [25]. ARMBoot is chosen as bootloader for DAMP, since it already supports the ARM9 processor and EPXA1 development board [26]. Furthermore, ARMBoot supports multiple types of Flash memory, network download via bootp, dhcp, tftp, PCMCIA CF booting and more. Compared to the port of the EPXA1 development board available in ARMBoot the following changes have been introduced to the ARMBoot source-code in order to support correct initialization of DAMP:

- The EPXA1 development board has 16MB SDRAM, while DAMP supports up to 512MB SDRAM. The DAMP prototype consists of the EPXA1 with speed grade -3, which has a maximum clock frequency of 83MHz. In order to support all speed grades and 512MB SDRAM, the SDRAM parameters have been changed to reflect the new situation.

- The DAMP prototype uses a different processor clock frequency. In order to support this, the frequency parameters have been adjusted to reflect this.

- The EPXA1 development board has support for the SMC911 Ethernet controller connected to the EBI. The support for this controller has been disabled in the DAMP port.

- The EPXA1 development board supports up to two Flash devices, DAMP supports up to four Flash devices. Additional support for this has been added.

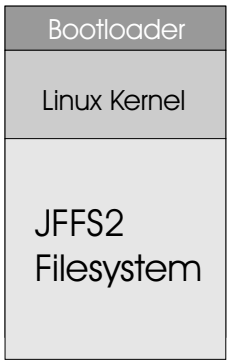

Figure 5.3: Layout used for Flash device.

There is still one issue which requires to be addressed, namely support for the CS8900a Ethernet controller should be implemented in the ARMBoot. Support for the CS8900a Ethernet controller is available in the ARMBoot, hence the driver only requires porting to DAMP.

Linux Kernel

As stated before, the Linux kernel should be ported to DAMP. There already exists a specific patch for ARM processors, which also enables support for the EPXA1 development board. Compared to the port of the EPXA1 development board the following changes have been made to ARM-Linux:

- The DAMP prototype uses a different processor clock frequency. In order to support DAMP, the frequency parameters have been changed to reflect the new situation.
- The UART baudrate divisor variable, used for determining the UART baudrate, has been given a constant value in order to provide a fully functioning UART interface. This is a work-around and requires attention in later versions.
- Support for the CS8900a Ethernet controller has been enabled, however it is currently untested.

Filesystem

The general method used to boot a processor is with the use of permanent memory, such as a Flash device. There are four possibilities to provide a filesystem for the Linux kernel:

1. The use of a Network File System (NFS). This allows machines to mount a disk partition on a remote machine via a network connection
2. The use of a Integrated Device Electronics (IDE) device, e.g., harddisk or cd-rom devices.

3. The use of a Ramdisk integrated in the Linux kernel.

4. The use of Flash devices for a filesystem.

The CS8900 Ethernet controller is not mounted on the current DAMP prototype. Furthermore the IDE interface is not connected due to the fact that the EPXA1 is mounted on the current DAMP prototype. Thus the first two options cannot be implemented on the current version of the prototype. The disadvantage of the third option is that any changes made to the filesystem will be lost when rebooting DAMP. The last option is therefore chosen to be utilized with DAMP. A common used filesystem for Flash devices, the Journaling Flash File System (JFFS2) [27], is used. It is a log-structured file system designed for use in embedded systems. Any changes made to the filesystem are directly stored on Flash and are permanent even when the power supply is switched off. The memory map of the Flash device is presented in Figure 5.3.

6 Conclusions

The Delft Altera-based Multimedia Platform is meant to produce a low-cost and flexible development platform. DAMP's main target is to provide hardware and software support for the hardware/software co-design paradigm for multimedia applications. The design trajectory of the DAMP project consists of creating a specification, designing a schematic, designing a PCB (Print Circuit Board) design, testing a prototype and finally redesigning the schematic and PCB. The presented work in this thesis is the final step in the DAMP Design Trajectory, in order to correct the design flaws which were made in the initial phase. Additionally, reference designs were created, in order to support future DAMP development. The DAMP specification was discussed, which consists of analysis of the requirements and restrictions, a design space exploration and an overview of the implemented DAMP features. Then, the followed testing methodology was presented, together with the results and the modifications made on the PCB. Furthermore, the redesign process was presented, which consists of redesigning the schematic and PCB. Finally, the reference designs were discussed.

This chapter presents conclusions, highlights the main contributions and overviews some possible future research directions. This chapter is organized as follows. Section 6.1 summarizes the main conclusions of the thesis. Section 6.2 presents the main contributions. Section 6.3 gives some possible future research directions.

6.1 Summary

Chapter 2 introduces the DAMP specification, which is required to design a complex multimedia platform for embedded systems development, such as DAMP. There are five requirements for the DAMP design:

1. **Multimedia support:** DAMP should incorporate video and audio interfaces as well as memory, in order to support multimedia applications.

2. **Excalibur compatibility:** DAMP should be compatible with all 672-pins versions of the Excalibur device family.

3. **Software support:** DAMP should work gluelessly with the Quartuss II development tool and standard Altera download cables.

4. **Nios support:** DAMP should provide a natural Nios-to-Excalibur migration path for any Nios customer developed Nios extension board.

5. **Testability:** DAMP should support a variety of testing capabilities.

The DAMP design process has only one restriction, namely that the DAMP platform is intended to be a low-cost development platform. A design space exploration was performed and it was decided to use either the EPXA1 or the EPXA4 Excalibur device. Furthermore, it was decided to incorporate the following features on the DAMP board:

- **Power circuit:** An external ATX Power Supply and a 1.8V voltage regulator.
- **Clock circuit:** Five clock oscillators with enable/disable function (jumpers).
- **Reset circuit:** A reset button for a manual reset and a power-on-reset component.
- **Connectors:** Access to the dedicated pins and I/O pins of the Excalibur device.
- **JTAG interface:** Two connectors to support serial JTAG programming and simultaneous JTAG programming.
- **User I/O interface:** Four push-buttons, one 8-switch dipswitch bank, eight LEDS and two 7-segment displays.
- **UART interface:** One dedicated UART interface.
- **PS/2 interface:** One PS/2 connector.
- **Flash memory:** Support of up to four AMD compatible (TLC320AD77C in DAMP's case) NOR Flash devices.
- **SDRAM interface:** 168-pins DIMM socket connected to the SDRAM controller of the Excalibur device.
- **High quality audio in/out (stereo):** TLC320AD77C Audio CODEC connected to the PLD.
- **Video out:** Support of up to 256 colors through a VGA connector and a simple DAC.
- **Ethernet support:** Cirus Logic CS8900A Ethernet controller connected to the EBI of the Excalibur device.
- **Nios daughter card interface:** Nios daughter card interface compatible part and an additional DAMP extension header.
- **IDE interface:** IDE header connected to the PLD of the Excalibur device.

In Chapter 3 the testing methodology, which was used to test a prototype of the DAMP design, was presented. All features were tested by closely following the testing methodology. The order in which the other features were tested is as follows:

1. Power circuit.
2. Clock circuit
3. Reset circuit.

4. Excalibur configuration selector circuits.
5. JTAG interface.
6. User I/O.
7. Flash memory.
8. SDRAM.
9. UART interface.
10. VGA.
11. Audio.
12. PS/2
13. Daughter card interface (Nios compatible part only).

The results were analyzed in order to correct any design flaw. All design flaws were successfully corrected by modifying the DAMP prototype. The result was a functional prototype. The following features were not tested:

- The Multi-ICE connector of the JTAG interface.
- The Ethernet controller.
- The DAMP extension header of the Daughter card interface.
- The IDE interface.

A speed verification was also not performed. Furthermore, a more thorough examination of the 32-bit mode of the SDRAM circuit and the Audio circuit were also not performed.

In Chapter 4 all design flaws, which were found during the testing process, were corrected by redesigning the DAMP schematic and the DAMP PCB. The following features were redesigned:

1. **Power circuit:**
   - `VCC_CKLK2` was connected to a voltage level of 1.8V.
   - `VCC_CKLK4` was connected to a voltage level of 1.8V.
   - `VCC_CKOUT1` was connected to a voltage level of 3.3V.
   - `VCC_CKOUT1` was connected to a voltage level of 3.3V.
   - All vias, which were located inside the footprint of the WH25 high power resistor, were relocated.
   - All traces, which were routed through the footprint of the WH25 high power resistor, were rerouted.

- The silkscreen that indicated the `Off` and `On` position of the power button, were corrected.

2. **Clock circuit:**

   - The silkscreen that indicated the `Off` and `On` position of the enable/disable jumpers, was corrected.

3. **Reset circuit:**

   - The value of capacitor C98, which was located in the debounce circuit, was changed into 100nF.

4. **User I/O:**

   - A buffer capacitor was added to the power connection of the 74LV14 hex invertor.
   - The connections of the dipswitch bank were altered, such that the `On` indication is correct.
   - The silkscreen of the push-buttons was altered, such that the push-button numbering is correct.

5. **Flash memory:**

   - Flash addresses 0 to 23 were connected to EBI addresses 1 to 24.

6. **UART interface:**

   - The footprint of the MAX3241 was resized.

7. **VGA:**

   - The 7407 open-collector hex buffer was replaced by a 7414 hex invertor.
   - The `VGA_R3` PCB signal was connected to `L2`.
   - The `VGA_R2` PCB signal was connected to `J3`.
   - The `VGA_R1` PCB signal was connected to `K2`.
   - The `VGA_G3` PCB signal was connected to `H4`.
   - The `VGA_G2` PCB signal was connected to `J5`.
   - The `VGA_G1` PCB signal was connected to `H5`.

8. **PS/2:**

   - Resistors were added to the data line and clock line of the PS/2 interface.

9. **Daughter card interface:**

   - Pin `nios40_27` was connected to `AE16`.
   - Pin `nios40_32` was connected to `AC21`.

Chapter 5 presented the three reference designs, which support future DAMP development. The VGA Slide show is a hardware/software design, which is based on a DMA controller. The DMA controller fetches data from SDRAM and generates the appropriate VGA signals, such that a picture can be shown on a VGA monitor. The software program alters the base address of the framebuffer after a predefined interval.

The DAMP Gamepack is a reference design which shows how the PLD of the Excalibur device can be reconfigured in run-time, by a software program running on the ARM processor within the stripe of the Excalibur device. Several games are included, which can be programmed into the PLD of the Excalibur device.

The DAMP Linux design offers support for future Linux development. Linux development can incorporate the development of device drivers for possible future daughter cards.

6.2 Main Contributions

The main contributions can be summarized as follows:

- The DAMP project was initiated and the requirements and restrictions for the DAMP project were created. The main target for the DAMP project was also established, namely to create a low cost development platform for the embedded systems specification and hardware-software co design with the main focus on (mobile) multimedia applications.

- A prototype of the DAMP design was created according to the DAMP design.

- A prototype of the DAMP design was tested by closely following the proposed testing methodology. Design flaws were corrected by modifying the DAMP prototype.

- Some of the parts of the DAMP schematic and PCB were redesigned. The test results were used as a guideline.

- Reference designs were created in order to support future DAMP development.

6.3 Future Directions

As stated before, the DAMP prototype was tested by closely following the proposed testing methodology. However some features were not tested or were not tested completely. Future work should therefore incorporate the following:

- Testing the Multi-ICE connector of the JTAG interface.

- Testing the Ethernet controller.

- Testing the DAMP extension header of the Daughter card interface.

- Testing the IDE interface.

- Testing the 32-bit mode of the SDRAM controller.

- Testing the audio interface.
- Performing a speed verification.
- Redesigning the DAMP schematic and PCB according to the newly acquired results.
- Enhancing DAMP Linux, such that it can mount the JFFS2 filesystem correctly.

Furthermore, there are some future research directions for DAMP. These research directions can further enhance the design and can provide more flexibility and support. The future research directions are:

- Another Ethernet controller can be implemented, which will offer 10/100 Mbit Ethernet.
- CS8900A support, or support for another Ethernet controller, should be enabled in DAMP Linux and ARMBoot.
- Daughter cards should be developed to add additional features, e.g, Bluetooth, USB and Wireless LAN.
- Device drivers should be added to DAMP Linux, in order to support these daughter cards.

Bibliography

[1] Altera Corporation. *EPXA10 Development Board - Hardware Reference Manual*, 2002.

[2] Xilinx Corporation. *Virtex-II Pro Platform FPGAs - Introduction and Overview*, 2003.

[3] Altera Corporation. *NIOS Development Kit Getting Started - User Guide*, 2002.

[4] Altera Corporation. *ARM-Based Embedded Processor PLDs Hardware Reference Manual v3.0*, 2002.

[5] ARM Limited. *AMBA Specification*, 1999.

[6] Altera Corporation. *APEX 20K Programmable Logic Device Family - Data Sheet*, 2002.

[7] Altera Corporation. *Introduction to Quartus II - Manual*, 2003.

[8] JTAG. *JTAG Specification*, 2003. http://www.jtag.com/.

[9] Altera Corporation. *MasterBlaster Serial/USB Communications Cable - Data sheet*, 2003.

[10] Intel Corporation. *Common Flash Interface (CFI) and Command Sets*, 2000.

[11] Toshiba. *64-MBIT CMOS FLASH Memory - Data sheet*, 2001.

[12] JEDEC. *JEDEC Specification*, 2003. http://www.jedec.org/.

[13] Texas Instruments. *TLC320AD77C 24-bit 96 KHz Stereo Audio Codec - Data Manual*, 1999.

[14] Cirrus Logic. *CS8900A - Product Data Sheet*, 2001.

[15] Intel Corporation. *ATX Power Supply Guide v1.2*, 2000.

[16] Welwyn Components. *Aluminium House Wireround Resistors - Data sheet*, 2003.

[17] Altera Corporation. *Using the Expansion Bus Interface - Application Note 143*, 2003.

[18] Texas Instruments. *SN5407, SN5417, SN7407, SN7417 Hex buffers/drivers with open-collector high-voltage outputs - Datasheet*, 1997.

[19] Altera Corporation. *5V Tolerance in APEX 20KE devices - White paper*, 2000.

[20] Altera Corporation. *Using the Embedded Stripe Bridges - Application Note 142*, 2002.

[21] Altera Corporation. *SOPC Builder User Guide*, 2003.

[22] The ARM Linux Project. *ARMLinux*. http://www.arm.linux.org.uk.

[23] Red Hat, Inc. *Red Hat*. http://www.redhat.com.

[24] TU Delft. *Blob, the boot loader*. http://www.lart.tudelft.nl/lartware/blob.

[25] Marius Groeger, Robert Kaiser. *ARMBoot*. http://armboot.sourceforge.net.

[26] Altera Corporation. *EPXA1 Development Board - Hardware Reference Manual*, 2002.

[27] Red Hat, Inc. *JFFS: The Journalling Flash File System*.

User I/O file listings

A

decoder.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY decoder IS
    PORT(
            bcd : in STD_LOGIC_VECTOR(3 DOWNTO 0);
            segment_1 : out STD_LOGIC_VECTOR(7 downto 0);
            segment_2 : out STD_LOGIC_VECTOR(7 downto 0)
        );
END decoder;

ARCHITECTURE arch_decoder OF decoder IS
BEGIN
    decodeer:
    PROCESS (bcd)
    BEGIN
        CASE bcd IS
            WHEN "0000" => -- 0
                segment_1 <= "00000011";
                segment_2 <= "00000011";
            WHEN "0001" => --  1
                segment_1 <= "10011111";
                segment_2 <= "10011111";
            WHEN "0010" => --  2
                segment_1 <= "00100101";
                segment_2 <= "00100101";
            WHEN "0011" => --  3
                segment_1 <= "00001101";
                segment_2 <= "00001101";
            WHEN "0100" => --  4
                segment_1 <= "10011001";
                segment_2 <= "10011001";
            WHEN "0101" => --  5
                segment_1 <= "01001001";
                segment_2 <= "01001001";
            WHEN "0110" => --  6
                segment_1 <= "01000001";
                segment_2 <= "01000001";
            WHEN "0111" => --  7
                segment_1 <= "00011111";
                segment_2 <= "00011111";
            WHEN "1000" => --  8
                segment_1 <= "00000001";
                segment_2 <= "00000001";
```

```
            WHEN "1001" => --  9
                segment_1 <= "00011001";
                segment_2 <= "00011001";
            WHEN "1010" => -- 10
                segment_1 <= "00000010";
                segment_2 <= "00000010";
            WHEN "1011" => -- 11
                segment_1 <= "10011110";
                segment_2 <= "10011110";
            WHEN "1100" => -- 12
                segment_1 <= "00100100";
                segment_2 <= "00100100";
            WHEN "1101" => -- 13
                segment_1 <= "00001100";
                segment_2 <= "00001100";
            WHEN "1110" => -- 14
                segment_1 <= "10011000";
                segment_2 <= "10011000";
            WHEN OTHERS => -- 15
                segment_1 <= "01001000";
                segment_2 <= "01001000";
        END CASE;
    END PROCESS decodeer;
END arch_decoder;
```

arm_pld.s

```
    AREA init, CODE, READONLY
    b   Start

Start

    ldr     r0, =0x00010000

Loop
    ldr     r3, =0x00000000
    str     r3, [r0]
    ldr     r3, =0x00000001
    str     r3, [r0]
    ldr     r3, =0x00000002
    str     r3, [r0]
    ldr     r3, =0x00000003
    str     r3, [r0]
    ldr     r3, =0x00000004
    str     r3, [r0]
    ldr     r3, =0x00000005
    str     r3, [r0]
    ldr     r3, =0x00000006
    str     r3, [r0]
    ldr     r3, =0x00000007
    str     r3, [r0]
    ldr     r3, =0x00000008
    str     r3, [r0]
    ldr     r3, =0x00000009
    str     r3, [r0]
```

```
ldr     r3, =0x0000000A
str     r3, [r0]
ldr     r3, =0x0000000B
str     r3, [r0]
ldr     r3, =0x0000000C
str     r3, [r0]
ldr     r3, =0x0000000D
str     r3, [r0]
ldr     r3, =0x0000000E
str     r3, [r0]
ldr     r3, =0x0000000F
str     r3, [r0]
b       Loop


END
```

UART file listings

B

armc_startup.s

```
;/*****************************************************************************
;*
;*      Code to startup the ARM C environment
;*      =====================================
;*
;* This file contains the code to initialise the C environment before
;* branching to __main which is ARM's C library initialisation routine.
;* For more information on ARM's C libraries see chapter 4 of the ARM
;* Developer Suite Tools manual.
;*
;* The c initialisation includes:
;* 1. Turn on the instruction cache
;* 2. Turn on the instruction cache and MMU, see below for details on the
;*    mapping
;* 3. Setup the stack for all modes
;* 4. Initialise the UART
;* 5. Switch to User mode with IRQ's enabled, FIQs disabled
;* 6. Branch to __main
;
;*****************************************************************************/

    IMPORT CIrqHandler
    IMPORT CFiqHandler
    IMPORT CPabtHandler
    IMPORT CDabtHandler
    IMPORT CSwiHandler
    IMPORT CUdefHandler
    IMPORT uart_init

    GET stripe.s

    ; --- Standard definitions of mode bits and interrupt (I & F)
    ; --- flags in PSRs

Mode_USR        EQU     0x10
Mode_FIQ        EQU     0x11
Mode_IRQ        EQU     0x12
Mode_SVC        EQU     0x13
Mode_ABT        EQU     0x17
Mode_UNDEF      EQU     0x1B
Mode_SYS        EQU     0x1F ; available on ARM Arch 4 and later

I_Bit           EQU     0x80 ; when I bit is set, IRQ is disabled
F_Bit           EQU     0x40 ; when F bit is set, FIQ is disabled
```

```
; --- System memory locations
; We have mapped 128K of SRAM at address 0x20000 I'm using the top of
; this memory as a stack
RAM_Limit       EQU     EXC_SPSRAM_BLOCK1_BASE + EXC_SPSRAM_BLOCK1_SIZE
SVC_Stack       EQU     RAM_Limit           ; 8K SVC stack at top of memory
IRQ_Stack       EQU     RAM_Limit-8192      ; followed by  1k IRQ stack
ABT_Stack       EQU     IRQ_Stack-1024      ; followed by  1k ABT stack
FIQ_Stack       EQU     ABT_Stack-1024      ; followed by  1k FIQ stack
UNDEF_Stack     EQU     FIQ_Stack-1024      ; followed by  1k UNDEF stack
USR_Stack       EQU     UNDEF_Stack-1024    ; followed by  USR stack


;
;   If booting from flash the entry point Start is not arrived at immediately
;   after reset the quartus project file is doing a few things under your feet
;   that you need to be aware of.
;
;   The Excalibur Megawizard generated a file (.sbd) which contains the
;   information about the setup you requested for your memory map, IO
;   settings, SDRAM settings etc.
;
;   This file, along with your hex file and PLD image is converted to an object
;   file, and compressed in the process.
;
;   This object file is linked with Altera's boot code. Altera's boot code
;   then configures excalibur's registers to give the setup you requested
;   via the MegaWizard, it uncompresses the PLD image and the hex file and
;   loads them.
;
;   So at this point your memory map should be setup and contain the memory
;   you initially requested.
;
;   For more information on this flow please see the document
;   Toolflow for ARM-Based Embedded Processor PLDs
;

    AREA init, CODE, READONLY
    b   Start
    b   UdefHnd
    b   SwiHnd
    b   PabtHnd
    b   DabtHnd
    b   Unexpected
    b   IrqHnd
    b   FiqHnd

Unexpected
    b   Unexpected

UdefHnd
    stmdb   sp!,{r0-r12,lr}
    bl  CUdefHandler
    ldmia   sp!,{r0-r12,lr}
```

```
    subs    pc,lr,#4

SwiHnd
    stmdb   sp!,{r0-r12,lr}

    ; put the swi argument in r0 and call
    ; CSwiHandler

    sub r0, lr, #4
    ldr r0, [r0]
    mvn r1, #0xff000000
    and r0, r0, r1
    bl CSwiHandler
    ldmia   sp!,{r0-r12,lr}
    movs        pc, lr

IrqHnd
    stmdb   sp!,{r0-r12,lr}
    bl  CIrqHandler
    ldmia   sp!,{r0-r12,lr}
    subs    pc,lr,#4

PabtHnd
    stmdb   sp!,{r0-r12,lr}
    bl  CPabtHandler
    ldmia   sp!,{r0-r12,lr}
    subs    pc,lr,#4

DabtHnd
    stmdb   sp!,{r0-r12,lr}
    bl  CDabtHandler
    ldmia   sp!,{r0-r12,lr}
    subs    pc,lr,#4

FiqHnd
    stmdb   sp!,{r0-r7,lr}
    bl  CFiqHandler
    ldmia   sp!,{r0-r7,lr}
    subs    pc,lr,#4

Start

    ;/* Turn on the instruction cache, Random replacement*/ ;
    MRC     p15,0,r0,c1,c0,0
    LDR r1,=0x1078
    ORRS    r0,r0,r1
    MCR     p15,0,r0,c1,c0,0

    LDR r0,=0xFFFFFFF7      ; set all domains to be manager, except domain 1
    MCR p15,0,r0,c3,c0,0

    ; Enable the MMU and DCache
    LDR r0,=page_table
    MCR p15,0,r0,c2,c0,0    ; set TTB address
```

```
    MRC     p15,0,r0,c1,c0,0
    ORR     r0,r0,#5         ; enable DCache and MMU
    MCR     p15,0,r0,c1,c0,0


    ; --- Initialise stack pointer registers
; Enter SVC mode and set up the SVC stack pointer
        MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit ; No interrupts
        LDR     SP, =SVC_Stack

; Enter IRQ mode and set up the IRQ stack pointer
        MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit ; No interrupts
        LDR     SP, =IRQ_Stack

; Enter FIQ mode and set up the FIQ stack pointer
        MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit ; No interrupts
        LDR     SP, =FIQ_Stack

; Enter UNDEF mode and set up the UNDEF stack pointer
        MSR     CPSR_c, #Mode_UNDEF:OR:I_Bit:OR:F_Bit ; No interrupts
        LDR     SP, =UNDEF_Stack

; Enter ABT mode and set up the ABT stack pointer
        MSR     CPSR_c, #Mode_ABT:OR:I_Bit:OR:F_Bit ; No interrupts
        LDR     SP, =ABT_Stack


; --- Initialise memory system
        ; ...

; --- Initialise critical IO devices
        ; ...

; --- Initialise interrupt system variables here
        ; ...

; --- Initialise critical IO devices

        BL uart_init

; --- Initialise interrupt system variables here

; --- Now change to User mode and set up User mode stack.
        MSR     CPSR_c, #Mode_USR
        LDR     SP, =USR_Stack

        IMPORT  __main


; --- Now enter the C code
        B      __main   ; note use B not BL, because an application will
                        ; never return this way

;--- Page table AREA
;
```

```
; General comment
;
; The page table below is configurred to use 1Mb sections to configure the
; MMU, it is set up as follows:

; 0 - 0x5fff ffff
; This area has the cache turned on in write-back mode there is a one to one
; mapping between virtual and physical addresses

; 0x6000 0000 - 0x6fff ffff
; This area has the cache off, it is set asside for PLD region 1

; 0x7000 0000 - 0x7fef ffff
; This area has the cache turned on in write-back mode there is a one to one
; mapping between virtual and physical addresses

; 0x7ff0 0000 - 0x7fff ffff
; This area has the cache turned off, as it contains the registers

; 0x8000 0000 - 0xbfff ffff
; This area has the cache off, it is set asside for PLD region 0

; 0xc000 0000 - 0xffff ffff
; This area has the cache turned on in write-back mode there is a one to one
; mapping between virtual and physical addresses
;
; In the page table below there are two setups used of the Form
;
; (Address << 20) | 0xc1e
; (Address << 20) | 0xc12
;
; The address element is the base address of the 1MByte section, the other none
; zero bits are:
;
; 0xC00 This sets the access permissions so the the section can be accessed from
; User mode and in priviledged modes
; 0x10 This bit must be written as 1
; 0xc  These two bits control whether the cache is enabled and the write buffer
;      is enabled
; 0x2  This bit marks the entry as a section descriptor
;

        AREA    L1_table, DATA, READONLY, ALIGN=14 ; ALign to 16kB
page_table
        GBLA    section
section     SETA    0
        WHILE   section < 1536
        DCD (0x$section:SHL:20):OR:0xC1E ; Flat mapping WB, domain 0.
section     SETA    section + 1
        WEND

;
;   256MByte region for PLD Region1 with the cache disabled
;   This needs doing for any area which contains memory mapped hardware
;
```

```
        WHILE   section < 1792
        DCD (0x$section:SHL:20):OR:0xC12 ; No cache no buffering
section     SETA    section + 1
        WEND

        WHILE   section < 2047
        DCD (0x$section:SHL:20):OR:0xC1E ; Flat mapping WB, domain 0.
section     SETA    section + 1
        WEND

;
;   Register region, turn off the cache and the buffers
;   This needs doing for any area which contains memory mapped hardware
;   It could have been included in the PLD region below, but is left
;   separate
;   for clarity
;
        DCD  0x7FF00C12 ; 1 MByte Register region no cache no buffering
                        ; Strictly I should have a small page table and
                        ; just turn of the cache for the 16k which contains
                        ; the registers, but this is much easier

section     SETA    2048
;
;   1GByte region for PLD Region0 with the cache disabled
;   This needs doing for any area which contains memory mapped hardware
;
        WHILE   section < 3072
        DCD (0x$section:SHL:20):OR:0xC12 ; No cache no buffering
section     SETA    section + 1
        WEND


        WHILE   section < 4096
        DCD (0x$section:SHL:20):OR:0xC1E ; Flat mapping WB, domain 0.
section     SETA    section + 1
        WEND

    END
```

exceptions.c

```
#include <stdio.h>

void CAbtHandler(void)
{
    printf("Data abort\r\n");
}

void CPabtHandler(void)
{
    printf("Error prefetch abort\r\n");
}

void CDabtHandler(void)
```

```
{
    printf("Error data abort\r\n");
}

void CSwiHandler(int swi)
{
    if (swi == 0x123456)
    {
        printf ("Exit\r\n");
    }
    else
    {
        printf("Error swi %x\r\n", swi);
    }
}


void CUdefHandler(void)
{
    printf("Error undefined instruction\r\n");
}
```

int_ctrl00.h

```
/*
 *
 *  This file contains the register definitions for the Excalibur
 *  Interrupnt controller INT_CTRL00.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
 * USA
 */

#ifndef __INT_CTRL00_H
#define __INT_CTRL00_H

#define INT_MS(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x00 ))
#define INT_MS_FC_MSK (0x10000)
#define INT_MS_FC_OFST (16)
#define INT_MS_CR_MSK (0x8000)
#define INT_MS_CR_OFST (15)
#define INT_MS_CT_MSK (0x4000)
#define INT_MS_CT_OFST (14)
```

```
#define INT_MS_AE_MSK (0x2000)
#define INT_MS_AE_OFST (13)
#define INT_MS_PE_MSK (0x1000)
#define INT_MS_PE_OFST (12)
#define INT_MS_EE_MSK (0x0800)
#define INT_MS_EE_OFST (11)
#define INT_MS_PS_MSK (0x0400)
#define INT_MS_PS_OFST (10)
#define INT_MS_T1_MSK (0x0200)
#define INT_MS_T1_OFST (9)
#define INT_MS_T0_MSK (0x0100)
#define INT_MS_T0_OFST (8)
#define INT_MS_UA_MSK (0x0080)
#define INT_MS_UA_OFST (7)
#define INT_MS_IP_MSK (0x0040)
#define INT_MS_IP_OFST (6)
#define INT_MS_P5_MSK (0x0020)
#define INT_MS_P5_OFST (5)
#define INT_MS_P4_MSK (0x0010)
#define INT_MS_P4_OFST (4)
#define INT_MS_P3_MSK (0x0008)
#define INT_MS_P3_OFST (3)
#define INT_MS_P2_MSK (0x0004)
#define INT_MS_P2_OFST (2)
#define INT_MS_P1_MSK (0x0002)
#define INT_MS_P1_OFST (1)
#define INT_MS_P0_MSK (0x0001)
#define INT_MS_P0_OFST (0)

#define INT_MC(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x04 ))
#define INT_MC_FC_MSK (0x10000)
#define INT_MC_FC_OFST (16)
#define INT_MC_CR_MSK (0x8000)
#define INT_MC_CR_OFST (15)
#define INT_MC_CT_MSK (0x4000)
#define INT_MC_CT_OFST (14)
#define INT_MC_AE_MSK (0x2000)
#define INT_MC_AE_OFST (13)
#define INT_MC_PE_MSK (0x1000)
#define INT_MC_PE_OFST (12)
#define INT_MC_EE_MSK (0x0800)
#define INT_MC_EE_OFST (11)
#define INT_MC_PS_MSK (0x0400)
#define INT_MC_PS_OFST (10)
#define INT_MC_T1_MSK (0x0200)
#define INT_MC_T1_OFST (9)
#define INT_MC_T0_MSK (0x0100)
#define INT_MC_T0_OFST (8)
#define INT_MC_UA_MSK (0x0080)
#define INT_MC_UA_OFST (7)
#define INT_MC_IP_MSK (0x0040)
#define INT_MC_IP_OFST (6)
#define INT_MC_P5_MSK (0x0020)
#define INT_MC_P5_OFST (5)
#define INT_MC_P4_MSK (0x0010)
```

```
#define INT_MC_P4_OFST (4)
#define INT_MC_P3_MSK (0x0008)
#define INT_MC_P3_OFST (3)
#define INT_MC_P2_MSK (0x0004)
#define INT_MC_P2_OFST (2)
#define INT_MC_P1_MSK (0x0002)
#define INT_MC_P1_OFST (1)
#define INT_MC_P0_MSK (0x0001)
#define INT_MC_P0_OFST (0)

#define INT_SS(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x08 ))
#define INT_SS_FC_MSK (0x8000)
#define INT_SS_FC_OFST (15)
#define INT_SS_CR_MSK (0x8000)
#define INT_SS_CR_OFST (15)
#define INT_SS_CT_MSK (0x4000)
#define INT_SS_CT_OFST (14)
#define INT_SS_AE_MSK (0x2000)
#define INT_SS_AE_OFST (13)
#define INT_SS_PE_MSK (0x1000)
#define INT_SS_PE_OFST (12)
#define INT_SS_EE_MSK (0x0800)
#define INT_SS_EE_OFST (11)
#define INT_SS_PS_MSK (0x0400)
#define INT_SS_PS_OFST (10)
#define INT_SS_T1_MSK (0x0200)
#define INT_SS_T1_OFST (9)
#define INT_SS_T0_MSK (0x0100)
#define INT_SS_T0_OFST (8)
#define INT_SS_UA_MSK (0x0080)
#define INT_SS_UA_OFST (7)
#define INT_SS_IP_MSK (0x0040)
#define INT_SS_IP_OFST (6)
#define INT_SS_P5_MSK (0x0020)
#define INT_SS_P5_OFST (5)
#define INT_SS_P4_MSK (0x0010)
#define INT_SS_P4_OFST (4)
#define INT_SS_P3_MSK (0x0008)
#define INT_SS_P3_OFST (3)
#define INT_SS_P2_MSK (0x0004)
#define INT_SS_P2_OFST (2)
#define INT_SS_P1_MSK (0x0002)
#define INT_SS_P1_OFST (1)
#define INT_SS_P0_MSK (0x0001)
#define INT_SS_P0_OFST (0)

#define INT_RS(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x0C ))
#define INT_RS_FC_MSK (0x10000)
#define INT_RS_FC_OFST (16)
#define INT_RS_CR_MSK (0x8000)
#define INT_RS_CR_OFST (15)
#define INT_RS_CT_MSK (0x4000)
#define INT_RS_CT_OFST (14)
#define INT_RS_AE_MSK (0x2000)
#define INT_RS_AE_OFST (13)
```

```
#define INT_RS_PE_MSK (0x1000)
#define INT_RS_PE_OFST (12)
#define INT_RS_EE_MSK (0x0800)
#define INT_RS_EE_OFST (11)
#define INT_RS_PS_MSK (0x0400)
#define INT_RS_PS_OFST (10)
#define INT_RS_T1_MSK (0x0200)
#define INT_RS_T1_OFST (9)
#define INT_RS_T0_MSK (0x0100)
#define INT_RS_T0_OFST (8)
#define INT_RS_UA_MSK (0x0080)
#define INT_RS_UA_OFST (7)
#define INT_RS_IP_MSK (0x0040)
#define INT_RS_IP_OFST (6)
#define INT_RS_P5_MSK (0x0020)
#define INT_RS_P5_OFST (5)
#define INT_RS_P4_MSK (0x0010)
#define INT_RS_P4_OFST (4)
#define INT_RS_P3_MSK (0x0008)
#define INT_RS_P3_OFST (3)
#define INT_RS_P2_MSK (0x0004)
#define INT_RS_P2_OFST (2)
#define INT_RS_P1_MSK (0x0002)
#define INT_RS_P1_OFST (1)
#define INT_RS_P0_MSK (0x0001)
#define INT_RS_P0_OFST (0)

#define INT_ID(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x10 ))
#define INT_ID_ID_MSK (0x3F)
#define INT_ID_ID_OFST (0)

#define INT_PLD_PRIORITY(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x14 ))
#define INT_PLD_PRIORITY_PRI_MSK (0x3F)
#define INT_PLD_PRIORITY_PRI_OFST (0)
#define INT_PLD_PRIORITY_GA_MSK (0x40)
#define INT_PLD_PRIORITY_GA_OFST (6)

#define INT_PLD_MODE(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x18 ))
#define INT_PLD_MODE_MODE_MSK (0x3)
#define INT_PLD_MODE_MODE_OFST (0)

#define INT_PRIORITY_P0(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x80 ))
#define INT_PRIORITY_P0_PRI_MSK (0x3F)
#define INT_PRIORITY_P0_PRI_OFST (0)
#define INT_PRIORITY_P0_FQ_MSK (0x40)
#define INT_PRIORITY_P0_FQ_OFST (6)

#define INT_PRIORITY_P1(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x84 ))
#define INT_PRIORITY_P1_PRI_MSK (0x3F)
#define INT_PRIORITY_P1_PRI_OFST (0)
#define INT_PRIORITY_P1_FQ_MSK (0x40)
#define INT_PRIORITY_P1_FQ_OFST (6)

#define INT_PRIORITY_P2(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x88 ))
#define INT_PRIORITY_P2_PRI_MSK (0x3F)
```

```
#define INT_PRIORITY_P2_PRI_OFST (0)
#define INT_PRIORITY_P2_FQ_MSK (0x40)
#define INT_PRIORITY_P2_FQ_OFST (6)

#define INT_PRIORITY_P3(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x8C ))
#define INT_PRIORITY_P3_PRI_MSK (0x3F)
#define INT_PRIORITY_P3_PRI_OFST (0)
#define INT_PRIORITY_P3_FQ_MSK (0x40)
#define INT_PRIORITY_P3_FQ_OFST (6)

#define INT_PRIORITY_P4(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x90 ))
#define INT_PRIORITY_P4_PRI_MSK (0x3F)
#define INT_PRIORITY_P4_PRI_OFST (0)
#define INT_PRIORITY_P4_FQ_MSK (0x40)
#define INT_PRIORITY_P4_FQ_OFST (6)

#define INT_PRIORITY_P5(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x94 ))
#define INT_PRIORITY_P5_PRI_MSK (0x3F)
#define INT_PRIORITY_P5_PRI_OFST (0)
#define INT_PRIORITY_P5_FQ_MSK (0x40)
#define INT_PRIORITY_P5_FQ_OFST (6)

#define INT_PRIORITY_IP(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x98 ))
#define INT_PRIORITY_IP_PRI_MSK (0x3F)
#define INT_PRIORITY_IP_PRI_OFST (0)
#define INT_PRIORITY_IP_FQ_MSK (0x40)
#define INT_PRIORITY_IP_FQ_OFST (6)

#define INT_PRIORITY_UA(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x9C ))
#define INT_PRIORITY_UA_PRI_MSK (0x3F)
#define INT_PRIORITY_UA_PRI_OFST (0)
#define INT_PRIORITY_UA_FQ_MSK (0x40)
#define INT_PRIORITY_UA_FQ_OFST (6)

#define INT_PRIORITY_T0(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xA0 ))
#define INT_PRIORITY_T0_PRI_MSK (0x3F)
#define INT_PRIORITY_T0_PRI_OFST (0)
#define INT_PRIORITY_T0_FQ_MSK (0x40)
#define INT_PRIORITY_T0_FQ_OFST (6)

#define INT_PRIORITY_T1(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xA4 ))
#define INT_PRIORITY_T1_PRI_MSK (0x3F)
#define INT_PRIORITY_T1_PRI_OFST (0)
#define INT_PRIORITY_T1_FQ_MSK (0x40)
#define INT_PRIORITY_T1_FQ_OFST (6)

#define INT_PRIORITY_PS(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xA8 ))
#define INT_PRIORITY_PS_PRI_MSK (0x3F)
#define INT_PRIORITY_PS_PRI_OFST (0)
#define INT_PRIORITY_PS_FQ_MSK (0x40)
#define INT_PRIORITY_PS_FQ_OFST (6)

#define INT_PRIORITY_EE(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xAC ))
#define INT_PRIORITY_EE_PRI_MSK (0x3F)
#define INT_PRIORITY_EE_PRI_OFST (0)
```

```
#define INT_PRIORITY_EE_FQ_MSK (0x40)
#define INT_PRIORITY_EE_FQ_OFST (6)

#define INT_PRIORITY_PE(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xB0 ))
#define INT_PRIORITY_PE_PRI_MSK (0x3F)
#define INT_PRIORITY_PE_PRI_OFST (0)
#define INT_PRIORITY_PE_FQ_MSK (0x40)
#define INT_PRIORITY_PE_FQ_OFST (6)

#define INT_PRIORITY_AE(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xB4 ))
#define INT_PRIORITY_AE_PRI_MSK (0x3F)
#define INT_PRIORITY_AE_PRI_OFST (0)
#define INT_PRIORITY_AE_FQ_MSK (0x40)
#define INT_PRIORITY_AE_FQ_OFST (6)

#define INT_PRIORITY_CT(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xB8 ))
#define INT_PRIORITY_CT_PRI_MSK (0x3F)
#define INT_PRIORITY_CT_PRI_OFST (0)
#define INT_PRIORITY_CT_FQ_MSK (0x40)
#define INT_PRIORITY_CT_FQ_OFST (6)

#define INT_PRIORITY_CR(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xBC ))
#define INT_PRIORITY_CR_PRI_MSK (0x3F)
#define INT_PRIORITY_CR_PRI_OFST (0)
#define INT_PRIORITY_CR_FQ_MSK (0x40)
#define INT_PRIORITY_CR_FQ_OFST (6)

#define INT_PRIORITY_FC(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xC0 ))
#define INT_PRIORITY_FC_PRI_MSK (0x3F)
#define INT_PRIORITY_FC_PRI_OFST (0)
#define INT_PRIORITY_FC_FQ_MSK (0x40)
#define INT_PRIORITY_FC_FQ_OFST (6)
#define INT_PRIORITY_IP(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x98 ))
#define INT_PRIORITY_IP_PRI_MSK (0x3F)
#define INT_PRIORITY_IP_PRI_OFST (0)
#define INT_PRIORITY_IP_FQ_MSK (0x40)
#define INT_PRIORITY_IP_FQ_OFST (6)

#define INT_PRIORITY_UA(base_addr) (INT_CTRL00_TYPE (base_addr  + 0x9C ))
#define INT_PRIORITY_UA_PRI_MSK (0x3F)
#define INT_PRIORITY_UA_PRI_OFST (0)
#define INT_PRIORITY_UA_FQ_MSK (0x40)
#define INT_PRIORITY_UA_FQ_OFST (6)

#define INT_PRIORITY_T0(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xA0 ))
#define INT_PRIORITY_T0_PRI_MSK (0x3F)
#define INT_PRIORITY_T0_PRI_OFST (0)
#define INT_PRIORITY_T0_FQ_MSK (0x40)
#define INT_PRIORITY_T0_FQ_OFST (6)

#define INT_PRIORITY_T1(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xA4 ))
#define INT_PRIORITY_T1_PRI_MSK (0x3F)
#define INT_PRIORITY_T1_PRI_OFST (0)
#define INT_PRIORITY_T1_FQ_MSK (0x40)
#define INT_PRIORITY_T1_FQ_OFST (6)
```

```
#define INT_PRIORITY_PS(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xA8 ))
#define INT_PRIORITY_PS_PRI_MSK (0x3F)
#define INT_PRIORITY_PS_PRI_OFST (0)
#define INT_PRIORITY_PS_FQ_MSK (0x40)
#define INT_PRIORITY_PS_FQ_OFST (6)

#define INT_PRIORITY_EE(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xAC ))
#define INT_PRIORITY_EE_PRI_MSK (0x3F)
#define INT_PRIORITY_EE_PRI_OFST (0)
#define INT_PRIORITY_EE_FQ_MSK (0x40)
#define INT_PRIORITY_EE_FQ_OFST (6)

#define INT_PRIORITY_PE(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xB0 ))
#define INT_PRIORITY_PE_PRI_MSK (0x3F)
#define INT_PRIORITY_PE_PRI_OFST (0)
#define INT_PRIORITY_PE_FQ_MSK (0x40)
#define INT_PRIORITY_PE_FQ_OFST (6)

#define INT_PRIORITY_AE(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xB4 ))
#define INT_PRIORITY_AE_PRI_MSK (0x3F)
#define INT_PRIORITY_AE_PRI_OFST (0)
#define INT_PRIORITY_AE_FQ_MSK (0x40)
#define INT_PRIORITY_AE_FQ_OFST (6)

#define INT_PRIORITY_M0(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xB8 ))
#define INT_PRIORITY_M0_PRI_MSK (0x3F)
#define INT_PRIORITY_M0_PRI_OFST (0)
#define INT_PRIORITY_M0_FQ_MSK (0x40)
#define INT_PRIORITY_M0_FQ_OFST (6)

#define INT_PRIORITY_M1(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xBC ))
#define INT_PRIORITY_M1_PRI_MSK (0x3F)
#define INT_PRIORITY_M1_PRI_OFST (0)
#define INT_PRIORITY_M1_FQ_MSK (0x40)
#define INT_PRIORITY_M1_FQ_OFST (6)

#define INT_PRIORITY_FC(base_addr) (INT_CTRL00_TYPE (base_addr  + 0xC0 ))
#define INT_PRIORITY_FC_PRI_MSK (0x3F)
#define INT_PRIORITY_FC_PRI_OFST (0)
#define INT_PRIORITY_FC_FQ_MSK (0x40)
#define INT_PRIORITY_FC_FQ_OFST (6)

#endif /* __INT_CTRL00_H */
```

irq.h

```
#ifndef IRQ_H
#define IRQ_H

#define INT_CTRL00_TYPE (volatile unsigned int *)
void irq_init(void);

#endif /* IRQ_H */
```

irq.c

```
#include <stdio.h>

#include "irq.h"

#include "stripe.h"
#include "int_ctrl00.h"
#include "uartcomm.h"

#define INT_CTRL00_TYPE (volatile unsigned int *)
#define UART_IRQ_PRI 2

void uart_irq_handler(void);

void irq_init(void)
{
    /*
    *   Disable the interrupts for all the PLD sources
    *   confusingly enough these are all on by default
    *   The reason is in case people want to implement their own
    *   interrupt controller in the PLD they don't have to write
    *   any code to enable interrupts in the Excalibur controller
    */
    *INT_MC(EXC_INT_CTRL00_BASE) =  INT_MC_P0_MSK | INT_MC_P1_MSK |
                                    INT_MC_P2_MSK | INT_MC_P3_MSK |
                                    INT_MC_P4_MSK | INT_MC_P5_MSK;

    /*
    * Set priority for the UART interrupts
    */
    *INT_PRIORITY_UA(EXC_INT_CTRL00_BASE)=UART_IRQ_PRI;

    /*
    *   Enable the UART interrupt
    */
    *INT_MS(EXC_INT_CTRL00_BASE)=INT_MS_UA_MSK;
}

void CIrqHandler(void)
{
    volatile int irqID;

    irqID = *INT_ID(EXC_INT_CTRL00_BASE);

    switch (irqID)
    {
    case UART_IRQ_PRI:
        uart_irq_handler();
        break;
    default:
        /* This shouldn't happen, but let's trap it in case */
        printf("Unknown irq %#x",irqID);
        break;
    }
```

```
    return;
}

void CFiqHandler(void)
{
    /* This shouldn't happen */
    return;
}
```

retarget.c

```
/*
*   This provides implementations of _sys_exit(), _ttywrch() and
*   ___stackheap(), which are the bare minimum required to enable
*   use of the ARM C libraries. Only some of the functionality of
*   the libraries is available, you may need to add more low level
*   functions if you get the linker error below.
*
*   Error   : L6200E: Symbol __semihosting_swi_guard multiply
*   defined (by use_semi.o and use_no_semi.o).
*
*   Please see Chapter 4 of the ARM Developer Suite Tools guide
*   for more information
*
*   In addition there are implementations of fputc and fgetc in
*   the file uartcomm.c so printf should be available.
*
*   This is based opon the embed/rom example supplied with the ARM
*   Developer Suite, so it's partly

#include <stdio.h>
#include <rt_misc.h>
#include "uartcomm.h"

extern char tx_buffer[BUFF_SIZE];
extern char rx_buffer[BUFF_SIZE];
extern volatile int tx_head,tx_tail,rx_head,rx_tail;

#ifdef __thumb
/* Thumb Semihosting SWI */
#define SemiSWI 0xAB
#else
/* ARM Semihosting SWI */
#define SemiSWI 0x123456
#endif


/* Exit */
__swi(SemiSWI) void _Exit(unsigned op, unsigned except);
#define Exit() _Exit (0x18,0x20026)

void _sys_exit(int return_code)
{
    Exit();         /* for debugging */
```

```
label:  goto label; /* endless loop */
}

void _ttywrch(int ch)
{
    /*
    *   This function is supposed to output a character to the console
    *   given that we don't have one, we can't really do very much
    */
}

__value_in_regs struct __initial_stackheap __user_initial_stackheap(
        unsigned R0, unsigned SP, unsigned R2, unsigned SL)
{
    extern unsigned int Image$$ZI$$Limit;

    struct __initial_stackheap config;

    /* Start the Heap at the end of the zero initialised data */
    config.heap_base = (unsigned int)&Image$$ZI$$Limit;
    config.stack_base = SP;

    return config;
}

/*
*   Simple implementation of putc
*/
int fputc(int ch, FILE *f)
{
    /* Copy the character into the tx buffer */
    tx_buffer[tx_tail++]=(char)ch;
    tx_tail&=BUFF_MASK;
    if(tx_tail==tx_head)
    {
        return EOF;
    }

    /* Give the transmitter a kick */
    uart_start_tx();

    /* Everything is OK... */
    return 0;
}

int fgetc( FILE *f)
{
    int character;

    while(rx_head==rx_tail);
    character=rx_buffer[rx_head++];
    rx_head&=BUFF_MASK;
    return character;
}
```

uart00.h

```
#ifndef __UART00_H
#define __UART00_H

/*
 * Register definitions for the UART
 */

#define UART_TX_FIFO_SIZE      (15)

#define UART_RSR(base_addr) (UART00_TYPE (base_addr  + 0x00 ))
#define UART_RSR_RX_LEVEL_MSK (0x1f)
#define UART_RSR_RX_LEVEL_OFST (0)
#define UART_RSR_RE_MSK (0x80)
#define UART_RSR_RE_OFST (7)

#define UART_RDS(base_addr) (UART00_TYPE (base_addr  + 0x04 ))
#define UART_RDS_BI_MSK (0x8)
#define UART_RDS_BI_OFST (4)
#define UART_RDS_FE_MSK (0x4)
#define UART_RDS_FE_OFST (2)
#define UART_RDS_PE_MSK (0x2)
#define UART_RDS_PE_OFST (1)
#define UART_RDS_OE_MSK (0x1)
#define UART_RDS_OE_OFST (0)

#define UART_RD(base_addr) (UART00_TYPE (base_addr  + 0x08 ))
#define UART_RD_RX_DATA_MSK (0xff)
#define UART_RD_RX_DATA_OFST (0)

#define UART_TSR(base_addr) (UART00_TYPE (base_addr  + 0x0c ))
#define UART_TSR_TX_LEVEL_MSK (0x1f)
#define UART_TSR_TX_LEVEL_OFST (0)
#define UART_TSR_TXI_MSK (0x80)
#define UART_TSR_TXI_OFST (7)

#define UART_TD(base_addr) (UART00_TYPE (base_addr  + 0x10 ))
#define UART_TD_TX_DATA_MSK (0xff)
#define UART_TD_TX_DATA_OFST (0)

#define UART_FCR(base_addr) (UART00_TYPE (base_addr  + 0x14 ))
#define UART_FCR_RX_THR_MSK (0xd0)
#define UART_FCR_RX_THR_OFST (5)
#define UART_FCR_RX_THR_1 (0x00)
#define UART_FCR_RX_THR_2 (0x20)
#define UART_FCR_RX_THR_4 (0x40)
#define UART_FCR_RX_THR_6 (0x60)
#define UART_FCR_RX_THR_8 (0x80)
#define UART_FCR_RX_THR_10 (0xa0)
#define UART_FCR_RX_THR_12 (0xc0)
#define UART_FCR_RX_THR_14 (0xd0)
#define UART_FCR_TX_THR_MSK (0x1c)
#define UART_FCR_TX_THR_OFST (2)
#define UART_FCR_TX_THR_0 (0x00)
#define UART_FCR_TX_THR_2 (0x04)
```

```
#define UART_FCR_TX_THR_4 (0x08)
#define UART_FCR_TX_THR_8 (0x0c)
#define UART_FCR_TX_THR_10 (0x10)
#define UART_FCR_TX_THR_12 (0x14)
#define UART_FCR_TX_THR_14 (0x18)
#define UART_FCR_TX_THR_15 (0x1c)
#define UART_FCR_RC_MSK (0x02)
#define UART_FCR_RC_OFST (1)
#define UART_FCR_TC_MSK (0x01)
#define UART_FCR_TC_OFST (0)

#define UART_IES(base_addr) (UART00_TYPE (base_addr  + 0x18 ))
#define UART_IES_ME_MSK (0x8)
#define UART_IES_ME_OFST (3)
#define UART_IES_TIE_MSK (0x4)
#define UART_IES_TIE_OFST (2)
#define UART_IES_TE_MSK (0x2)
#define UART_IES_TE_OFST (1)
#define UART_IES_RE_MSK (0x1)
#define UART_IES_RE_OFST (0)

#define UART_IEC(base_addr) (UART00_TYPE (base_addr  + 0x1c ))
#define UART_IEC_ME_MSK (0x8)
#define UART_IEC_ME_OFST (3)
#define UART_IEC_TIE_MSK (0x4)
#define UART_IEC_TIE_OFST (2)
#define UART_IEC_TE_MSK (0x2)
#define UART_IEC_TE_OFST (1)
#define UART_IEC_RE_MSK (0x1)
#define UART_IEC_RE_OFST (0)

#define UART_ISR(base_addr) (UART00_TYPE (base_addr  + 0x20 ))
#define UART_ISR_MI_MSK (0x8)
#define UART_ISR_MI_OFST (3)
#define UART_ISR_TII_MSK (0x4)
#define UART_ISR_TII_OFST (2)
#define UART_ISR_TI_MSK (0x2)
#define UART_ISR_TI_OFST (1)
#define UART_ISR_RI_MSK (0x1)
#define UART_ISR_RI_OFST (0)

#define UART_IID(base_addr) (UART00_TYPE (base_addr  + 0x24 ))
#define UART_IID_IID_MSK (0x7)
#define UART_IID_IID_OFST (0)
#define UART_IID_IID_RI (1)
#define UART_IID_IID_TI (2)
#define UART_IID_IID_TII (3)
#define UART_IID_IID_MI (4)

#define UART_MC(base_addr) (UART00_TYPE (base_addr  + 0x28 ))
#define UART_MC_OE_MSK (0x40)
#define UART_MC_OE_OFST (6)
#define UART_MC_SP_MSK (0x20)
#define UART_MC_SP_OFST (5)
#define UART_MC_EP_MSK (0x10)
```

```
#define UART_MC_EP_OFST (4)
#define UART_MC_PE_MSK (0x08)
#define UART_MC_PE_OFST (3)
#define UART_MC_ST_MSK (0x04)
#define UART_MC_ST_ONE (0x0)
#define UART_MC_ST_TWO (0x04)
#define UART_MC_ST_OFST (2)
#define UART_MC_CLS_MSK (0x03)
#define UART_MC_CLS_OFST (0)
#define UART_MC_CLS_CHARLEN_5 (0)
#define UART_MC_CLS_CHARLEN_6 (1)
#define UART_MC_CLS_CHARLEN_7 (2)
#define UART_MC_CLS_CHARLEN_8 (3)

#define UART_MCR(base_addr) (UART00_TYPE (base_addr  + 0x2c ))
#define UART_MCR_AC_MSK (0x80)
#define UART_MCR_AC_OFST (7)
#define UART_MCR_AR_MSK (0x40)
#define UART_MCR_AR_OFST (6)
#define UART_MCR_BR_MSK (0x20)
#define UART_MCR_BR_OFST (5)
#define UART_MCR_LB_MSK (0x10)
#define UART_MCR_LB_OFST (4)
#define UART_MCR_DCD_MSK (0x08)
#define UART_MCR_DCD_OFST (3)
#define UART_MCR_RI_MSK (0x04)
#define UART_MCR_RI_OFST (2)
#define UART_MCR_DTR_MSK (0x02)
#define UART_MCR_DTR_OFST (1)
#define UART_MCR_RTS_MSK (0x01)
#define UART_MCR_RTS_OFST (0)

#define UART_MSR(base_addr) (UART00_TYPE (base_addr  + 0x30 ))
#define UART_MSR_DCD_MSK (0x80)
#define UART_MSR_DCD_OFST (7)
#define UART_MSR_RI_MSK (0x40)
#define UART_MSR_RI_OFST (6)
#define UART_MSR_DSR_MSK (0x20)
#define UART_MSR_DSR_OFST (5)
#define UART_MSR_CTS_MSK (0x10)
#define UART_MSR_CTS_OFST (4)
#define UART_MSR_DDCD_MSK (0x08)
#define UART_MSR_DDCD_OFST (3)
#define UART_MSR_TERI_MSK (0x04)
#define UART_MSR_TERI_OFST (2)
#define UART_MSR_DDSR_MSK (0x02)
#define UART_MSR_DDSR_OFST (1)
#define UART_MSR_DCTS_MSK (0x01)
#define UART_MSR_DCTS_OFST (0)

#define UART_DIV_LO(base_addr) (UART00_TYPE (base_addr  + 0x34 ))
#define UART_DIV_LO_DIV_MSK (0xff)
#define UART_DIV_LO_DIV_OFST (0)

#define UART_DIV_HI(base_addr) (UART00_TYPE (base_addr  + 0x38 ))
```

```
#define UART_DIV_HI_DIV_MSK (0xff)
#define UART_DIV_HI_DIV_OFST (0)

#endif /* __UART00_H */
```

uartcomm.h

```
/*
*   UARTComm.h
*
*       Function prototypes and definitions to use the UART IO
*       functions
*/

#ifndef UARTCOMM_H
#define UARTCOMM_H


#define BUFF_SIZE   16384
#define BUFF_MASK   0x3FFF
#define EOF (-1)

#define UART00_TYPE (volatile unsigned int*)

void uart_start_tx(void);
void uart_init(void);
void uart_irq_handler(void);

#endif /* UARTCOMM_H */
```

uartcomm.c

```
/*
*   General UART IO functions (interrupt driven), they are
*   configured by default for a baud rate of 38400, 8 bits
*   per character, 1 stop bit, no parity, with no flow control.
*
*/

#include <stdio.h>

#include "uartcomm.h"

#include "stripe.h"
#include "uart00.h"
#include "int_ctrl00.h"

#include "irq.h"

#define DIVISOR_FROM_BAUD(baud,clk) ((clk) /(16*(baud)))

char tx_buffer[BUFF_SIZE];
char rx_buffer[BUFF_SIZE];
volatile int tx_head,tx_tail,rx_head,rx_tail;
```

```
void uart_init(void)
{
    /* setup the rx and tx circular buffers */

    rx_head=rx_tail=0;
    tx_head=tx_tail=0;

    irq_init();

    /*
     * configure the uart for 38400 baud, 8 data,
     * 1 stop, no parity
     */

    *UART_MC(EXC_UART00_BASE) = UART_MC_CLS_CHARLEN_8;
    *UART_DIV_LO(EXC_UART00_BASE) =
                DIVISOR_FROM_BAUD(38400,EXC_AHB2_CLK_FREQUENCY)& 0xFF;
    *UART_DIV_HI(EXC_UART00_BASE) =
                (DIVISOR_FROM_BAUD(38400,EXC_AHB2_CLK_FREQUENCY)& 0xFF00) >> 8;

    /* Setup and clear FIFOs */
    *UART_FCR(EXC_UART00_BASE)=UART_FCR_RX_THR_1 | UART_FCR_TX_THR_2 |
        UART_FCR_RC_MSK | UART_FCR_TC_MSK;

    /* Clear pending interrupt */
    *UART_IEC(EXC_UART00_BASE) = UART_IEC_RE_MSK | UART_IEC_TE_MSK;

    /* Enable receive & transmit interrupts */
    *UART_IES(EXC_UART00_BASE)=UART_IES_RE_MSK;
}


static void uart_tx_handler(void)
{
    int dummy;
    /* Read the status register to clear the interrupt */
    dummy=*UART_TSR(EXC_UART00_BASE);

    /*
     * Write data to the fifo until it either
     * fills up, or we run out of stuff in the
     * tx buffer
     */


    while(((*UART_TSR(EXC_UART00_BASE) & UART_TSR_TX_LEVEL_MSK)<15)&&
          (tx_head!=tx_tail))
    {

        /* transmit the next character */
        *UART_TD(EXC_UART00_BASE)=tx_buffer[tx_head++];
        tx_head&=BUFF_MASK;
    }
    /*
```

```
     * If there's nothing left to transmit, turn the
     * interrupt off
     */
    if(tx_head==tx_tail)
    {
        *UART_IEC(EXC_UART00_BASE)=UART_IEC_TE_MSK;
    }
    else
    {
        *UART_IES(EXC_UART00_BASE) = UART_IES_TE_MSK;
    }

}

static void uart_rx_handler(void)
{
    int next_loc;

    /* Read the status register to clear the interrupt */
    while(*UART_RSR(EXC_UART00_BASE) & UART_RSR_RX_LEVEL_MSK)
    {
          next_loc=(rx_tail+1)&BUFF_MASK;
          if(next_loc==rx_head)
          {
              /*
               * Hmm, the buffer is full so we'll
               * ditch the stuff in the fifo
               */
              *UART_FCR(EXC_UART00_BASE)=UART_FCR_RC_MSK;
              break;
          }
        /* receive the next character */
        rx_buffer[rx_tail]=*UART_RD(EXC_UART00_BASE);
        rx_tail++;
        rx_tail&=BUFF_MASK;

    }
}

void uart_irq_handler(void)
{

    while(*UART_IID(EXC_UART00_BASE) & UART_IID_IID_MSK)
    {
        switch(*UART_IID(EXC_UART00_BASE) & UART_IID_IID_MSK)
        {
        case UART_IID_IID_RI:

            uart_rx_handler();
            break;

        case UART_IID_IID_TI:
            uart_tx_handler();
            break;
```

```
        case UART_IID_IID_TII:
        case UART_IID_IID_MI:
        default:

            /*
             * Tricky to know what to do here
             * so we'll do nothing and hope the
             * irq goes away. We'll probably just
             * get stuck in the while loop, but
             * there we go.
             */
            break;
        }
    }
}

void uart_start_tx(void)
{
    /*
     * if the tx interrupt is already running
     * then we need do nothing. Otherwise calling
     * the tx handler shoud kick things off
     */
    if(!(*UART_IES(EXC_UART00_BASE) & UART_IES_TE_MSK))
    {
        uart_tx_handler();
    }
}
```

main.c

```
/*
*   C Code for a simple application
*
*   This code prints a message using the embedded uart
*   as stdout. This software is programmed into flash using the
*   JTAG and flash interface.
*
*   It tests the UART by outputting information
*   over the UART at a baud rate of 38400 with 8 bits per
*   character no parity and one stop bit, with no flow control.
*/

#include <stdio.h>
#include "stripe.h"


int main(void)
{
    printf("\r\n");
    printf("**************************************************\r\n");
    printf("**                                              **\r\n");
    printf("** Hello DAMP Team!                             **\r\n");
    printf("**                                              **\r\n");
    printf("** DAMP is alive and kicking....                **\r\n");
```

```
    printf("**                                                  **\r\n");
    printf("****************************************************\r\n");

    while(1);

    return 0;
}
```

C

VGA file listings

vgacore.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity vgacore is
    port
    (
        reset: in std_logic;    -- reset
        clock: in std_logic;    -- VGA dot clock
        hsyncb: buffer std_logic;   -- horizontal (line) sync
        vsyncb: out std_logic;  -- vertical (frame) sync
        rgb: out std_logic_vector(7 downto 0)   -- red,green,blue colors
    );
end vgacore;

architecture vgacore_arch of vgacore is
signal hcnt: std_logic_vector(9 downto 0);  -- horizontal pixel counter
signal vcnt: std_logic_vector(9 downto 0);  -- vertical line counter
signal tel: std_logic_vector(5 downto 0);
signal red: std_logic_vector(2 downto 0);
signal green: std_logic_vector(2 downto 0);
signal blue: std_logic_vector(1 downto 0);
signal blank: std_logic;                    -- video blanking signal
signal pblank: std_logic;                   -- pipelined video blanking signal
begin

A: process(clock,reset)
begin
    -- reset asynchronously clears pixel counter
    if reset='1' then
        hcnt <= "0000000000";
    -- horiz. pixel counter increments on rising edge of dot clock
    elsif (clock'event and clock='1') then
        -- horiz. pixel counter rolls-over after 832 pixels
        if hcnt<831 then
            hcnt <= hcnt + 1;
        else
            hcnt <= "0000000000";
        end if;
    end if;
end process;

B: process(hsyncb,reset)
begin
    -- reset asynchronously clears line counter
```

```
    if reset='1' then
        vcnt <= "0000000000";
        red <= "000";
        green <= "000";
        blue <= "00";
        tel <= "000000";
    elsif (hsyncb'event and hsyncb='1') then
        if vcnt<519 then
            if vcnt(2 downto 0)="000" then
                if (tel>0 and tel<8) then -- 1
                    red <= red + 1;
                    green <= "000";
                    blue <= "00";
                elsif tel=8 then
                    red <= "000";
                    green <= "000";
                    blue <= "00";
                elsif (tel>8 and tel<16) then -- 2
                    red <= "000";
                    green <= green + 1;
                    blue <= "00";
                elsif tel=16 then -- 3
                    red <= "000";
                    green <= "000";
                    blue <= "00";
                elsif (tel>16 and tel<20) then -- 4
                    blue <= blue + 1;
                    red <= "000";
                    green <= "000";
                end if;
                if tel<20 then
                    tel <= tel + 1;
                else
                    tel <= "000000";
                    red <= "000";
                    green <= "000";
                    blue <= "00";
                end if;
            end if;
            vcnt <= vcnt + 1;
--          if tel="100010" then
--              tel <= "000000";
--          end if;
        else
            red <= "000";
            green <= "000";
            blue <= "00";
            vcnt <= "0000000000";
            tel <= "000000";
        end if;
    end if;
end process;

C: process(clock,reset)
begin
```

```
    -- reset asynchronously sets horizontal sync to inactive
    if reset='1' then
        hsyncb <= '0';
    -- horizontal sync is recomputed on the rising edge of every dot clock
    elsif (clock'event and clock='1') then
        -- horiz. sync is low in this interval to signal start of a new line
        if (hcnt>=667 and hcnt<707) then
            hsyncb <= '0';
        else
            hsyncb <= '1';
        end if;
    end if;
end process;

D: process(hsyncb,reset)
begin
    -- reset asynchronously sets vertical sync to inactive
    if reset='1' then
        vsyncb <= '0';
    -- vertical sync is recomputed at the end of every line of pixels
    elsif (hsyncb'event and hsyncb='1') then
        -- vert. sync is low in this interval to signal start of a new frame
        if (vcnt>=491 and vcnt<494) then
            vsyncb <= '0';
        else
            vsyncb <= '1';
        end if;
    end if;
end process;

E: blank <= '1' when (hcnt>639 or vcnt>479) else '0';
-- store the blanking signal for use in the next pipeline stage
F: process(clock,reset)
begin
    if reset='1' then
        pblank <= '0';
    elsif (clock'event and clock='1') then
        pblank <= blank;
    end if;
end process;

J: process(clock,reset)
begin
    -- blank the video on reset
    if reset='1' then
        rgb <= "00000000";
    -- update the color outputs on every dot clock
    elsif (clock'event and clock='1') then
        -- map the pixel to a color if the video is not blanked
        if pblank='0' then
            rgb <= red & green & blue;
        -- otherwise, output black if the video is blanked
        else
            rgb <= "00000000";  -- black
        end if;
```

```
    end if;
end process;

end vgacore_arch;
```

PS/2 file listings

D

PS2simpl.vhd

```
LIBRARY ieee;
  USE ieee.std_logic_1164.ALL;

-- ---------------------------------------------
    Entity PS2SIMPL is
-- ---------------------------------------------
      Port ( Clk      : In  std_logic;
             Reset    : In  std_logic;
             D7SEG_L  : Out std_logic_vector (1 to 7);
             D7SEG_H  : Out std_logic_vector (1 to 7);
             PS2_Data : In  std_logic;
             PS2_Clk  : In  std_logic;
             nLED     : Out std_logic );
end PS2SIMPL;

-- ---------------------------------------------
    Architecture SCHEMATIC of PS2SIMPL is
-- ---------------------------------------------
  component SEVENSEG
     Port (    Data : in  std_logic_vector (3 downto 0);
                Pol : in  std_logic;
              Segout : out std_logic_vector (1 to 7) );
  end component;
  component PS2_CTRL
  Generic ( FILTERSIZE : POSITIVE := 8 );
     Port (     Clk : in  std_logic;
             DoRead : in  std_logic;
            PS2_Clk : in  std_logic;
           PS2_Data : in  std_logic;
              Reset : in  std_logic;
          Scan_Code : out std_logic_vector (7 downto 0);
           Scan_DAV : out std_logic;
           Scan_Err : out std_logic );
  end component;

  signal Gnd,Vcc : std_logic;
  signal LED     : std_logic;
  signal DoRead  : std_logic;
  signal Code    : std_logic_vector (7 downto 0);

begin
  Gnd <= '0';  Vcc <= '1';

  PS2_CTRL_i : PS2_CTRL
    Generic Map ( FILTERSIZE => 8 )
```

```
    Port Map ( Clk=>Clk, Reset=>Reset, DoRead=>DoRead,
               PS2_Clk=>PS2_Clk, PS2_Data=>PS2_Data,
                Scan_Code=>Code, Scan_DAV=>DoRead, Scan_Err=>LED );

  -- Note: use Pol=>Gnd if display is active high type.
  Dec7SegL : SEVENSEG
     Port Map ( Data => Code(3 downto 0), Pol=>Vcc, Segout => D7SEG_L );

  Dec7SegH : SEVENSEG
     Port Map ( Data => Code(7 downto 4), Pol=>Vcc, Segout => D7SEG_H );

  nLED <= not LED; -- Note: remove the "not" if nLED is active high

end SCHEMATIC;
```

SevenSeg.vhd

```
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

-- --------------------------------
    Entity SevenSeg is
-- --------------------------------
  port ( Data   : in  std_logic_vector(3 downto 0);
         Pol    : in  std_logic;
         Segout : out std_logic_vector(1 to 7) ); -- a, b c d e f g
end;

-- -------------------------------------------------
    Architecture Comb of SevenSeg is
-- -------------------------------------------------

  signal Seg : std_logic_vector(SegOut'range);

begin

Xrg: for i in SegOut'range generate
  SegOut(i) <= Seg(i) xor Pol;
end generate;

process(Data)
begin
  case Data is
    when  x"0"  => Seg <= "1111110";
    when  x"1"  => Seg <= "0110000";
    when  x"2"  => Seg <= "1101101";
    when  x"3"  => Seg <= "1111001";
    when  x"4"  => Seg <= "0110011";
    when  x"5"  => Seg <= "1011011";
    when  x"6"  => Seg <= "1011111";
    when  x"7"  => Seg <= "1110000";
    when  x"8"  => Seg <= "1111111";
    when  x"9"  => Seg <= "1111011";
    when  x"A"  => Seg <= "1110111";
```

```
    when  x"B"  => Seg <= "0011111";
    when  x"C"  => Seg <= "0001101";
    when  x"D"  => Seg <= "0111101";
    when  x"E"  => Seg <= "1001111";
    when  x"F"  => Seg <= "1000111";
    when others => Seg <= (others => '-');
  end case;
end process;

end Comb;
```

PS2_Ctrl.vhd

```
library IEEE;
  use IEEE.Std_Logic_1164.all;
  use IEEE.Numeric_Std.all;

-- ---------------------------------------
    Entity PS2_Ctrl is
-- ---------------------------------------
  generic (FilterSize : positive := 8);
  port( Clk       : in  std_logic;  -- System Clock
        Reset     : in  std_logic;  -- System Reset
        PS2_Clk   : in  std_logic;  -- Keyboard Clock Line
        PS2_Data  : in  std_logic;  -- Keyboard Data Line
        DoRead    : in  std_logic;  -- From outside when reading the scan code
        Scan_Err  : out std_logic;  -- To outside : Parity or Overflow error
        Scan_DAV  : out std_logic;  -- To outside when a scan code has arrived
        Scan_Code : out std_logic_vector(7 downto 0) -- Eight bits Data Out
        );
end PS2_Ctrl;

-- ---------------------------------------
    Architecture Plain_Wrong of PS2_Ctrl is
-- ---------------------------------------
-- Comments : Bad solution !
-- 3 clock domains, no global reset, disguised FSM...
-- There is also a bad problem with the synchronous reset
-- which is not on the right clock domain !
-- Note : the VHDL style has been fixed...

  signal Bit_Cnt    : unsigned (3 downto 0);
  signal Shift_Reg  : std_logic_vector(8 downto 0);
  signal Read_CHAR  : std_logic;
  signal Ready_set  : std_logic;
  signal PS2_Clk_f  : std_logic;
  signal Filter     : std_logic_vector(FilterSize-1 downto 0);

begin

Scan_Err <= '0';  -- not used in this architecture

  -- Clock domain #1 , async reset #1

  process (DoRead, Ready_set)
```

```
  begin
    if DoRead = '1' then
      Scan_DAV <= '0';
    elsif rising_edge(Ready_set) then
      Scan_DAV <= '1';
    end if;
  end process;


  -- Clock domain #2, no reset !

-- This process filters the raw clock signal coming from the keyboard
-- using a shift register and two 4-inputs AND gates
-- Implies a 320 ns delay
  Clock_filter : process (Clk)
  begin
    if rising_edge (Clk) then
      Filter <= PS2_Clk & Filter(Filter'high downto 1);
      if Filter = "11111111" then
        PS2_Clk_f <= '1';
      elsif Filter = "00000000" then
        PS2_Clk_f <= '0';
      end if;
    end if;
  end process Clock_filter;

  -- Clock domain #3, partial synchronous reset !
  -- PS2_Data is not resynchronized

  --This process Reads in serial data coming from the keyboard
  process(PS2_Clk_f)
  begin
    if rising_edge (PS2_Clk_f) then
      if RESET = '1' then
        Bit_Cnt  <= (others => '0');
        Read_CHAR <= '0';
      else
        if PS2_Data = '0' and Read_CHAR = '0' then
          Read_CHAR <= '1';
          Ready_set <= '0';
        else
          -- Shift in next 8 data bits to assemble a scan code
          if Read_CHAR = '1' then
            if Bit_Cnt < 9 then
              Bit_Cnt  <= Bit_Cnt + 1;
              Shift_Reg <= PS2_Data & Shift_Reg(8 downto 1);
              Ready_set <= '0';
            else -- End of scan code character, so set flags and exit loop
              Scan_Code <= Shift_Reg (7 downto 0);
              Read_CHAR <= '0';
              Ready_set <= '1';
              Bit_Cnt     <= (others => '0');
            end if;
          end if;
        end if;
```

```
      end if;
    end if;
  end process;

end Plain_Wrong;


-- --------------------------------------
    Architecture ALSE_RTL of PS2_Ctrl is
-- --------------------------------------
-- Fully synchronous solution, same Filter on PS2_Clk.
-- Still as compact as "Plain_wrong"...
-- Possible improvement : add TIMEOUT on PS2_Clk while shifting
-- Note: PS2_Data is resynchronized though this should not be
-- necessary (qualified by Fall_Clk and does not change at that time).
-- Note the tricks to correctly interpret 'H' as '1' in RTL simulation.

  signal PS2_Datr  : std_logic;

  subtype Filter_t is std_logic_vector(FilterSize-1 downto 0);
  signal Filter    : Filter_t;
  signal Fall_Clk  : std_logic;
  signal Bit_Cnt   : unsigned (3 downto 0);
  signal Parity    : std_logic;
  signal Scan_DAVi : std_logic;

  signal S_Reg     : std_logic_vector(8 downto 0);

  signal PS2_Clk_f : std_logic;

  Type   State_t is (Idle, Shifting);
  signal State : State_t;

begin

Scan_DAV <= Scan_DAVi;

-- This filters digitally the raw clock signal coming from the keyboard :
--  * Eight consecutive PS2_Clk=1 makes the filtered_clock go high
--  * Eight consecutive PS2_Clk=0 makes the filtered_clock go low
-- Implies a (FilterSize+1) x Tsys_clock delay on Fall_Clk wrt Data
-- Also in charge of the re-synchronization of PS2_Data

process (Clk,Reset)
begin
  if Reset='1' then
    PS2_Datr  <= '0';
    PS2_Clk_f <= '0';
    Filter    <= (others=>'0');
    Fall_Clk  <= '0';
  elsif rising_edge (Clk) then
    PS2_Datr <= PS2_Data and PS2_Data; -- also turns 'H' into '1'
    Fall_Clk <= '0';
    Filter   <= (PS2_Clk and PS2_CLK) & Filter(Filter'high downto 1);
    if Filter = Filter_t'(others=>'1') then
```

```
      PS2_Clk_f <= '1';
    elsif Filter = Filter_t'(others=>'0') then
      PS2_Clk_f <= '0';
      if PS2_Clk_f = '1' then
        Fall_Clk <= '1';
      end if;
    end if;
  end if;
end process;


-- This simple State Machine reads in the Serial Data
-- coming from the PS/2 peripheral.

process(Clk,Reset)
begin

  if Reset='1' then
    State     <= Idle;
    Bit_Cnt   <= (others => '0');
    S_Reg     <= (others => '0');
    Scan_Code <= (others => '0');
    Parity    <= '0';
    Scan_Davi <= '0';
    Scan_Err  <= '0';

  elsif rising_edge (Clk) then

    if DoRead='1' then
      Scan_Davi <= '0'; -- note: this assgnmnt can be overriden
    end if;

    case State is

      when Idle =>
        Parity  <= '0';
        Bit_Cnt <= (others => '0');
        -- note that we dont need to clear the Shift Register
        if Fall_Clk='1' and PS2_Datr='0' then -- Start bit
          Scan_Err <= '0';
          State <= Shifting;
        end if;

      when Shifting =>
          if Bit_Cnt >= 9 then
            if Fall_Clk='1' then -- Stop Bit
              -- Error is (wrong Parity) or (Stop='0') or Overflow
              Scan_Err  <= (not Parity) or (not PS2_Datr) or Scan_DAVi;
              Scan_Davi <= '1';
              Scan_Code <= S_Reg(7 downto 0);
              State <= Idle;
            end if;
          elsif Fall_Clk='1' then
            Bit_Cnt  <= Bit_Cnt + 1;
            S_Reg <= PS2_Datr & S_Reg (S_Reg'high downto 1); -- Shift right
```

```
            Parity <= Parity xor PS2_Datr;
          end if;

      when others => -- never reached
        State <= Idle;

    end case;

    --Scan_Err <= '0'; -- to create an on-purpose error on Scan_Err !

  end if;

end process;

end ALSE_RTL;
```

SDRAM file listings

E

armc_startup.s

```
    AREA init, CODE, READONLY
    b   Start

Start

    ldr     r0, =0x00008000 ; Address 0x0 of DPRAM
    ldr     r1, =0x0000C539 ; Fills the 7-segments with the end mark

    str     r1, [r0]

    ldr     r2, =0x20000000 ; Base address of the SDRAM
    ldr     r3, =0x22000000 ; End address of the SDRAM

Loop1
    str     r2, [r2]        ; Filling the SDRAM with addresses
    add     r2, r2, #4
    cmp     r2, r3
    bne     Loop1

    ldr     r2, =0x20000000

Loop2
    ldr     r4, [r2]        ; Checking the contents of the SDRAM
    cmp     r2, r4
    bne     Error

    add     r2, r2, #4
    cmp     r2, r3
    bne     Loop2

    ldr     r1, =0x000039C5 ; Fills the 7-segments with the end mark
    str     r1, [r0]

    b       TheEnd

Error
    ldr     r1, =0x000061F5 ; Fills the 7-segment displays with an Error mark
    str     r1, [r0]

TheEnd
    b   TheEnd
    nop

    END
```

VGA Slideshow file listings

F

video_system.vhd

```
-- video_system.vhd
--
-- AN287 top-level design file

LIBRARY ieee,work;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE work.video_components.ALL;

ENTITY video_system IS
    PORT
    (
        -- clock and reset signals
        clk_ref     : IN    STD_LOGIC;
        sys_clk     : IN    STD_LOGIC;
        clock25     : IN    STD_LOGIC;
        npor        : IN    STD_LOGIC;
        nreset      : INOUT STD_LOGIC;
        intextpin   : IN    STD_LOGIC;
        --reset_n       : IN    STD_LOGIC;  -- use SW4 on XA1 board

        -- EBI signals
        ebiack      : IN    STD_LOGIC;
        ebidq       : INOUT STD_LOGIC_VECTOR(15 downto 0);
        ebiclk      : OUT   STD_LOGIC;
        ebiwen      : OUT   STD_LOGIC;
        ebioen      : OUT   STD_LOGIC;
        ebiaddr     : OUT   STD_LOGIC_VECTOR(24 downto 0);
        ebibe       : OUT   STD_LOGIC_VECTOR(1 downto 0);
        ebicsn      : OUT   STD_LOGIC_VECTOR(3 downto 0);

        -- SDRAM signals
        sdramdq     : INOUT STD_LOGIC_VECTOR(15 downto 0);
        sdramdqs    : INOUT STD_LOGIC_VECTOR(1 downto 0);
        sdramclk    : OUT   STD_LOGIC;
        sdramclkn   : OUT STD_LOGIC;
        sdramclke   : OUT   STD_LOGIC;
        sdramwen    : OUT   STD_LOGIC;
        sdramcasn   : OUT   STD_LOGIC;
        sdramrasn   : OUT   STD_LOGIC;
        sdramaddr   : OUT   STD_LOGIC_VECTOR(14 downto 0);
        sdramcsn    : OUT   STD_LOGIC_VECTOR(1 downto 0);
        sdramdqm    : OUT   STD_LOGIC_VECTOR(1 downto 0);

        -- UART signals
```

```
        uartrxd     : IN    STD_LOGIC;
        uartdsrn    : IN    STD_LOGIC;
        uartctsn    : IN    STD_LOGIC;
        uartrin     : INOUT STD_LOGIC;
        uartdcdn    : INOUT STD_LOGIC;
        uarttxd     : OUT   STD_LOGIC;
        uartrtsn    : OUT   STD_LOGIC;
        uartdtrn    : OUT   STD_LOGIC;

        -- VGA outputs
        vsync       : OUT std_logic;
        hsync       : OUT std_logic;
        R       : OUT std_logic_vector(2 downto 0);
        G       : OUT std_logic_vector(2 downto 0);
        B       : OUT std_logic_vector(1 downto 0)
    );
END video_system;

ARCHITECTURE rtl OF video_system IS

    SIGNAL reset_n      : std_logic;
    SIGNAL vcc          : std_logic;
    SIGNAL gnd          : std_logic;

    SIGNAL hclk         : std_logic;
    SIGNAL clock50          : std_logic;

    -- PLD-to-Stripe signals
    SIGNAL dma_masterhready     : std_logic;
    SIGNAL dma_masterhbusreq    : std_logic;
    SIGNAL dma_masterhlock      : std_logic;
    SIGNAL dma_masterhresp      : std_logic_vector(1 downto 0);
    SIGNAL dma_masterhrdata     : std_logic_vector(31 downto 0);
    SIGNAL dma_masterhaddr      : std_logic_vector(31 downto 0);
    SIGNAL dma_masterhsize      : std_logic_vector(1 downto 0);
    SIGNAL dma_masterhtrans     : std_logic_vector(1 downto 0);
    SIGNAL dma_masterhburst     : std_logic_vector(2 downto 0);
    SIGNAL dma_masterhwdata     : std_logic_vector(31 downto 0);

    -- Stripe-to-PLD bridge signals
    SIGNAL stripe_hrdata        : std_logic_vector(31 downto 0);
    SIGNAL stripe_hresp         : std_logic_vector(1 downto 0);
    SIGNAL stripe_hwrite        : std_logic;
    SIGNAL stripe_hlock         : std_logic;
    SIGNAL stripe_hbusreq       : std_logic;
    SIGNAL stripe_haddr         : std_logic_vector(31 downto 0);
    SIGNAL stripe_hburst        : std_logic_vector(2 downto 0);
    SIGNAL stripe_hsize         : std_logic_vector(1 downto 0);
    SIGNAL stripe_htrans        : std_logic_vector(1 downto 0);
    SIGNAL stripe_hwdata        : std_logic_vector(31 downto 0);
    SIGNAL stripe_hready        : std_logic;

    -- slave iface signals to and from decoder and mux
    SIGNAL hready_slave_iface   : std_logic;
    SIGNAL hsel_slave_iface     : std_logic;
```

```
    SIGNAL hresp_slave_iface    : std_logic_vector(1 downto 0);
    SIGNAL hrdata_slave_iface   : std_logic_vector(31 downto 0);

    -- default slave signals to and from decoder and mux
    SIGNAL hready_default_slave : std_logic;
    SIGNAL hsel_default_slave   : std_logic;
    SIGNAL hresp_default_slave  : std_logic_vector(1 downto 0);

    SIGNAL irq                  : std_logic;
    SIGNAL irq_to_processor     : std_logic_vector(5 downto 0);

BEGIN
    vcc <= '1';
    gnd <= '0';
    reset_n <= '1';
    irq_to_processor <= "0000" & irq & '0';

    clockgen: system_pll
    PORT MAP
    (
        inclock => sys_clk,
        clock0 => hclk,
        clock1 => clock50
    );

    the_stripe : stripe
    PORT MAP
    (
        clk_ref => clk_ref,
        npor    => npor,
        nreset  => nreset,
        intextpin => intextpin,
        uartrxd => uartrxd,
        uartdsrn => uartdsrn,
        uartctsn => uartctsn,
        uartrin => uartrin,
        uartdcdn => uartdcdn,
        uarttxd => uarttxd,
        uartrtsn => uartrtsn,
        uartdtrn => uartdtrn,
        ebiack  => ebiack,
        ebidq   => ebidq,
        ebiclk  => ebiclk,
        ebiwen  => ebiwen,
        ebioen  => ebioen,
        ebiaddr => ebiaddr,
        ebibe   => ebibe,
        ebicsn  => ebicsn,
        sdramdq => sdramdq,
        sdramdqs => sdramdqs,
        sdramclk => sdramclk,
        sdramclkn => sdramclkn,
        sdramclke => sdramclke,
        sdramwen => sdramwen,
        sdramcasn => sdramcasn,
```

```
    sdramrasn => sdramrasn,
    sdramaddr => sdramaddr,
    sdramcsn => sdramcsn,
    sdramdqm => sdramdqm,
    slavehclk => hclk,
    slavehwrite => gnd,
    slavehreadyi => dma_masterhready,
    slavehselreg => gnd,
    slavehsel => vcc,
    slavehmastlock => dma_masterhlock,
    slavehaddr => dma_masterhaddr,
    slavehtrans => dma_masterhtrans,
    slavehsize => dma_masterhsize,
    slavehburst => dma_masterhburst,
    slavehwdata => dma_masterhwdata,
    slavehreadyo => dma_masterhready,
    slavebuserrint => gnd,
    slavehresp => dma_masterhresp,
    slavehrdata => dma_masterhrdata,
    masterhclk => clock50,
    masterhready => vcc,
    masterhgrant => vcc,
    masterhrdata => stripe_hrdata,
    masterhresp => stripe_hresp,
    masterhwrite => stripe_hwrite,
    masterhlock => stripe_hlock,
    masterhbusreq => stripe_hbusreq,
    masterhaddr => stripe_haddr,
    masterhburst => stripe_hburst,
    masterhsize => stripe_hsize,
    masterhtrans => stripe_htrans,
    masterhwdata => stripe_hwdata,
    intpld => irq_to_processor
);

the_slave_decoder: slave_decoder
PORT MAP
(
    hbusreq         => stripe_hbusreq,
    haddr           => stripe_haddr,
    hsel_slave_iface    => hsel_slave_iface,
    hsel_default_slave  => hsel_default_slave
);

the_default_slave: default_slave
PORT MAP
(
    hclock      => clock50,
    hresetn     => reset_n,
    hsel        => hsel_default_slave,
    htrans      => stripe_htrans,
    hready      => hready_default_slave,
    hresp       => hresp_default_slave,
    hrdata      => open
);
```

```
    the_response_and_data_mux :response_and_data_mux
    PORT MAP
    (
        hclock          => clock50,
        hresetn         => reset_n,
        hsel_slave_iface    => hsel_slave_iface,
        hready_slave_iface  => hready_slave_iface,
        hresp_slave_iface   => hresp_slave_iface,
        hrdata_slave_iface  => hrdata_slave_iface,
        hsel_default_slave  => hsel_default_slave,
        hready_default_slave    => hready_default_slave,
        hresp_default_slave => hresp_default_slave,
        hready          => stripe_hready,
        hresp           => stripe_hresp,
        hrdata          => stripe_hrdata
    );

    the_video_dma: video_dma
    PORT MAP
    (
        reset_n             => reset_n,
        hclock              => hclk,
        clock25             => clock25,
        clock50             => clock50,
        siface_hsel                 => hsel_slave_iface,
        siface_hwrite_from_stripe   => stripe_hwrite,
        siface_hbusreq_from_stripe  => stripe_hbusreq,
        siface_htrans_from_stripe   => stripe_htrans,
        siface_hsize_from_stripe    => stripe_hsize,
        siface_hburst_from_stripe   => stripe_hburst,
        siface_haddress_from_stripe => stripe_haddr,
        siface_hwdata_from_stripe   => stripe_hwdata,
        siface_hready_to_stripe     => hready_slave_iface,
        siface_hresp_to_stripe      => hresp_slave_iface,
        siface_hrdata_to_stripe     => hrdata_slave_iface,
        miface_hready_from_stripe   => dma_masterhready,
        miface_hgrant_from_stripe   => vcc,
        irq                         => irq,
        miface_hlock_to_stripe      => dma_masterhlock,
        miface_hbusreq_to_stripe    => dma_masterhbusreq,
        miface_htrans_to_stripe     => dma_masterhtrans,
        miface_hsize_to_stripe      => dma_masterhsize,
        miface_hburst_to_stripe     => dma_masterhburst,
        miface_hwdata_to_stripe     => dma_masterhwdata,
        miface_hrdata_from_stripe   => dma_masterhrdata,
        miface_haddr_to_stripe      => dma_masterhaddr,
        vsync               => vsync,
        hsync               => hsync,
        R               => R,
        G               => G,
        B               => B
    );
END rtl;
```

video_dma.vhd

```
LIBRARY ieee,work;
USE ieee.std_logic_1164.ALL;
USE work.video_components.ALL;

ENTITY video_dma IS
    PORT
    (
        -- System signals
        reset_n             : IN std_logic;
        hclock              : IN std_logic;
             -- clock to the AHM master interace
        clock25             : IN std_logic;
             -- clock to the VGA driver
        clock50             : IN std_logic;
             -- clock to the slave interface and VGA driver

        --Slave Interface signals
        siface_hsel         : IN std_logic;
        siface_hwrite_from_stripe   : IN std_logic;
        siface_hbusreq_from_stripe  : IN std_logic;
        siface_htrans_from_stripe   : IN std_logic_vector(1 downto 0);
        siface_hsize_from_stripe    : IN std_logic_vector(1 downto 0);
        siface_hburst_from_stripe   : IN std_logic_vector(2 downto 0);
        siface_haddress_from_stripe : IN std_logic_vector(31 downto 0);
        siface_hwdata_from_stripe   : IN std_logic_vector(31 downto 0);
        siface_hready_to_stripe     : OUT std_logic;
        siface_hresp_to_stripe      : OUT std_logic_vector(1 downto 0);
        siface_hrdata_to_stripe     : OUT std_logic_vector(31 downto 0);

        -- DMA Controller Master Interface signals
        miface_hready_from_stripe   : IN std_logic;
        miface_hgrant_from_stripe   : IN std_logic;
        irq                         : OUT std_logic;
        miface_hlock_to_stripe      : OUT std_logic;
        miface_hbusreq_to_stripe    : OUT std_logic;
        miface_htrans_to_stripe     : OUT std_logic_vector(1 downto 0);
        miface_hsize_to_stripe      : OUT std_logic_vector(1 downto 0);
        miface_hburst_to_stripe     : OUT std_logic_vector(2 downto 0);
        miface_hwdata_to_stripe     : OUT std_logic_vector(31 downto 0);
        miface_hrdata_from_stripe   : IN std_logic_vector(31 downto 0);
        miface_haddr_to_stripe      : OUT std_logic_vector(31 downto 0);

        -- VGA outputs
        vsync                       : OUT std_logic;
        hsync                       : OUT std_logic;
        R                           : OUT std_logic_vector(2 downto 0);
        G                           : OUT std_logic_vector(2 downto 0);
        B                           : OUT std_logic_vector(1 downto 0)
    );
END video_dma;

ARCHITECTURE rtl OF video_dma IS

    SIGNAL vcc          : std_logic;
```

```
    -- connection signals between the slave interface and the control logic
    -- of the DMA and the VGA driver
    SIGNAL buffer_address       : std_logic_vector(31 downto 0);
    SIGNAL image_dimensions     : std_logic_vector(31 downto 0);
    SIGNAL dma_status           : std_logic_vector(31 downto 0);
    SIGNAL dma_control_register : std_logic_vector(31 downto 0);
    SIGNAL num_words_per_line   : std_logic_vector(15 downto 0);
    SIGNAL num_pixels_per_line  : std_logic_vector(9 downto 0);
    SIGNAL num_lines            : std_logic_vector(15 downto 0);
    SIGNAL enable_video         : std_logic;

    -- signals from DMA controller to the image/line buffer
    SIGNAL set_irq              : std_logic;
    SIGNAL buffer_select        : std_logic;
    SIGNAL dma_data_valid       : std_logic;
    SIGNAL dma_data         : std_logic_vector(31 downto 0);
    SIGNAL rx_buffer_wraddress  : std_logic_vector(9 downto 0);
    SIGNAL rx_buffer_rdaddress  : std_logic_vector(31 downto 0);

    -- VGA driver interal signals
    SIGNAL vblank           : std_logic;
    SIGNAL hblank           : std_logic;
    SIGNAL video_data       : std_logic_vector(31 downto 0);

    SIGNAL miface_haddr_to_stripe_sig : std_logic_vector(31 downto 0);

    CONSTANT siface_base_address : std_logic_vector(31 downto 0) := X"80000000";

BEGIN

vcc <= '1';
miface_haddr_to_stripe <= miface_haddr_to_stripe_sig;

-- Set the IRQ signal when we have transmitted the last pixel of a frame.
-- The IRQ signal is deasserted at the start of the next frame or if
-- the processor performs a valid read or write transaction to the
-- slave interface.
irq_control: PROCESS(hclock,reset_n)
BEGIN
    IF reset_n = '0' THEN
        irq <= '0';
    ELSIF rising_edge(hclock) THEN
        IF set_irq = '1' THEN
            irq <= '1';
        ELSIF siface_hsel = '1' AND siface_haddress_from_stripe = X"80000000"
                                AND siface_hwrite_from_stripe = '1' THEN
            irq <= '0';
        END IF;
    END IF;
END PROCESS;

controller: video_dma_controller
PORT MAP
(
```

```
    buffer_address      => buffer_address,
    num_words_per_line  => num_words_per_line,
    num_lines           => num_lines,
    enable_dma          => enable_video,
    vblank              => vblank,
    hblank              => hblank,
    reset_n             => reset_n,
    masterhclk          => hclock,
    masterhgrant        => miface_hgrant_from_stripe,
    masterhlock         => miface_hlock_to_stripe,
    masterhready        => miface_hready_from_stripe,
    masterhrdata        => miface_hrdata_from_stripe,
    masterhaddr         => miface_haddr_to_stripe_sig,
    masterhbusreq       => miface_hbusreq_to_stripe,
    masterhsize         => miface_hsize_to_stripe,
    masterhtrans        => miface_htrans_to_stripe,
    masterhburst        => miface_hburst_to_stripe,
    masterhwdata        => miface_hwdata_to_stripe,
    set_irq             => set_irq,
    buffer_select       => buffer_select,
    dma_data            => dma_data,
    rx_buffer_wraddress => rx_buffer_wraddress,
    dma_data_valid      => dma_data_valid
);

slave_side: slave_interface
PORT MAP
(
    hresetn             => reset_n,
    hclock              => clock50,
    hwrite              => siface_hwrite_from_stripe,
    hsel                => siface_hsel,
    htrans              => siface_htrans_from_stripe,
    hsize               => siface_hsize_from_stripe,
    hburst              => siface_hburst_from_stripe,
    haddress            => siface_haddress_from_stripe,
    hwdata              => siface_hwdata_from_stripe,
    hready              => siface_hready_to_stripe,
    hresp               => siface_hresp_to_stripe,
    hrdata              => siface_hrdata_to_stripe,
    status              => dma_status,
    current_address     => miface_haddr_to_stripe_sig,
    buffer_address      => buffer_address,
    image_dimensions    => image_dimensions,
    control             => dma_control_register
);

-- distribute the control and status signals between the slave interface
-- and the dma controller and vga driver
num_words_per_line <= "00" & image_dimensions(31 downto 18);
num_pixels_per_line <= image_dimensions(25 downto 16);
num_lines <= image_dimensions(15 downto 0);

dma_status(0) <= buffer_select;
dma_status(1) <= hblank;
```

```
dma_status(2) <= vblank;
dma_status(31 downto 3) <= (others => '0');

enable_video <= dma_control_register(0);

video_line_buffer: line_buffer
PORT MAP
(
    data        => dma_data,
    wraddress   => rx_buffer_wraddress(8 downto 0),
    rdaddress   => rx_buffer_rdaddress(10 downto 2),
    wren        => dma_data_valid,
    rden        => vcc,
    wrclock     => hclock,  -- must be same clock as master interface
    rdclock     => clock25, -- same clock as core VGA driver clock
    q           => video_data
);

vga640x480: vga_driver
PORT MAP
(
    reset_n             => reset_n,
    clock25             => clock25,  -- vga driver core clock
    enable              => enable_video,
    num_lines           => num_lines(9 downto 0),
    num_pixels_per_line => num_pixels_per_line,
    video_address       => rx_buffer_rdaddress,
    video_data          => video_data,
    vsync               => vsync,
    hsync               => hsync,
    vblank              => vblank,
    hblank              => hblank,
    R                   => R,
    G                   => G,
    B                   => B
);
END rtl;
```

video_dma_controller.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY video_dma_controller IS
    PORT (

    -- inputs from AHB slave side interface
    buffer_address  : IN std_logic_vector(31 downto 0);
    num_words_per_line  : IN std_logic_vector(15 downto 0);
    num_lines       : IN std_logic_vector(15 downto 0);
    enable_dma      : IN std_logic;

    -- inputs from VGA driver
    vblank          : IN std_logic;
    hblank          : IN std_logic;
```

```
    -- AHB master interface signals
    -- These signals connect to the PLD-to-stripe bridge
    reset_n         : IN std_logic;
    masterhclk      : IN std_logic;
    masterhgrant        : IN std_logic;
    masterhready        : IN std_logic;
    masterhrdata        : IN std_logic_vector(31 downto 0);
    masterhlock     : OUT std_logic;
    masterhaddr     : OUT std_logic_vector(31 downto 0);
    masterhbusreq       : OUT std_logic;
    masterhsize     : OUT std_logic_vector(1 downto 0);
    masterhtrans        : OUT std_logic_vector(1 downto 0);
    masterhburst        : OUT std_logic_vector(2 downto 0);
    masterhwdata        : OUT std_logic_vector(31 downto 0);

    -- data and control signals for output of DMA
    set_irq             : OUT std_logic;
    buffer_select       : OUT std_logic;
    dma_data        : OUT std_logic_vector(31 downto 0);
    rx_buffer_wraddress : OUT std_logic_vector(9 downto 0);
    dma_data_valid      : OUT std_logic
    );
END video_dma_controller;

ARCHITECTURE rtl OF video_dma_controller IS
    SIGNAL masterhaddr_sig      : std_logic_vector(31 downto 0);
    SIGNAL masterhbusreq_sig    : std_logic;
    SIGNAL rx_buffer_wraddress_sig  : std_logic_vector(9 downto 0);

    -- Declare signals for AHB Transactor FSM
    TYPE ahb_state_type IS (idle,seq,nonseq);
    SIGNAL ahb_state : ahb_state_type;

    -- Declare signals for DMA control FSM
    TYPE control_state_type IS (idle, begin_line, transmit_line, end_line);
    SIGNAL control_state : control_state_type;

    SIGNAL start_transfer           : std_logic;
    SIGNAL end_transfer         : std_logic;

    SIGNAL buffered_hrdata          : std_logic_vector(31 downto 0);
    SIGNAL load_address         : std_logic;
    SIGNAL incr_masterhaddr_sig     : std_logic;
    SIGNAL incr_burst_beat_counter      : std_logic;

    SIGNAL burst_counter            : std_logic_vector(5 downto 0);
    SIGNAL incr_burst_counter       : std_logic;

    SIGNAL burst_beat_counter       : std_logic_vector(2 downto 0);

    SIGNAL current_line_address     : std_logic_vector(31 downto 0);
    SIGNAL line_counter         : std_logic_vector(9 downto 0);
    SIGNAL reset_line_counter       : std_logic;
```

```
    SIGNAL dma_vblank           : std_logic;
    SIGNAL dma_hblank           : std_logic;
    SIGNAL delay_dma_hblank         : std_logic;
    SIGNAL start_of_line            : std_logic;


BEGIN

masterhaddr <= masterhaddr_sig;
masterhbusreq <= masterhbusreq_sig;
rx_buffer_wraddress <= rx_buffer_wraddress_sig;

-- Register the hrdata signal to help ensure that we can run the dma
-- at fast clock frequencies.
register_hrdata: PROCESS(masterhclk, reset_n)
BEGIN
    IF reset_n = '0' THEN
        buffered_hrdata <= (others => '0');
    ELSIF rising_edge(masterhclk) THEN
        buffered_hrdata <= masterhrdata;
    END IF;
END PROCESS;

-- hsize is always 32 bits
masterhsize <= "10";
-- we always do unspecified length bursts to get maximum bandwidth
-- out of the PLD-to-stripe bridge
masterhburst <= "001";

-- The VGA controller is on a 25MHZ clock domain so we need to transfer
-- the controller signals from the 25MHz domain to the 100MHz master
-- domain. Ideally the 100MHz and 25MHz clocks will be synchronous and
-- we will not see metastability issues.
register_vga_signals: PROCESS(masterhclk,reset_n)
BEGIN
    IF reset_n = '0' THEN
        dma_vblank <= '0';
        dma_hblank <= '0';
        start_of_line <= '0';
        delay_dma_hblank <= '0';
    ELSIF rising_edge(masterhclk) THEN
        dma_vblank <= vblank;
        dma_hblank <= hblank;
        delay_dma_hblank <= dma_hblank;
        -- This segment of code generates a start of line signal which
        -- is active for a single clock cycle immedietly following the
        -- deassertion of the hblank signal.
        IF delay_dma_hblank = '1' AND dma_hblank = '0' AND dma_vblank = '1' THEN
            start_of_line <= '1';
        ELSE
            start_of_line <= '0';
        END IF;
    END IF;
END PROCESS;
```

```
-- CONTROL PROCESS
control_FSM: PROCESS(masterhclk,reset_n)
BEGIN
    IF reset_n = '0' THEN
        control_state <= idle;
        start_transfer <= '0';      -- begin transferring a line from SDRAM
        end_transfer <= '0';        -- end line transfer
        incr_burst_counter <= '0';  -- enable_dma signal for burst counter
        reset_line_counter <= '0';
        masterhbusreq_sig <= '0';   -- request access to the stripe AHB buses
        masterhlock <= '0';
        set_irq <= '0';
    ELSIF rising_edge(masterhclk) THEN
        CASE control_state IS
            -- We are in the idle state in between horizontal line transfers.
            -- When in this state we do not request access to the stripe.
            WHEN idle =>
                IF start_of_line = '1' AND enable_dma = '1' THEN
                    masterhbusreq_sig <= '1';
                    control_state <= begin_line;
                ELSE
                    masterhbusreq_sig <= '0';
                    control_state <= idle;
                END IF;
                incr_burst_counter <= '0';
                start_transfer <= '0';
                end_transfer <= '0';
                reset_line_counter <= '0';
                masterhlock <= '0';
                set_irq <= '0';

            -- Once the blanking signals are detected we enter this state
            -- which initiates an undefined length burst read
            WHEN begin_line =>
                control_state <= transmit_line;
                start_transfer <= '1';
                incr_burst_counter <= '0';
                end_transfer <= '0';
                reset_line_counter <= '0';
                masterhbusreq_sig <= '1';
                masterhlock <= '1';
                set_irq <= '0';

            -- During this state we are performing an undefined length
            -- burst read from the stripe.
            WHEN transmit_line =>
                -- Count a burst every time the burst beat counter
                -- has a value of 1.
                IF burst_beat_counter = 1 AND
                                    incr_burst_beat_counter = '1' THEN
                    incr_burst_counter <= '1';
                ELSE
                    incr_burst_counter <= '0';
                END IF;
```

```
                -- We will transition to the end line state if the
                -- burst_counter is equal to the number of words per
                -- line divided by 8.  We divivde by 8 because the
                -- burst counter counts 8 beats at a time.
                IF burst_counter = num_words_per_line(8 downto 3)
                                        AND burst_beat_counter = 5 THEN
                    end_transfer <= '1';
                    masterhbusreq_sig <= '0';
                    control_state <= end_line;
                ELSE
                    end_transfer <= '0';
                    masterhbusreq_sig <= '1';
                    control_state <= transmit_line;
                END IF;
                start_transfer <= '0';
                reset_line_counter <= '0';
                masterhlock <= '1';
                set_irq <= '0';

            WHEN end_line =>
                control_state <= idle;
                incr_burst_counter <= '0';
                start_transfer <= '0';
                end_transfer <= '0';
                masterhbusreq_sig <= '0';
                IF line_counter = num_lines THEN
                    reset_line_counter <= '1';
                    set_irq <= '1';
                ELSE
                    reset_line_counter <= '0';
                    set_irq <= '0';
                END IF;
                masterhlock <= '0';

            WHEN others =>
                null;
        END CASE;
    END IF;
END PROCESS;

-- keeps track of which line is being displayed on the screen
line_count: PROCESS(masterhclk,reset_n)
BEGIN
    IF reset_n = '0' THEN
        line_counter <= (others => '0');
    ELSIF rising_edge(masterhclk) THEN
        IF reset_line_counter = '1' THEN
            line_counter <= (others => '0');
        ELSIF start_transfer = '1' THEN
            line_counter <= line_counter + 1;
        END IF;
    END IF;
END PROCESS;

line_address_generator: PROCESS(masterhclk,reset_n)
```

```
BEGIN
    IF reset_n = '0' THEN
        current_line_address <= (others => '0');
    ELSIF rising_edge(masterhclk) THEN
        -- Load the current_line_address with the base address of the
        -- current frame buffer whenever the vga driver is asserting
        -- the vertical blanking signal.
        IF dma_vblank = '0' THEN
            current_line_address <= buffer_address;
        -- Otherwise if we are starting a new line that is not the
        -- first line of a frame we will keep the current address value.
        ELSIF end_transfer = '1' AND masterhbusreq_sig = '0' THEN
            current_line_address <= masterhaddr_sig;
        END IF;
    END IF;
END PROCESS;

-- Keep track of the number of burst we have read across the
-- PLD-to-Stripe bridge
burst_count: PROCESS(masterhclk,reset_n)
BEGIN
    IF reset_n = '0' THEN
        burst_counter <= (others => '0');
    ELSIF rising_edge(masterhclk) THEN
        IF end_transfer = '1' THEN
            burst_counter <= (others => '0');
        ELSIF incr_burst_counter = '1' THEN
            burst_counter <= burst_counter + 1;
        END IF;
    END IF;
END PROCESS;

ahb_transactor_fsm: PROCESS(masterhclk,reset_n)
BEGIN
    IF reset_n = '0' THEN
        ahb_state <= idle;
        incr_burst_beat_counter <= '0';
    ELSIF rising_edge(masterhclk) THEN
        CASE ahb_state IS
            WHEN idle =>
                -- Begin an AHB transaction if the DMA controller has
                -- been granted the bus and if the control state machine
                -- has signalled that it is beginning a transfer.
                IF start_transfer = '1' AND masterhgrant = '1'
                                        AND masterhready = '1' THEN
                    ahb_state <= nonseq;
                -- Return to the nonseq state if we have been sent to the idle
                -- state due to crossing a 1K boundary or if we had
                -- an hready timeout.
                ELSIF control_state = transmit_line AND masterhgrant = '1'
                                                    AND masterhready = '1' THEN
                    ahb_state <= nonseq;
                ELSE
                    ahb_state <= idle;
                END IF;
```

```
                -- incr_burst_beat_counter is used to increment the burst beat
                -- counter but it is also used as a wren signal for the video line
                -- buffer.  Therefore it is important to make sure that this signal
                -- is high for the very last word of a line.
                IF burst_beat_counter /= 0 AND burst_beat_counter /= 7
                                           AND masterhready = '1' THEN
                    incr_burst_beat_counter <= '1';
                ELSIF end_transfer = '1' THEN
                    incr_burst_beat_counter <= '1';
                ELSE
                    incr_burst_beat_counter <= '0';
                END IF;

            -- We can always transition to the seq state from the nonseq state.
            WHEN nonseq =>
                IF burst_beat_counter /=7 AND burst_beat_counter /=0
                                          AND masterhready = '1' THEN
                    incr_burst_beat_counter <= '1';
                ELSIF burst_beat_counter = 7 AND incr_burst_beat_counter = '0'
                                             AND masterhready = '1' THEN
                    incr_burst_beat_counter <= '1';
                ELSE
                    incr_burst_beat_counter <= '0';
                END IF;

                ahb_state <= seq;

            WHEN seq =>
                -- We will leave the seq state and transition to idle state if
                -- we have reached the end of a line.
                IF end_transfer = '1' OR enable_dma = '0' THEN
                    ahb_state <= idle;
                -- This case takes care of 1K boundaries.  We know that we need to
                -- do a idle transfer if we have an address value of 0xXXXXX3FC as
                -- the next address is a 1K boundary.
                ELSIF masterhaddr_sig(9 downto 0) = "11" & X"FC" THEN
                    ahb_state <= idle;
                ELSE
                    ahb_state <= seq;
                END IF;
                IF masterhready = '1' THEN
                    incr_burst_beat_counter <= '1';
                ELSE
                    incr_burst_beat_counter <= '0';
                END IF;
            WHEN others =>
                null;
        END CASE;
    END IF;
END PROCESS;

-- Decide when we should increment the masterhaddr signal and
-- drive the htrans signal.
decode_ahb_transactor_fsm: PROCESS(ahb_state,masterhready,masterhgrant,
                      masterhbusreq_sig,control_state,masterhaddr_sig)
```

```
BEGIN
    CASE ahb_state IS
        WHEN idle =>
            masterhtrans <= "00";  -- idle
            IF masterhready = '1' AND masterhgrant = '1'
                                  AND masterhbusreq_sig = '1'
                                   AND masterhaddr_sig(4 downto 0)
                                    = '1' & X"C" THEN
                incr_masterhaddr_sig <= '1';
            ELSE
                incr_masterhaddr_sig <= '0';
            END IF;

        -- We are never in the nonseq state for more than one consecutive
        -- clock cycle so we can always increment the haddr signal when in
        -- this state.
        WHEN nonseq =>
            masterhtrans <= "10";  -- nonseq
            incr_masterhaddr_sig <= '1';

        WHEN seq =>
            masterhtrans <= "11";  -- seq
            IF masterhready = '1' AND masterhgrant = '1'
                                  AND masterhbusreq_sig = '1' THEN
                incr_masterhaddr_sig <= '1';
            ELSE
                incr_masterhaddr_sig <= '0';
            END IF;
    END CASE;
END PROCESS;

-- Generate the address to send to the bridge.  The address should be incremented
-- by 4 if we are in any state other than the idle state and the slave has
-- asserted the masterhready signal.
masterhaddr_sig_generator: PROCESS(masterhclk,reset_n)
BEGIN
    IF reset_n = '0' THEN
        masterhaddr_sig <= (others => '0');
    ELSIF rising_edge(masterhclk) THEN
        IF start_transfer = '1' THEN
            masterhaddr_sig <= current_line_address;
        ELSE
            IF incr_masterhaddr_sig = '1' THEN
                masterhaddr_sig <= masterhaddr_sig + 4;
            END IF;
        END IF;
    END IF;
END PROCESS;


-- Keep track of the number of beats within a burst transfer.  An XA10 will send 8
-- beats in a burst whereas an XA1 will only send 4.  This counter counts 8 bursts
-- but it will work for both XA1 and XA10 devices.
burst_beat_counter_generator: PROCESS(masterhclk,reset_n)
BEGIN
```

```
    IF reset_n = '0' THEN
        burst_beat_counter <= (others => '0');
    ELSIF rising_edge(masterhclk) THEN
        IF burst_beat_counter = 7 AND incr_burst_beat_counter = '1' THEN
            burst_beat_counter <= (others => '0');
        ELSIF incr_burst_beat_counter = '1' THEN
            burst_beat_counter <= burst_beat_counter + 1;
        END IF;
    END IF;
END PROCESS;

-- here is yet another counter.  This one is used to index the addresses into the
-- line buffer. It should be incremented whenver the incr_burst_beat_counter is
-- active and it should reset when we hit the last horizontal line.
rx_buffer_index: PROCESS(masterhclk,reset_n)
BEGIN
    IF reset_n = '0' THEN
        rx_buffer_wraddress_sig <= (others => '0');
    ELSIF rising_edge(masterhclk) THEN
        IF rx_buffer_wraddress_sig = (num_words_per_line-1)
                    AND incr_burst_beat_counter = '1' THEN
            rx_buffer_wraddress_sig <= (others => '0');
        ELSIF incr_burst_beat_counter = '1' THEN
            rx_buffer_wraddress_sig <= rx_buffer_wraddress_sig + 1;
        END IF;
    END IF;
END PROCESS;

dma_data <= buffered_hrdata;
dma_data_valid <= incr_burst_beat_counter;

END rtl;
```

line_buffer.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY line_buffer IS
    PORT
    (
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wraddress       : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
        rdaddress       : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
        wren        : IN STD_LOGIC  := '1';
        rden        : IN STD_LOGIC  := '1';
        wrclock     : IN STD_LOGIC ;
        rdclock     : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END line_buffer;
```

```
ARCHITECTURE SYN OF line_buffer IS

    SIGNAL sub_wire0    : STD_LOGIC_VECTOR (31 DOWNTO 0);



    COMPONENT lpm_ram_dp
    GENERIC (
        lpm_width       : NATURAL;
        lpm_widthad     : NATURAL;
        rden_used       : STRING;
        intended_device_family      : STRING;
        lpm_indata      : STRING;
        lpm_wraddress_control       : STRING;
        lpm_rdaddress_control       : STRING;
        lpm_outdata     : STRING;
        use_eab     : STRING;
        lpm_type        : STRING
    );
    PORT (
            rdclock : IN STD_LOGIC ;
            wren    : IN STD_LOGIC ;
            wrclock : IN STD_LOGIC ;
            q   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
            rden    : IN STD_LOGIC ;
            data    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            rdaddress   : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
            wraddress   : IN STD_LOGIC_VECTOR (8 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    q    <= sub_wire0(31 DOWNTO 0);

    lpm_ram_dp_component : lpm_ram_dp
    GENERIC MAP (
        lpm_width => 32,
        lpm_widthad => 9,
        rden_used => "TRUE",
        intended_device_family => "UNUSED",
        lpm_indata => "REGISTERED",
        lpm_wraddress_control => "REGISTERED",
        lpm_rdaddress_control => "REGISTERED",
        lpm_outdata => "REGISTERED",
        use_eab => "ON",
        lpm_type => "LPM_RAM_DP"
    )
    PORT MAP (
        rdclock => rdclock,
        wren => wren,
        wrclock => wrclock,
        rden => rden,
        data => data,
        rdaddress => rdaddress,
        wraddress => wraddress,
```

```
        q => sub_wire0
    );



END SYN;

-- ============================================================
-- CNX file retrieval info
-- ============================================================
-- Retrieval info: PRIVATE: WidthData NUMERIC "32"
-- Retrieval info: PRIVATE: WidthAddr NUMERIC "9"
-- Retrieval info: PRIVATE: Clock NUMERIC "1"
-- Retrieval info: PRIVATE: rden NUMERIC "1"
-- Retrieval info: PRIVATE: UseDPRAM NUMERIC "0"
-- Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "EXCALIBUR_ARM"
-- Retrieval info: PRIVATE: REGdata NUMERIC "1"
-- Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
-- Retrieval info: PRIVATE: REGwren NUMERIC "1"
-- Retrieval info: PRIVATE: REGrdaddress NUMERIC "1"
-- Retrieval info: PRIVATE: REGrren NUMERIC "1"
-- Retrieval info: PRIVATE: REGq NUMERIC "1"
-- Retrieval info: PRIVATE: enable NUMERIC "0"
-- Retrieval info: PRIVATE: CLRdata NUMERIC "0"
-- Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
-- Retrieval info: PRIVATE: CLRwren NUMERIC "0"
-- Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
-- Retrieval info: PRIVATE: CLRrren NUMERIC "0"
-- Retrieval info: PRIVATE: CLRq NUMERIC "0"
-- Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
-- Retrieval info: PRIVATE: MIFfilename STRING ""
-- Retrieval info: PRIVATE: UseLCs NUMERIC "0"
-- Retrieval info: CONSTANT: LPM_WIDTH NUMERIC "32"
-- Retrieval info: CONSTANT: LPM_WIDTHAD NUMERIC "9"
-- Retrieval info: CONSTANT: RDEN_USED STRING "TRUE"
-- Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "UNUSED"
-- Retrieval info: CONSTANT: LPM_INDATA STRING "REGISTERED"
-- Retrieval info: CONSTANT: LPM_WRADDRESS_CONTROL STRING "REGISTERED"
-- Retrieval info: CONSTANT: LPM_RDADDRESS_CONTROL STRING "REGISTERED"
-- Retrieval info: CONSTANT: LPM_OUTDATA STRING "REGISTERED"
-- Retrieval info: CONSTANT: USE_EAB STRING "ON"
-- Retrieval info: CONSTANT: LPM_TYPE STRING "LPM_RAM_DP"
-- Retrieval info: USED_PORT: data 0 0 32 0 INPUT NODEFVAL data[31..0]
-- Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL q[31..0]
-- Retrieval info: USED_PORT: wraddress 0 0 9 0 INPUT NODEFVAL wraddress[8..0]
-- Retrieval info: USED_PORT: rdaddress 0 0 9 0 INPUT NODEFVAL rdaddress[8..0]
-- Retrieval info: USED_PORT: wren 0 0 0 0 INPUT VCC wren
-- Retrieval info: USED_PORT: rden 0 0 0 0 INPUT VCC rden
-- Retrieval info: USED_PORT: wrclock 0 0 0 0 INPUT NODEFVAL wrclock
-- Retrieval info: USED_PORT: rdclock 0 0 0 0 INPUT NODEFVAL rdclock
-- Retrieval info: CONNECT: @data 0 0 32 0 data 0 0 32 0
-- Retrieval info: CONNECT: q 0 0 32 0 @q 0 0 32 0
-- Retrieval info: CONNECT: @wraddress 0 0 9 0 wraddress 0 0 9 0
-- Retrieval info: CONNECT: @rdaddress 0 0 9 0 rdaddress 0 0 9 0
-- Retrieval info: CONNECT: @wren 0 0 0 0 wren 0 0 0 0
```

```
-- Retrieval info: CONNECT: @rden 0 0 0 0 rden 0 0 0 0
-- Retrieval info: CONNECT: @wrclock 0 0 0 0 wrclock 0 0 0 0
-- Retrieval info: CONNECT: @rdclock 0 0 0 0 rdclock 0 0 0 0
-- Retrieval info: LIBRARY: lpm lpm.lpm_components.all
```

slave_interface.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY slave_interface IS
    PORT (

        -- AHB interface
        hresetn     : IN std_logic;
        hclock      : IN std_logic;
        hwrite      : IN std_logic;
        hsel        : IN std_logic;
        htrans      : IN std_logic_vector(1 downto 0);
        hsize       : IN std_logic_vector(1 downto 0);
        hburst      : IN std_logic_vector(2 downto 0);
        haddress    : IN std_logic_vector(31 downto 0);
        hwdata      : IN std_logic_vector(31 downto 0);
        hready      : OUT std_logic;
        hresp       : OUT std_logic_vector(1 downto 0);
        hrdata      : OUT std_logic_vector(31 downto 0);

        -- Inteface to DMA Controller and VGA Driver
        status          : IN std_logic_vector(31 downto 0);
        current_address     : IN std_logic_vector(31 downto 0);
        buffer_address  : OUT std_logic_vector(31 downto 0);
        image_dimensions    : OUT std_logic_vector(31 downto 0);
        control         : OUT std_logic_vector(31 downto 0)
        );
END slave_interface;

ARCHITECTURE rtl OF slave_interface IS
-- Memory Map
-- ADDRESS  NAME            DESCRIPTION
-- 0x00     buffer_addr     This register contains the address from which
--                          the DMA should begin an image xfer from
-- 0x04     image_dimensions    This address contains the number of lines
--                              in the image and the number of pixels per
--                              line
-- 0x08     control_reg     This register is used to enable or disable the
--                          DMA, and clear or enable IRQs
-- 0x0C     current_address     The current address that the DMA is
--                              reading from
-- 0x10     status          This register is used to check the current
--                          status of the DMA
-- 0x14     reserved        Reserved for future use
-- 0x18     reserved        Reserved for future use
-- 0x1C     reserved        Reserved for future use

    SIGNAL buffer_address_reg   : std_logic_vector(31 downto 0);
    SIGNAL image_dimensions_reg : std_logic_vector(31 downto 0);
```

```
    SIGNAL control_reg      : std_logic_vector(31 downto 0);

    SIGNAL internal_write   : std_logic;
    SIGNAL internal_address : std_logic_vector(2 downto 0);

    TYPE state_type IS (address,data);
    SIGNAL state : state_type;

BEGIN

-- We are always ready and we always respond with an OKAY repsponces
-- to the initiating master.
hready <= '1';
hresp <= "00";

-- Create a FSM to control the internal read and write signals to
-- the register bank.
PROCESS(hclock,hresetn)
BEGIN
    IF hresetn = '0' THEN
        internal_write <= '0';
        state <= address;
    ELSIF rising_edge(hclock) THEN
        CASE state IS
            WHEN address =>
                IF hsel = '1' AND htrans = "10" THEN
                    IF hwrite = '1' THEN
                        internal_write <= '1';
                    ELSE
                        internal_write <= '0';
                    END IF;
                    state <= data;
                ELSE
                    internal_write <= '0';
                    state <= address;
                END IF;
            WHEN data =>
                -- Remain in data state on burst transfers
                IF htrans = "11" THEN
                    IF hwrite = '1' THEN
                        internal_write <= '1';
                    ELSE
                        internal_write <= '0';
                    END IF;
                    state <= data;
                ELSE
                    internal_write <= '0';
                    state <= address;
                END IF;
            WHEN others =>
                internal_write <= '0';
                state <= address;
        END CASE;
    END IF;
END PROCESS;
```

```
-- Create the Register Bank
PROCESS(hclock,hresetn)
BEGIN
    IF hresetn = '0' THEN
        internal_address <=(others => '0');
        buffer_address_reg <= (others => '0');
        image_dimensions_reg <= (others => '0');
        control_reg <= (others => '0');
        hrdata <= (others => '0');
    ELSIF rising_edge(hclock) THEN
        internal_address <= haddress(4 downto 2);
        IF internal_write = '1' THEN
            CASE internal_address IS
                WHEN "000" =>
                    buffer_address_reg <= hwdata;
                WHEN "001" =>
                    image_dimensions_reg <= hwdata;
                WHEN "010" =>
                    control_reg <= hwdata;
                -- Not all of the registers are writeable.  This design
                -- does not return any errors if the processor tries to
                -- write to a non-writeable register.  Instead the design
                -- will ignore writes to non-writeable registers.
                WHEN others =>
                    null;
            END CASE;
        END IF;
        IF hsel = '1' AND hwrite = '0' THEN
            CASE haddress(4 downto 2) IS
                WHEN "000" =>
                    hrdata <= buffer_address_reg;
                WHEN "001" =>
                    hrdata <= image_dimensions_reg;
                WHEN "010" =>
                    hrdata <= control_reg;
                WHEN "011" =>
                    hrdata <= current_address;
                WHEN "100" =>
                    hrdata <= status;
                WHEN "101" =>
                    hrdata <= (others => '0');
                WHEN "110" =>
                    hrdata <= (others => '0');
                WHEN "111" =>
                    hrdata <= (others => '0');
                WHEN others =>
                    hrdata <= (others => '0');
            END CASE;
        END IF;
    END IF;
END PROCESS;

-- These registers get assigned to output pins because they are used by
```

```
-- the DMA controller and the VGA driver
buffer_address <= buffer_address_reg;
image_dimensions <= image_dimensions_reg;
control <= control_reg;

END rtl;
```

vga_driver.vhd

```
LIBRARY ieee,work;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;
USE work.image_package.ALL;
USE work.video_components.ALL;

ENTITY vga_driver IS
    PORT (
        reset_n         : IN std_logic;
        clock25         : IN std_logic;
        enable          : IN std_logic;

        -- video memory signals
        video_data      : IN std_logic_vector(31 downto 0);
        video_address       : OUT std_logic_vector(31 downto 0);

        -- parameter signals from AHB slave interface
        num_lines       : IN std_logic_vector(9 downto 0);
        num_pixels_per_line : IN std_logic_vector(9 downto 0);

        -- blanking outputs to any external logic that might need it
        hblank          : OUT std_logic;
        vblank          : OUT std_logic;

        -- outputs to VGA daughter card
        vsync           : OUT std_logic;
        hsync           : OUT std_logic;
        R           : OUT std_logic_vector(2 downto 0);
        G           : OUT std_logic_vector(2 downto 0);
        B           : OUT std_logic_vector(1 downto 0)
    );
END vga_driver;

ARCHITECTURE rtl OF vga_driver IS

    SIGNAL pixel_counter        : std_logic_vector(9 downto 0);
    SIGNAL line_counter     : std_logic_vector(9 downto 0);
    SIGNAL word_sel         : std_logic_vector(1 downto 0);
    SIGNAL video_address_sig    : std_logic_vector(31 downto 0);
    SIGNAL vblank_sig       : std_logic;
    SIGNAL hblank_sig       : std_logic;
    SIGNAL clockext_sig     : std_logic;

    -- These signals determine the location of an images smaller than
    -- 640x480 on the screen
    SIGNAL x_offset         : std_logic_vector(9 downto 0);
```

```
    SIGNAL y_offset         : std_logic_vector(9 downto 0);

    SIGNAL active           : std_logic;
    SIGNAL active_delay     : std_logic;

    SIGNAL actual_num_lines : std_logic_vector(9 downto 0);
    SIGNAL actual_num_pixels_per_line : std_logic_vector(9 downto 0);
    SIGNAL last_hblank      : std_logic;

BEGIN

hblank <= hblank_sig;
vblank <= vblank_sig;

pll_ext: vga_pll
PORT MAP (
    inclock => clock25,
    clock1 => clockext_sig
);

PROCESS(video_address_sig,num_lines,num_pixels_per_line)
BEGIN
    video_address <= video_address_sig;
    actual_num_lines <= num_lines;
    actual_num_pixels_per_line <= num_pixels_per_line;
END PROCESS;

-- This is a roll-over counter which will count a period
-- from 0 to HWIDTH.  It is used to update the pixels as
-- horizontally across the screen.
pixel_count: PROCESS(clock25,reset_n)
BEGIN
    IF reset_n = '0' THEN
        pixel_counter <= (others => '0');
    ELSIF rising_edge(clock25) THEN
        IF pixel_counter < HWIDTH THEN
            pixel_counter <= pixel_counter + 1;
        ELSE
            pixel_counter <= (others => '0');
        END IF;
    END IF;
END PROCESS;

-- This counter keeps track of the current line being displayed
-- on the screen.  It resets when it reaches a value of VDEPTH.
-- The line counter is enabled whenever the pixel counter has
-- reached a value of END_HSYNC
line_count: PROCESS(clock25,reset_n)
BEGIN
    IF reset_n = '0' THEN
        line_counter <= (others => '0');
    ELSIF rising_edge(clock25) THEN
        IF pixel_counter = END_HSYNC THEN
            IF line_counter < VDEPTH THEN
                line_counter <= line_counter + 1;
```

```
            ELSE
                line_counter <= (others => '0');
            END IF;
        END IF;
    END IF;
END PROCESS;

-- The driver supports images that are smaller than 640x480.  It looks a little
-- nicer to display smaller images directly in the center of the screen so we
-- need to calculate the offset in both the x and y domains.
PROCESS(clock25,reset_n)
    VARIABLE x_offset_reg : std_logic_vector(9 downto 0);
    VARIABLE y_offset_reg : std_logic_vector(9 downto 0);
BEGIN
    IF reset_n = '0' THEN
        x_offset_reg := (others => '0');
        y_offset_reg := (others => '0');
    ELSIF rising_edge(clock25) THEN
        IF vblank_sig = '0' THEN
            x_offset_reg := LINE_WIDTH - actual_num_pixels_per_line;
            y_offset_reg := NUMBER_LINES - actual_num_lines;
        END IF;
    END IF;
    -- divide offsets by 2 to balance borders on each side of image
    x_offset <= '0' & x_offset_reg(9 downto 1);
    y_offset <= '0' & y_offset_reg(9 downto 1);
END PROCESS;

-- Generation of the hsync and hblank pulses and address signal to
-- line buffer
-- *** Important Note ***
-- This vga driver pulls images from a line buffer therefore the video
-- adddress signal to the line buffer will get reset after each line
-- has been transmitted. If at some point a frame buffer is needed
-- instead of a line buffer then this process will need to be modified
-- such that the address gets reset on every frame instead of on every
-- line.
hsync_gen: PROCESS(clock25,reset_n)
BEGIN
    IF reset_n = '0' THEN
        hsync <= '0';
        hblank_sig <= '0';
        video_address_sig <= (others => '0');
        active <= '0';
        vblank_sig <= '0';
    ELSIF rising_edge(clock25) THEN
        -- generate the blanking signal and the address to the line buffer
        IF (pixel_counter >= BEGIN_HBLANK AND pixel_counter < END_HBLANK) THEN
            hblank_sig <= last_hblank;
        ELSE
            hblank_sig <= '0';
        END IF;

        -- Generate the active signal.  This signal indicates when we are
        -- driving a pixel out.  The active singal would normally be
```

```
        -- asserted for the duration of hblank for a 640x480 image. However,
        -- for smaller images we want to wrap a black border around the
        -- image. Therefore the active signal will be of shorter duration
        -- than the hblank signal when transmitting smaller images.
        IF (line_counter > (CONV_STD_LOGIC_VECTOR(BEGIN_VBLANK,10) + y_offset)
         AND line_counter <= (CONV_STD_LOGIC_VECTOR(END_VBLANK,10) - y_offset))
          THEN
            IF (pixel_counter >
                 (CONV_STD_LOGIC_VECTOR(BEGIN_HBLANK,10) + x_offset-2)
               AND pixel_counter <= (CONV_STD_LOGIC_VECTOR(END_HBLANK,10)
                                     - x_offset-2)) THEN
                active <= '1';
                video_address_sig <= video_address_sig + 1;
            ELSE
                active <= '0';
                video_address_sig <= (others => '0');
            END IF;
        ELSE
            active <= '0';
            video_address_sig <= (others => '0');
        END IF;
        IF (line_counter > (CONV_STD_LOGIC_VECTOR(BEGIN_VBLANK,10) + y_offset-1)
         AND line_counter <= (CONV_STD_LOGIC_VECTOR(END_VBLANK,10) - y_offset-2))
          THEN
            vblank_sig <= '1';
        ELSE
            vblank_sig <= '0';
        END IF;

        -- Generate the hsync signal to send the the VGA daughter card
        IF (pixel_counter < END_HSYNC) THEN
            hsync <= '0';
        ELSE
            hsync <= '1';
        END IF;
    END IF;
END PROCESS;

PROCESS(clock25,reset_n)
BEGIN
    IF reset_n = '0' THEN
        last_hblank <= '0';
    ELSIF rising_edge(clock25) THEN
        last_hblank <= '1';
    END IF;
END PROCESS;

-- Generate the vertical synchronization and blanking signals
vsync_gen: PROCESS(clock25,reset_n)
BEGIN
    IF reset_n = '0' THEN
        vsync <= '0';
    ELSIF rising_edge(clock25) THEN
        -- If the hsync signal is transitioning from a 1 to a 0 then...
        IF pixel_counter = end_hsync THEN
```

```
            IF (line_counter < END_VSYNC) THEN
                vsync <= '0';
            ELSE
                vsync <= '1';
            END IF;
        END IF;
    END IF;
END PROCESS;

-- This is a 16-bit driver.  Words coming out of the line buffer are
-- 32-bits.  Therefore we will read a single 32-bit word twice in
-- order to extract the 2 16-bit words.  The half_word_sel signal
-- determines which half-word we are reading.
PROCESS(video_address_sig(1 downto 0))
BEGIN
    word_sel(1 downto 0) <= video_address_sig(1 downto 0);
END PROCESS;

-- This process generates the RGB signals.  Whenever the active signal
-- is asserted the process will drive the video_data out onto the RGB
-- signals.  If active is low the RGB signals will be driven to black.
-- The 16-bit video_data signal is also converted to a 24-bit RGB
-- value by stuffing 1's onto the LSBs of the RGB signals.
color_generator: PROCESS(reset_n,clock25)
BEGIN
    IF reset_n = '0' THEN
        R <= (others => '1');
        G <= (others => '1');
        B <= (others => '1');
        active_delay <= '0';
    ELSIF rising_edge(clock25) THEN
        active_delay <= active;
        IF active_delay = '1' AND enable = '1' THEN
            IF word_sel = "10" THEN
                R <= not video_data(7 downto 5);
                G <= not video_data(4 downto 2);
                B <= not video_data(1 downto 0);
            ELSIF word_sel = "11" THEN
                R <= not video_data(15 downto 13);
                G <= not video_data(12 downto 10);
                B <= not video_data(9 downto 8);
            ELSIF word_sel = "00" THEN
                R <= not video_data(23 downto 21);
                G <= not video_data(20 downto 18);
                B <= not video_data(17 downto 16);
            ELSE
                R <= not video_data(31 downto 29);
                G <= not video_data(28 downto 26);
                B <= not video_data(25 downto 24);
            END IF;
        ELSE
            R <= "111";
            G <= "111";
            B <= "11";
        END IF;
```

```
    END IF;
END PROCESS;

END rtl;
```

slave_decoder.vhd

```
-- slave_decoder.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY slave_decoder IS PORT
    (
    -- bridge signals
    hbusreq         : IN std_logic;
    haddr           : IN std_logic_vector(31 downto 0);

    -- outputs to slave iface
    hsel_slave_iface    : OUT std_logic;
    hsel_default_slave  : OUT std_logic
    );
END slave_decoder;

ARCHITECTURE rtl OF slave_decoder IS

    CONSTANT siface_base_address    : std_logic_vector(31 downto 0) := X"80000000";

BEGIN

PROCESS(hbusreq,haddr)
BEGIN
    IF hbusreq = '1' AND haddr(31 downto 10) =
                    siface_base_address(31 downto 10) THEN
        hsel_slave_iface <= '1';
        hsel_default_slave <= '0';
    -- The AHB spec defines slave as having a minimum 1K address space.  The slave
    -- interface only contains 8 registers so accesses to address within the 1K
    -- address space of the slave that are not targeting one of the 8 registers
    -- will still generate an error response.
    ELSIF hbusreq = '1' AND haddr(31 downto 10) /=
                    siface_base_address(31 downto 10) THEN
        hsel_slave_iface <= '0';
        hsel_default_slave <= '1';
    END IF;
END PROCESS;

END rtl;
```

default_slave.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY default_slave IS PORT (
    hclock      : IN std_logic;
```

```
    hresetn     : IN std_logic;
    hsel        : IN std_logic;
    htrans      : IN std_logic_vector(1 downto 0);
    hready      : OUT std_logic;
    hresp       : OUT std_logic_vector(1 downto 0);
    hrdata      : OUT std_logic_vector(31 downto 0)
);
END default_slave;

ARCHITECTURE rtl OF default_slave IS

    TYPE state_type IS (address_phase,error_phase);
    SIGNAL state : state_type;

BEGIN

-- we don't care what gets driven onto hrdata so drive 0's for simplicity
hrdata <= (others => '0');

PROCESS(hclock,hresetn)
BEGIN
    IF hresetn = '0' THEN
        state <= address_phase;
    ELSIF rising_edge(hclock) THEN
        CASE state IS
            WHEN address_phase =>
                IF hsel = '1' THEN
                    -- check for SEQ or NONSEQ transaction
                    IF htrans = "10" OR htrans = "11" THEN
                        state <= error_phase;
                        hresp <= "01";  -- ERROR responce
                        hready <= '0';
                    ELSE
                        state <= address_phase;
                        hresp <= "00";  -- OKAY responce
                        hready <= '1';
                    END IF;
                ELSE
                    state <= address_phase;
                    hresp <= "00";  -- OKAY responce
                    hready <= '0';
                END IF;
            WHEN error_phase =>
                state <= address_phase;
                hresp <= "01";  -- ERROR responce
                hready <= '1';
        END CASE;
    END IF;
END PROCESS;

END rtl;
```

resposne_and_data_mux.vhd

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;

ENTITY response_and_data_mux IS PORT (
    -- system signals
    hclock          : IN std_logic;
    hresetn         : IN std_logic;

    -- slave iface signals
    hsel_slave_iface    : IN std_logic;
    hready_slave_iface  : IN std_logic;
    hresp_slave_iface   : IN std_logic_vector(1 downto 0);
    hrdata_slave_iface  : IN std_logic_vector(31 downto 0);

    -- default slave signals
    hsel_default_slave  : IN std_logic;
    hready_default_slave    : IN std_logic;
    hresp_default_slave : IN std_logic_vector(1 downto 0);

    -- outputs to bridge
    hready          : OUT std_logic;
    hresp           : OUT std_logic_vector(1 downto 0);
    hrdata          : OUT std_logic_vector(31 downto 0));
END response_and_data_mux;

ARCHITECTURE rtl OF response_and_data_mux IS

    SIGNAL hsel_slave_iface_delay : std_logic;

BEGIN

-- delay the hsel signal by 1 clock cycle as the data phase of a
-- transaction occurs 1 clock cycle after the address phase;
PROCESS(hclock,hresetn)
BEGIN
    IF hresetn = '0' THEN
        hsel_slave_iface_delay <= '0';
    ELSIF rising_edge(hclock) THEN
        hsel_slave_iface_delay <= hsel_slave_iface;
    END IF;
END PROCESS;

-- hrdata mux
PROCESS(hsel_slave_iface_delay,hsel_default_slave,hrdata_slave_iface,
                              hresp_slave_iface,hresp_default_slave)
BEGIN
    IF hsel_slave_iface_delay = '1' THEN
        hrdata <= hrdata_slave_iface;
        hresp <= hresp_slave_iface;
    ELSE
        hresp <= hresp_default_slave;
        hrdata <= (others => '0');
              -- we are idle or have a default slave hit
    END IF;
END PROCESS;
```

```
hready <= hready_slave_iface AND hready_default_slave;

END rtl;
```

video_components.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE video_components IS

COMPONENT vga_pll PORT (
    inclock : IN std_logic;
    clock1 : OUT std_logic
);
END COMPONENT;

COMPONENT vga_driver
    PORT (
        reset_n     : IN std_logic;
        clock25     : IN std_logic;
        enable      : IN std_logic;

        -- video memory signals
        video_data  : IN std_logic_vector(31 downto 0);
        video_address   : OUT std_logic_vector(31 downto 0);

        num_lines       : IN std_logic_vector(9 downto 0);
        num_pixels_per_line : IN std_logic_vector(9 downto 0);

        -- blanking and clock outputs to any logic that could use it
        hblank      : OUT std_logic;
        vblank      : OUT std_logic;

        -- outputs to VGA daughter card
        vsync       : OUT std_logic;
        hsync       : OUT std_logic;
        R       : OUT std_logic_vector(2 downto 0);
        G       : OUT std_logic_vector(2 downto 0);
        B       : OUT std_logic_vector(1 downto 0)
    );
END COMPONENT;

COMPONENT line_buffer
    PORT
    (
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wraddress   : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
        rdaddress   : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
        wren        : IN STD_LOGIC  := '1';
        rden        : IN STD_LOGIC  := '1';
        wrclock     : IN STD_LOGIC ;
        rdclock     : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
```

```
END COMPONENT;

COMPONENT slave_interface
    PORT (

        -- AHB interface
        hresetn     : IN std_logic;
        hclock      : IN std_logic;
        hwrite      : IN std_logic;
        hsel        : IN std_logic;
        htrans      : IN std_logic_vector(1 downto 0);
        hsize       : IN std_logic_vector(1 downto 0);
        hburst      : IN std_logic_vector(2 downto 0);
        haddress    : IN std_logic_vector(31 downto 0);
        hwdata      : IN std_logic_vector(31 downto 0);
        hready      : OUT std_logic;
        hresp       : OUT std_logic_vector(1 downto 0);
        hrdata      : OUT std_logic_vector(31 downto 0);

        -- Inteface to DMA Controller and VGA Driver
        status          : IN std_logic_vector(31 downto 0);
        current_address     : IN std_logic_vector(31 downto 0);
        buffer_address  : OUT std_logic_vector(31 downto 0);
        image_dimensions    : OUT std_logic_vector(31 downto 0);
        control         : OUT std_logic_vector(31 downto 0)
        );
END COMPONENT;

COMPONENT default_slave
    PORT (
        hclock      : IN std_logic;
        hresetn     : IN std_logic;
        hsel        : IN std_logic;
        htrans      : IN std_logic_vector(1 downto 0);
        hready      : OUT std_logic;
        hresp       : OUT std_logic_vector(1 downto 0);
        hrdata      : OUT std_logic_vector(31 downto 0)
    );
END COMPONENT;

COMPONENT response_and_data_mux
    PORT (
    -- system signals
    hclock          : IN std_logic;
    hresetn         : IN std_logic;

    -- slave iface signals
    hsel_slave_iface    : IN std_logic;
    hready_slave_iface  : IN std_logic;
    hresp_slave_iface   : IN std_logic_vector(1 downto 0);
    hrdata_slave_iface  : IN std_logic_vector(31 downto 0);

    -- default slave signals
    hsel_default_slave  : IN std_logic;
    hready_default_slave    : IN std_logic;
```

```
    hresp_default_slave : IN std_logic_vector(1 downto 0);

    -- outputs to bridge
    hready          : OUT std_logic;
    hresp           : OUT std_logic_vector(1 downto 0);
    hrdata          : OUT std_logic_vector(31 downto 0));
END COMPONENT;

COMPONENT slave_decoder PORT
    (
    -- bridge signals
    hbusreq         : IN std_logic;
    haddr           : IN std_logic_vector(31 downto 0);

    -- outputs to slave iface
    hsel_slave_iface    : OUT std_logic;
    hsel_default_slave  : OUT std_logic
    );
END COMPONENT;

COMPONENT system_pll
    PORT
    (
        inclock     : IN STD_LOGIC ;
        clock0      : OUT STD_LOGIC ;
        clock1      : OUT STD_LOGIC
    );
END COMPONENT;

COMPONENT video_dma_controller
    PORT (

    -- inputs from AHB slave side interface
    buffer_address  : IN std_logic_vector(31 downto 0);
    num_words_per_line  : IN std_logic_vector(15 downto 0);
    num_lines       : IN std_logic_vector(15 downto 0);
    enable_dma      : IN std_logic;

    -- inputs from VGA driver
    vblank          : IN std_logic;
    hblank          : IN std_logic;

    -- AHB master interface signals
    -- These signals connect to the PLD-to-stripe bridge
    reset_n         : IN std_logic;
    masterhclk      : IN std_logic;
    masterhgrant        : IN std_logic;
    masterhready        : IN std_logic;
    masterhrdata        : IN std_logic_vector(31 downto 0);
    masterhlock     : OUT std_logic;
    masterhaddr     : OUT std_logic_vector(31 downto 0);
    masterhbusreq       : OUT std_logic;
    masterhsize     : OUT std_logic_vector(1 downto 0);
    masterhtrans        : OUT std_logic_vector(1 downto 0);
    masterhburst        : OUT std_logic_vector(2 downto 0);
```

```
    masterhwdata        : OUT std_logic_vector(31 downto 0);

    -- data and control signals for output of DMA
    set_irq         : OUT std_logic;
    buffer_select       : OUT std_logic;
    dma_data        : OUT std_logic_vector(31 downto 0);
    rx_buffer_wraddress : OUT std_logic_vector(9 downto 0);
    dma_data_valid      : OUT std_logic
    );
END COMPONENT;

COMPONENT video_dma
    PORT
    (
        -- System signals
        reset_n             : IN std_logic;
        hclock              : IN std_logic;
             -- clock to the AHB master interface
        clock25             : IN std_logic;
             -- clock to the VGA driver
        clock50             : IN std_logic;
             -- clock to the slave interface and VGA driver

        --Slave Interface signals
        siface_hsel         : IN std_logic;
        siface_hwrite_from_stripe   : IN std_logic;
        siface_hbusreq_from_stripe  : IN std_logic;
        siface_htrans_from_stripe   : IN std_logic_vector(1 downto 0);
        siface_hsize_from_stripe    : IN std_logic_vector(1 downto 0);
        siface_hburst_from_stripe   : IN std_logic_vector(2 downto 0);
        siface_haddress_from_stripe : IN std_logic_vector(31 downto 0);
        siface_hwdata_from_stripe   : IN std_logic_vector(31 downto 0);
        siface_hready_to_stripe     : OUT std_logic;
        siface_hresp_to_stripe      : OUT std_logic_vector(1 downto 0);
        siface_hrdata_to_stripe     : OUT std_logic_vector(31 downto 0);

        -- DMA Controller Master Interface signals
        miface_hready_from_stripe   : IN std_logic;
        miface_hgrant_from_stripe   : IN std_logic;
        irq             : OUT std_logic;  -- this is currently not used
        miface_hlock_to_stripe      : OUT std_logic;
        miface_hbusreq_to_stripe    : OUT std_logic;
        miface_htrans_to_stripe     : OUT std_logic_vector(1 downto 0);
        miface_hsize_to_stripe      : OUT std_logic_vector(1 downto 0);
        miface_hburst_to_stripe     : OUT std_logic_vector(2 downto 0);
        miface_hwdata_to_stripe     : OUT std_logic_vector(31 downto 0);
        miface_hrdata_from_stripe   : IN std_logic_vector(31 downto 0);
        miface_haddr_to_stripe      : OUT std_logic_vector(31 downto 0);

        -- VGA outputs
        vsync               : OUT std_logic;
        hsync               : OUT std_logic;
        R               : OUT std_logic_vector(2 downto 0);
        G               : OUT std_logic_vector(2 downto 0);
        B               : OUT std_logic_vector(1 downto 0)
```

```
    );
END COMPONENT;

COMPONENT stripe
    PORT (
        clk_ref : IN    STD_LOGIC;
        npor    : IN    STD_LOGIC;
        nreset  : INOUT STD_LOGIC;
        uartrxd : IN    STD_LOGIC;
        uartdsrn    : IN    STD_LOGIC;
        uartctsn    : IN    STD_LOGIC;
        uartrin : INOUT STD_LOGIC;
        uartdcdn    : INOUT STD_LOGIC;
        uarttxd : OUT   STD_LOGIC;
        uartrtsn    : OUT   STD_LOGIC;
        uartdtrn    : OUT   STD_LOGIC;
        intextpin   : IN    STD_LOGIC;
        ebiack  : IN    STD_LOGIC;
        ebidq   : INOUT STD_LOGIC_VECTOR(15 downto 0);
        ebiclk  : OUT   STD_LOGIC;
        ebiwen  : OUT   STD_LOGIC;
        ebioen  : OUT   STD_LOGIC;
        ebiaddr : OUT   STD_LOGIC_VECTOR(24 downto 0);
        ebibe   : OUT   STD_LOGIC_VECTOR(1 downto 0);
        ebicsn  : OUT   STD_LOGIC_VECTOR(3 downto 0);
        sdramdq : INOUT STD_LOGIC_VECTOR(15 downto 0);
        sdramdqs    : INOUT STD_LOGIC_VECTOR(1 downto 0);
        sdramclk    : OUT   STD_LOGIC;
        sdramclkn   : OUT   STD_LOGIC;
        sdramclke   : OUT   STD_LOGIC;
        sdramwen    : OUT   STD_LOGIC;
        sdramcasn   : OUT   STD_LOGIC;
        sdramrasn   : OUT   STD_LOGIC;
        sdramaddr   : OUT   STD_LOGIC_VECTOR(14 downto 0);
        sdramcsn    : OUT   STD_LOGIC_VECTOR(1 downto 0);
        sdramdqm    : OUT   STD_LOGIC_VECTOR(1 downto 0);
        slavehclk   : IN    STD_LOGIC;
        slavehwrite : IN    STD_LOGIC;
        slavehreadyi    : IN    STD_LOGIC;
        slavehselreg    : IN    STD_LOGIC;
        slavehsel   : IN    STD_LOGIC;
        slavehmastlock  : IN    STD_LOGIC;
        slavehaddr  : IN    STD_LOGIC_VECTOR(31 downto 0);
        slavehtrans : IN    STD_LOGIC_VECTOR(1 downto 0);
        slavehsize  : IN    STD_LOGIC_VECTOR(1 downto 0);
        slavehburst : IN    STD_LOGIC_VECTOR(2 downto 0);
        slavehwdata : IN    STD_LOGIC_VECTOR(31 downto 0);
        slavehreadyo    : OUT   STD_LOGIC;
        slavebuserrint  : OUT   STD_LOGIC;
        slavehresp  : OUT   STD_LOGIC_VECTOR(1 downto 0);
        slavehrdata : OUT   STD_LOGIC_VECTOR(31 downto 0);
        masterhclk  : IN    STD_LOGIC;
        masterhready    : IN    STD_LOGIC;
        masterhgrant    : IN    STD_LOGIC;
        masterhrdata    : IN    STD_LOGIC_VECTOR(31 downto 0);
```

```
        masterhresp : IN    STD_LOGIC_VECTOR(1 downto 0);
        masterhwrite    : OUT   STD_LOGIC;
        masterhlock : OUT   STD_LOGIC;
        masterhbusreq   : OUT   STD_LOGIC;
        masterhaddr : OUT   STD_LOGIC_VECTOR(31 downto 0);
        masterhburst    : OUT   STD_LOGIC_VECTOR(2 downto 0);
        masterhsize : OUT   STD_LOGIC_VECTOR(1 downto 0);
        masterhtrans    : OUT   STD_LOGIC_VECTOR(1 downto 0);
        masterhwdata    : OUT   STD_LOGIC_VECTOR(31 downto 0);
        intpld      : IN    STD_LOGIC_VECTOR(5 downto 0)
    );
END COMPONENT;

END video_components;
```

video_components.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE image_package IS
    CONSTANT hwidth         : integer := 794;
    CONSTANT vdepth         : integer := 528;
    CONSTANT line_width     : integer := 640;
    CONSTANT number_lines   : integer := 480;

    -- blanking and sync pulse constants
    CONSTANT begin_hblank   : integer := 94+46;  -- 140
    CONSTANT end_hblank     : integer := 140+line_width;
    CONSTANT end_hsync      : integer := 94;

    CONSTANT begin_vblank   : integer := 2+32;  -- 34
    CONSTANT end_vblank     : integer := 34+number_lines;
    CONSTANT end_vsync      : integer := 2;
END image_package;
```

main.c

```
#include <stdio.h>
#include "..\quartus\stripe.h"

#define FRAME_BUFFER_BASE (EXC_SDRAM_BLOCK0_BASE + 0x00100000)

volatile int *DMA = (int*) EXC_PLD_BLOCK0_BASE;
volatile int *FRAME_BUFFER = (int*) FRAME_BUFFER_BASE;

int main(void)
{
//  clear();

    // initialize the video driver
    *DMA = FRAME_BUFFER_BASE;  // base address of frame buffer
    *(DMA+1) = 0x028001E0;  // image size (640x480)
    *(DMA+2) = 0x00000001; // start driver
```

```
    printf("\r\n");
    printf("****************************************\r\n");
    printf("* VGA is running!!!                    *\r\n");
    printf("*                                      *\r\n");
    printf("*                                      *\r\n");
    printf("****************************************\r\n");
    printf("\r\n");

    while(1);

    return 0;
}
```

irq.c

```
#include <stdio.h>

#include "irq.h"
#include "stripe.h"
#include "int_ctrl00.h"
#include "uartcomm.h"

#define INT_CTRL00_TYPE (volatile unsigned int *)
#define PLDINT0_FIQ_PRI     INT_PRIORITY_P1_FQ_MSK
#define PLDINT1_IRQ_PRI     1
#define PLDINT2_IRQ_PRI     2
#define PLDINT3_IRQ_PRI     3
#define PLDINT4_IRQ_PRI     4
#define PLDINT5_IRQ_PRI     5
#define PLDINT_IRQ_PRI      1
#define UART_IRQ_PRI        10

#define FRAME_BUFFER_BASE (EXC_SDRAM_BLOCK0_BASE + 0x00100000)

int counter=0;

void uart_irq_handler(void);
void vga_irq_handler(void);

void irq_init(void)
{
    /*
    *   Disable the interrupts for all the PLD sources
    *   confusingly enough these are all on by default
    *   The reason is in case people want to implement their own
    *   interrupt controller in the PLD they don't have to write
    *   any code to enable interrupts in the Excalibur controller
    */
    *INT_MC(EXC_INT_CTRL00_BASE) =  INT_MC_P0_MSK | INT_MC_P1_MSK |
                                    INT_MC_P2_MSK | INT_MC_P3_MSK |
                                    INT_MC_P4_MSK | INT_MC_P5_MSK;

    /*
    * Set priority for the UART interrupts
```

```
    */
    *INT_PRIORITY_UA(EXC_INT_CTRL00_BASE)=UART_IRQ_PRI;

    /*
    *   Enable the UART interrupt
    */
    *INT_MS(EXC_INT_CTRL00_BASE)= INT_MS_UA_MSK;

    // Set interrupt mode to Six Individual Interrupts from the PLD
    *INT_MODE(EXC_INT_CTRL00_BASE) = INT_MODE_SIX_IND;

    // Set each PLD interrupt with different priority levels.
    // INT_PLD[0] generates a FIQ interrupt, while INT_PLD[1-5]
    // generate an IRQ interrupt.
    *INT_PRIORITY_P0(EXC_INT_CTRL00_BASE) = PLDINT0_FIQ_PRI;
    *INT_PRIORITY_P1(EXC_INT_CTRL00_BASE) = PLDINT1_IRQ_PRI;
    *INT_PRIORITY_P2(EXC_INT_CTRL00_BASE) = PLDINT2_IRQ_PRI;
    *INT_PRIORITY_P3(EXC_INT_CTRL00_BASE) = PLDINT3_IRQ_PRI;
    *INT_PRIORITY_P4(EXC_INT_CTRL00_BASE) = PLDINT4_IRQ_PRI;
    *INT_PRIORITY_P5(EXC_INT_CTRL00_BASE) = PLDINT5_IRQ_PRI;

    // Enable PLD interrupts
    *INT_MS(EXC_INT_CTRL00_BASE) |= INT_MS_P0_MSK | INT_MS_P1_MSK |
                                                    INT_MS_P2_MSK |
                                    INT_MS_P3_MSK | INT_MS_P4_MSK |
                                                    INT_MS_P5_MSK;
}

void CIrqHandler(void)
{
    volatile int irqID;

    irqID = *INT_ID(EXC_INT_CTRL00_BASE);

    switch (irqID)
    {
        case PLDINT1_IRQ_PRI:
            vga_irq_handler();
        case UART_IRQ_PRI:
            uart_irq_handler();
            break;
        default:
            /* This shouldn't happen, but let's trap it in case */
            printf("Unknown irq %#x",irqID);
            break;
    }

    return;
}

void CFiqHandler(void)
{
    /* This shouldn't happen */
    return;
}
```

```
void vga_irq_handler(void)
{
    volatile int *DMA = (int*) EXC_PLD_BLOCK0_BASE;
    int address;

    if (counter<100)
    {
        address = 0x00100000;
        counter++;
    }
    else if (counter < 200)
    {
        address = 0x00200000;
        counter++;
    }
    else if (counter < 300)
    {
        address = 0x00300000;
        counter++;

    }
    else if (counter==300)
    {
        address = 0x00100000;
        counter=0;
    }

    *DMA = address;  // base address of frame buffer
}
```

bmp2hex.pl

```
# bmp2hex.pl
unless ( @ARGV >= 1) {
    die "$0: You must enter at least 1 file name to operate on!\n";
}

# this sub-routine writes converts an address and data string into
# Intel-hex format
# The script takes 3 arguments
# arg 0 - size in bytes of the data
# arg 1 - the data itself
# arg 2 - the start address of the data
sub hex_line {
    my $size = $_[2];
    my $data;
    my $data2;
    my $rec_length;
    my $temp;
    my $msb;
    my $lsb;
    my $msb2;
    my $lsb2;
```

```
    # format the data record depending on $size
    if ($size == 1) {
        $data = $_[0];
        #$data = sprintf("%02x",$_[0]);
        $rec_length = sprintf("%02x",1);
    } elsif ($size == 2) {
        $data = $_[0];
#       $data2 = $_[1];
        #$data = sprintf("%04x",$_[0]);
        $rec_length = sprintf("%02x",2);
    } elsif ($size == 3) {
        $data = $_[0];
        $rec_length = sprintf("%02x",3);
    }

    # This swaps the endianness of the pixels
    #print "data before  is $data\n";
    # byte swap the data

    my $msb = (((hex $data) << 8) >> 8);
    my $lsb = (((hex $data) >> 8) << 8);
    my $data = $lsb.$msb;
    my $data = $data & 65535; # 65535 = 0xFFFF
    my $data = sprintf("%02x",$data);

#   my $msb2 = (((hex $data2) << 8) >> 8);
#   my $lsb2 = (((hex $data2) >> 8) << 8);
#   my $data2 = $msb2.$lsb2;
#   my $data2 = $data2 & 65535; # 65535 = 0xFFFF
#   my $data2 = sprintf("%02x",$data2);

    my $address = sprintf("%04x",$_[1]); # address are always 2 bytes
    my $rec_type = sprintf("%02x",0); # record type is 1 byte and is 00
    my $concat = $rec_length . $address . $rec_type . $data;
    # Calculate the sum of each of the bytes in the $concat variable
    my $sum = 0;
    for ($sub_index=0;$sub_index < (length $concat)/2;$sub_index++) {
        $sum = $sum + hex substr($concat,$sub_index*2,2);
    }
    # the checksum + the (sum of the data) mode 256 must be 00
    $sum = $sum % 256;
    if ($sum != 0) {
        $temp = 256 - $sum;
    } else {
        $temp = 0;
    }
    my $chksum = sprintf("%02x",$temp);
    my $line = ":".$concat.$chksum;
    return $line;
}

# this sub-routine writes converts an address and data string into
# Intel-hex format
# The script takes 3 arguments
# arg 0 - size in bytes of the data
```

```
# arg 1 - the data itself
# arg 2 - the start address of the data
sub hex_address_line {
    my $temp = 48 + $_[0];  # base address for image -> 16 = 0x00100000,
                            # 32 = 0x00200000, 48 = 0x00100000, ...
    my $ulba = sprintf("%04x",$temp);
    my $rec_length = sprintf("%02x",2); # record length is 1 byte and is 02
    my $offset = sprintf("%04x",0); # offset is 2 bytes and is 0000
    my $rec_type = sprintf("%02x",4); # record type is 1 byte and is 00

    my $concat = $rec_length . $offset . $rec_type . $ulba;
    # Calculate the sum of each of the bytes in the $concat variable
    my $sum = 0;
    for ($sub_index=0;$sub_index < (length $concat)/2;$sub_index++) {
        $sum = $sum + hex substr($concat,$sub_index*2,2);
    }
    # the checksum + the (sum of the data) mode 256 must be 00
    $sum = $sum % 256;
    if ($sum != 0) {
        $temp = 256 - $sum;
    } else {
        $temp = 0;
    }
    my $chksum = sprintf("%02x",$temp);
    my $line = ":".$concat.$chksum;
    return $line;
}

# This subbroutine converts a 24-bit RGB value into a 16-bit RGB value.
# By chopping the LSBs from each of the colors.  We chop off 3 LSBs from
# R and B and 2 LSBs from G
# arg 0 = R
# arg 1 = G
# arg 2 = B
sub color_compress {
        my $R = $_[0];
        my $G = $_[1];
        my $B = $_[2];
        my $pixel;
        # lop off LSBs and and shift R and G colors out so that
        # we can add all 3 color values to produce a 6 bit pixel.
        # Example
        # 24-bit pixel = FF8040
        # R = FF, B = 80, G = 40
        # lop of LSBs
        # R = 1F, B = 20, G = 08
        # shift R left by 11 and B left by 5 so that pixels align
        # R = F800, B = 0400, G = 08
        # add the shifted pixels to get a 16 bit pixel of value FC08
        my $R = ((hex $R) >> 5) << 5;
        my $G = ((hex $G) >> 5) << 2;
        my $B = (hex $B) >> 6;
        my $pixel = sprintf("%04X",($R+$G+$B));
        return $pixel;
}
```

```
# copy the names of the input files into an the @file_list array and
# convert each bit-map to a tabular text format using the 'od' utility
while (@ARGV) {
    $bmp_name = shift(@ARGV);
    $bmp_convert = $bmp_name;
    $bmp_convert =~ s/\.bmp/\.txt/i;
    push(@file_list,$bmp_convert);
    # call the "object dump" utility and pipe the output to $bmp_convert
    `od -x -v $bmp_name > $bmp_convert`;
}

open (HEADER, ">image_header.h") or die "$0: Can't create image_header.h: $!";
open (HEX_OUT, ">image_data.hex") or die "$0: Can't create image_data.hex: $!";

$hex_address = 0; # this is used to specify the offsets in the .hex file
$pallete_used = 0;
$bit = 0;

# loop through each of the files in @file_list array
for($index=0;$index <= $#file_list;$index++) {
    open (READ_FILE, "$file_list[$index]")
             or die "$0: Can't open $file_list[$index]: $!";
    # initialize variables for each loop iteration
    undef @bmp_array;
    @bmp_array[0] = "";
    $. = 0;
    # read the current object dump file pointed to by @file_list
    while(<READ_FILE>) {
        # strip off the address field from each line,
        # the address field is 7 bits wide
        s/\d{7}\s{1}//i;
        # pull off the various header fields from the file
        # the width and height are stored in line 2 of the text file
        if ($. == 2) {
            # the width is is col 2, height in col 4
            /\w{4}\s(\w{4})\s\w{4}\s(\w{4})\s/;
            $width = $1;
            $height = $2;
            #determine the character width of a line
            $width = hex $width; # convert length and width to decimal
            $height = hex $height;
            # each line of an image will be 0 padded if the number
            # of characters in an image line is not word aligned.
            # Calculate how many 0's will be padded to a line and
            # store result in $pad_value
            if ((($width*3)%4) != 0) {
                $pad_value = 2*(4-(($width*3)%4));
            } else {
                $pad_value = 0;
            }
            # the total number of characters in a line is therfore ...
            $line_character_width = ($width*6) + $pad_value;
        # the data record begins in line 4
        } elsif ($. == 4) {
```

```
    # data begins in column 4 of row 4
    /\w{4}\s\w{4}\s\w{4}\s(\w{2})(\w{2})\s(\w{2})(\w{2})\s(\w{2})
                        (\w{2})\s(\w{2})(\w{2})\s(\w{2})(\w{2})\s/;
    # reverse the endianness of the columns
    $line_reformat = $2 . $1 . $4 . $3 . $6 . $5 . $8 . $7 . $10 . $9;
    if (length $line_reformat <= $line_character_width) {
        @bmp_array[0] = $line_reformat;
        $residue="";
        $array_index = 0;
    } else {
        @bmp_array[0] = substr($line_reformat,0,$line_character_width);
        $residue = substr($line_reformat,$line_character_width);
        $array_index = 1;
    }
} elsif ($. > 4) {
    $line_length = length $_;
    # this takes care of the last line of the od which may have
    # less than 8 columns
    # 4 data bits + 1 space make up a column
    $line_reformat = "";
    for ($i=0;$i<$line_length/5;$i++) {
        $sub_string = substr($_,$i*5,5);
        $sub_string =~ /(\w{2})(\w{2})\s/;
        $line_reformat = $line_reformat . $2 . $1;
                # reverse endianess of each column of data
    }

    $line_reformat_length = length $line_reformat;
    $current_array_length = length @bmp_array[$array_index];
    $current_residue_length = length $residue;
    if ($current_array_length + $current_residue_length <=
                                    $line_character_width) {
        @bmp_array[$array_index] .= $residue;
        if ($current_array_length + $current_residue_length
                + $line_reformat_length <= $line_character_width) {
            @bmp_array[$array_index] .= $line_reformat;
            $residue = "";
            unless (length @bmp_array[$array_index] <
                                    $line_character_width) {
                $array_index++;
            }
        } else {
            $grab_char = $line_character_width
                                - $current_array_length
                                - $current_residue_length;
            @bmp_array[$array_index] .=
                     substr($line_reformat,0,$grab_char);
            $residue = substr($line_reformat,$grab_char);
            $array_index++;
        }
    } else {
        $grab_char = $line_character_width - $current_array_length;
        @bmp_array[$array_index] .= substr($residue,0,$grab_char);
        $fox = substr($residue,0,$grab_char);
        $residue = substr($residue,$grap_char);
```

```
                                        # change from grab_char + 1
            $residue .= $line_reformat;
            $array_index++;
            while (length $residue >= $line_character_width) {
                $grab_char = $line_character_width - $current_array_length;
                @bmp_array[$array_index] .= substr($residue,0,$grab_char);
                $residue = substr($residue,$grab_char);
                $array_index++;
            }
        }
    }
} # end while(<READ_FILE>)

if ($index==0) {
    # check to see if a pallete is being used
    $image_name = $file_list[$index];
    if ($image_name =~ /pallete/) {
        open (PALLETE, ">pallete_data.hex") or
                        die "$0: Can't create pallete_data.hex: $!";
        # read the pallete values into an array
        for ($pallete_arr_index=0;$pallete_arr_index<$width;
                            $pallete_arr_index++) {
            $pixel = substr(@bmp_array[0],
                            $pallete_arr_index*6,$pallete_arr_index*6+6);
            $pixel =~ /(\w{2})(\w{2})(\w{2})/;
            $pixel = $3 . $2 . $1;
            # write these values out to the pallete_data.hex file
            $to_pallete_file = hex_line($pixel,$pallete_arr_index,3);
            print PALLETE "$to_pallete_file\n";
            @pallete_array[$pallete_arr_index] = $pixel;
        }
        print PALLETE ":00000001ff\n";
        close PALLETE;
        $pallete_used = 1;
    }
}

unless ($pallete_used == 1 and $index == 0) {
    # write the base address and dimensions of the current image to the
    # C header file
    $image_name = $file_list[$index];
    $image_name =~ s/\.txt//i;
    $image_name = uc $image_name;
    print HEADER "#define $image_name\_BASE $hex_address\n";
    print HEADER "#define $image_name\_WIDTH $width\n";
    print HEADER "#define $image_name\_HEIGHT $height\n";
    $num_pixels = $width*$height;
    print HEADER "#define $image_name\_PIXELS $num_pixels\n\n";

    # At this point bmp_array contains the RBG values of the bit-map
    # array index 0 represents the last line of the image.  The lines
    # in the array are 0-padded according to $pad_value
    # loop through the @bmp_array and write into the hex file.
    for ($line_index=$height-1;$line_index >= 0; $line_index--) {
        for ($pixel_index=0;$pixel_index < $width; $pixel_index=$pixel_index+1) {
```

```
                #pull of 6 characters at a time from the array and reoder the pixels
                #to be in RGB format instead of BGR format
                $pixel = substr(@bmp_array[$line_index],$pixel_index*6,6);
                $pixel =~ /(\w{2})(\w{2})(\w{2})/;
                $pixel = color_compress($3,$2,$1);
                        # uncomment this if output is to be 16-bit

                # palletize the pixel if necessary
                $hex_width = 1; # set to 3 for 24-bit output
                if ($pallete_used == 1) {
                    # get the best match for the current pixel from the
                    # pallete array this is probably not the best
                    # palletizing algorithm.  I may use a
                    # different one at some point.
                    $selection_index = 0;
                    $diff = 0xFFFFF;
                    for ($pallete_looper=0;
                            $pallete_looper<scalar @pallete_array;
                                $pallete_looper++) {
                        $dec_pallete = hex @pallete_array[$pallete_looper];
                        $dec_pixel = hex $pixel;
                        if (abs($dec_pallete - $dec_pixel)< hex $diff) {
                            $selection_index = $pallete_looper;
                            $diff =  abs($dec_pallete - $dec_pixel);
                        }
                    }
                    $pixel = $selection_index;
                    if (scalar @pallete_array > 255) {
                        $hex_width = 2;
                    } else {
                        $hex_width = 1;
                    }
                }
                #print "hex width $hex_width\n";
                #print "$pixel\n";
                $line_address = $hex_address % 65536;
                if ($line_address == 0) {
                    $to_hex_file = hex_address_line($bit);
                    print "writing data\n";
                    print HEX_OUT "$to_hex_file\n";
                    $bit = $bit + 1;

                }
                $to_hex_file = hex_line($pixel,$line_address,$hex_width);
                print HEX_OUT "$to_hex_file\n";

                $hex_address = $hex_address + 1;
            }
        }
    } # end unless
    # delete the temporary files created by object dump
    `rm $file_list[$index]`;
} # end main for

print HEX_OUT ":00000001ff\n";
```

```
close HEX_OUT;
close HEADER;
```

DAMP Gamepack file listings

sbi_data.s

```
;*******************************************************************************
;               Filename: sbi_data.s
;
; This file is used to bring the different Slave Binary Image(SBI) that contain the
; FPGA configuration data. The start location of the SBI data for each image can be
; access via the lables therefore they are exported.
;
;*******************************************************************************

    EXPORT start_1_sbi
    EXPORT end_1_sbi
    EXPORT start_2_sbi
    EXPORT end_2_sbi
    EXPORT start_3_sbi
    EXPORT end_3_sbi
    EXPORT start_4_sbi
    EXPORT end_4_sbi
    EXPORT start_5_sbi
    EXPORT end_5_sbi
    AREA sbi_data, DATA, READONLY

start_1_sbi

    INCBIN sbi/minesweeper.sbi

end_1_sbi


start_2_sbi

    INCBIN sbi/pong01.sbi

end_2_sbi


start_3_sbi

    INCBIN sbi/snakes.sbi

end_3_sbi

start_4_sbi

    INCBIN sbi/rivers.sbi
```

```
end_4_sbi


start_5_sbi

    INCBIN sbi/mips.sbi

end_5_sbi

    END
```

germs_monitor.c

```
#include "excalibur.h"
#include <string.h>
#include <stdio.h>
#include "errors.h"
#include "config_logic.h"


// +-----------------------------------------------
// | Various constants to control the monitor
// |



#ifndef GM_FlashBase
 #ifdef nasys_main_flash
  #define GM_FlashBase nasys_main_flash
 #endif
#endif

#ifdef GM_FlashBase

 #ifndef GM_FlashTop
  #ifdef nasys_main_flash_end
   #define GM_FlashTop nasys_main_flash_end
  #else
   #define GM_FlashTop (GM_FlashBase+0x100000)
  #endif
 #endif

 #ifndef GM_FlashExec
  #define GM_FlashExec (GM_FlashBase+0x4000C)
 #endif

#endif

#define k_line_buffer_length 256
#define k_default_show_range 64
#define k_words_per_line 4
#define k_word_size 4
#define k_input_string_length 512
```

```
// +--------------------------
// | Handful of evil globals

typedef struct
{
    char b[k_line_buffer_length]; // general purpose buffer
    unsigned long memory_position; // location to display next
    unsigned long memory_range; // how much to show on next <return>
    unsigned long relocation_offset;
    unsigned long ihex_upper_address; // upper 16 bits of address for ihex (':') fmt data
    unsigned long checksum; // for S-records
} globals;

static globals g = {0};

// +--------------------------------
// | Local Prototypes

static int r_hex_char_to_value(char c);
static void r_show_range(unsigned int addr_lo,unsigned int addr_hi);
#ifdef GM_FlashBase
 static int r_in_flash_range(unsigned int addr);
#endif
static unsigned int r_fetch_next_hex_byte(char **wp);
static int r_stash_byte(unsigned int addr,unsigned char value);

static void r_scan_string_for_addr_range
        (
        char **wp,
        char *break_char_out,
        unsigned int *addr_lo_out,
        unsigned int *addr_hi_out
        );

static unsigned int r_scan_string_for_number(char **wp, char *break_char_out);

// Slight abstraction layer of communication port

int r_send_string(char *s);
int r_send_char(int c);
int r_get_line(char *s,int maxlen);
int r_get_char(void);

// Various GERMS commands

int do_1_command(void);
int do_2_command(void);
int do_3_command(void);
int do_4_command(void);
int do_5_command(void);
int do_g_command(char *command);
int do_e_command(char *command);
int do_r_command(char *command);
int do_m_command(char *command);
int do_s_command(char *command);
```

```
int do_colon_command(char *command);
int do_show_info(void);
int do_help(void);




// +--------------------------------
// | Local Little Routines

typedef void (*r_dull_proc_ptr)(void);

void r_goto(unsigned int addr)
    {
    r_dull_proc_ptr p = (r_dull_proc_ptr)addr;

    (*p)();
    }


// |
// | send and get character routines
// | uses default printf stuff
// |

int r_send_char(int c)
{
    nr_txchar(c);
    return 0;
}

int r_get_char(void)
{
    return nr_rxchar();
}

int r_send_string(char *string)
{
    // Just to be pedantic, call sendchar over and over
    while(*string)
        r_send_char(*string++);
    return 0;
}

// |
// | Accumulate a line of text from the
// | input source (call r_get_char),
// | and return the length. ESC means abort, and
// | return length of -1.
// |

int r_get_line(char *string_out,int maxlen)
{
    int len = 0;
    int done = 0;
```

```
    int c;

    while(!done)
    {
        // Wait for a character
        while((c = r_get_char()) < 0)
            ;

        // Echo it
        r_send_char(c);

        // See if it is special
        if(c == 27) // ESC escape
        {
            len = -1; // error return: length of -1
            done = 1;
        }
        else if(c == 8) // Backspace ^H
        {
            if(len)
                len--;
        }
        else if(c == 13) // carriage return (end of line)
        {
            done = 1;
        }
        else if(c >= 0x20 && c <= 0x7e) // regular character
        {
            if(len < maxlen)
                string_out[len++] = c;
        }
    }

    // done: set the string end
    // (And, as a courtesy, zero out the rest
    // of the string))
    if(len >= 0)
    {
        int i;
        for(i = len; i <= maxlen; i++)
            string_out[i] = 0;
    }

    return len;
}


static int r_mask_short(int x)
    {
        return x & 0x0000ffff;
    }

#ifdef GM_FlashBase

static int r_in_flash_range(unsigned int addr)
```

```
    {
    int result = 0;

#ifdef GM_FlashBase
    result = (addr >= (unsigned int)GM_FlashBase) && (addr < (unsigned int)GM_FlashTop);
#endif

    return result;
    }

#endif

// | Write a byte to memory. Maybe do some special things for
// | special ranges, like flash.

static int r_stash_byte(unsigned int addr,unsigned char value)
    {
    volatile unsigned char *a;
    int result = 0;

#ifdef GM_FlashBase
    // | Are we writing to flash range?
    // | We do that by stashing even bytes and writing on odd bytes.
    if(r_in_flash_range(addr));
        {
        if((addr & 1) == 0)
            g.even_byte = value;
        else
            {
            us halfword;

            halfword = g.even_byte + (value << 8);
            nr_flash_write(GM_FlashBase,(us *)(addr & ~1), halfword);
            }
        goto go_home;
        }
#endif

    a = (unsigned char *)addr;
    *a = value;

    if(*a != value)
        result = -1;

    // |
    // | print a bang if it didn't write
    // |

    if(result)
        r_send_string("!");
#ifdef GM_FlashBase
go_home:
#endif
    return result;
    }
```

```
static void r_show_range(unsigned int addr_lo,unsigned int addr_hi)
    {
    g.memory_range = addr_hi - addr_lo;
    g.memory_position = addr_hi;

    addr_hi &= ~1;
    addr_lo &= ~1;

    while( (((int)addr_hi) - ((int)addr_lo)) > 0 )
        {
        int word_count;
        int i;

        sprintf(g.b,addr_lo <= 0xffff ? "# %04X:" : "# %08X:",addr_lo);
        word_count = (addr_hi - addr_lo) / k_word_size;
        if(word_count > k_words_per_line)
            word_count = k_words_per_line;

        for(i = 0; i < word_count; i++)
        {
            unsigned int v = *(unsigned long *)(addr_lo + i * k_word_size);

            // |
            // | Some processors (like the ARM) can actually
            // | allow v to now contain a value outside the
            // | unsigned integer range!
            // |
            // | How can this be? Well, if there is an address
            // | exception, and the handler is unsophisticated,
            // | it seems to be possible.
            // |

            //v = r_mask_short(v);

            // | there.

            sprintf(g.b + strlen(g.b)," %08X",v);
        }

        for(i = word_count; i < k_words_per_line; i++)
            sprintf(g.b + strlen(g.b),"         ");

        sprintf(g.b + strlen(g.b)," # ");

        for(i = 0; i < word_count * k_words_per_line; i++)
            {
            char c;

            c = *(char *)(addr_lo + i);
            if(c < 0x20 || c > 0x7e)
                c = '.';
            sprintf(g.b + strlen(g.b),"%c",c);
            }
```

```
        sprintf(g.b + strlen(g.b),"\n");
        r_send_string(g.b);

        addr_lo += word_count * k_word_size;
        }
    }


int do_germs_command(char *command)
{
    int result = 0;
    char command_letter;

    if(command[0] == 0) // no command letter?
    {
        // | null command means show more memory
        // |

        if(g.memory_range == 0)
            g.memory_range = k_default_show_range;
        r_show_range(g.memory_position,g.memory_position + g.memory_range);

        goto go_home;
    }

    command_letter = command[0];
    if(command_letter >= 'A' && command_letter <= 'Z')
        command_letter += 'a' - 'A';

    // | Dont pass the command letter on.

    command++;

    switch(command_letter)
    {
        case '1':
            do_1_command();
            break;

        case '2':
            do_2_command();
            break;

        case '3':
            do_3_command();
            break;

        case '4':
            do_4_command();
            break;

        case '5':
            do_5_command();
            break;
```

```
        case 'g':
            do_g_command(command);
            break;

        case 'e':
            do_e_command(command);
            break;

        case 'r':
            do_r_command(command);
            break;

        case 'm':
            do_m_command(command);
            break;

        case 's':
            do_s_command(command);
            break;

        case ':':
            do_colon_command(command);
            break;

        case '#':
            // comment character: do nothing
            break;

        // | c_germs extensions

        case 'v':
            do_show_info();
            break;

        case '?':
        case 'h':
            do_help();
            break;

        default:
        {
            r_send_string("(unknown command; try h for help)\n");
        }
    }

go_home:
    return result;
}

// return 1 if is break character

int is_break_char(char x)
    {
    int xx = x;
    if
```

```
        (
            (xx <= 0)
        ||
            (xx >= 0x20 && xx < 0x2f)
        ||
            (xx >= 0x3a && xx < 0x3e)
        ||
            (xx > 0x7f)
        )
        return 1;

    return 0;
    }

static int r_hex_char_to_value(char c)
    {
    int value;

    value = c & 0x000f;
    if(c > '9')
        value += 9;

    return value;
    }

// +-----------------------------------------
// | read characters forward until either end of string,
// | or a non-alphanumeric-non-blank char. Return that
// | as the break character.

static unsigned int r_scan_string_for_number(char **wp, char *break_char_out)
    {
    char *w;
    char c;
    unsigned int value = 0;

    w = *wp;

    // | skip past any leading blanks for the number

    while(*w == ' ')
        w++;

    while(!is_break_char(c = *w++))
        {
        if(c != ' ')
            {
            value = value * 16 + r_hex_char_to_value(c);
            }
        }

    *wp = w;
    *break_char_out = c;
    return value;
    }
```

```
// |
// | Get either a single address or a hyphen-delimited address range,
// | returned as addr_lo & addr_hi. If they match, only one was
// | entered (or user typed 300-300 for a range, which we pretend
// | is just 300).
// |
static void r_scan_string_for_addr_range(char **wp,char *break_char_out,
                                    unsigned int *addr_lo_out,unsigned int *addr_hi_out)
    {
    char break_char;
    unsigned int addr_lo = 0;
    unsigned int addr_hi = 0;

    addr_lo = r_scan_string_for_number(wp,&break_char);
    if(break_char == '-')
        addr_hi = r_scan_string_for_number(wp,&break_char);
    else
        addr_hi = addr_lo;

    *break_char_out = break_char;
    *addr_lo_out = addr_lo;
    *addr_hi_out = addr_hi;
    }


int do_g_command(char *command)
{
    unsigned int addr;
    char break_char;
    addr = r_scan_string_for_number(&command,&break_char);
    sprintf(g.b,"# executing at %08x\n",addr);
    r_send_string(g.b);
    r_goto(addr);

    return 0;
}

int do_e_command(char *command)
{
    unsigned int addr;
    char break_char;

    addr = r_scan_string_for_number(&command,&break_char);
    sprintf(g.b,"# erasing: %08x\n",addr);
    r_send_string(g.b);

#ifdef GM_FlashBase
    if(r_in_flash_range(addr))
        nr_flash_erase_sector(GM_FlashBase,addr);
    else
#endif
    r_send_string("not flash\n");
```

```
    g.memory_position = addr; // (so a <return> shows what we just erased
    return 0;
    }

int do_r_command(char *command)
    {
    unsigned int addr_lo;
    unsigned int addr_hi;
    char break_char;

    r_scan_string_for_addr_range(&command,&break_char,&addr_lo,&addr_hi);
    if(addr_lo == addr_hi)
        addr_lo = 0;

    g.relocation_offset = addr_hi - addr_lo;

    sprintf(g.b,"# relocation offset: %08X\n",(int)g.relocation_offset);
    r_send_string(g.b);

    return 0;
    }

int do_m_command(char *command)
    {
    char break_char = 1;
    unsigned int addr_lo;
    unsigned int addr_hi;
    unsigned int v;
    int result = 0;

    r_scan_string_for_addr_range(&command,&break_char,&addr_lo,&addr_hi);

    if(break_char != ':') // anything other than colon after address means "show"
        {

        // | Range given? if not, use default range

        if(addr_lo == addr_hi)
            {
            addr_lo = addr_hi;
            addr_hi += k_default_show_range;
            }

        r_show_range(addr_lo,addr_hi);
        goto go_home;
        }

    // | We've gotten either 1 or 2 addresses and a colon. If 1 address,
    // | we're writing words in memory. If 2 addresses, we're filling a
    // | range of memory with a value.

    // | range fill?
    if(addr_lo != addr_hi)
        {
        v = r_scan_string_for_number(&command,&break_char);
```

```
        g.memory_position = addr_lo;

        while(addr_lo < addr_hi)
            {

            r_stash_byte(addr_lo,v & 0xff);
            r_stash_byte(addr_lo+1,(v >> 8) & 0xff);

            addr_lo += 2;
            }
        goto go_home;
        }

    // | word-by-word

    addr_lo = addr_hi;
    g.memory_position = addr_lo;
    while(break_char != 0)
        {
        v = r_scan_string_for_number(&command,&break_char);

        r_stash_byte(addr_lo,v & 0xff);
        r_stash_byte(addr_lo+1,(v >> 8) & 0xff);

        addr_lo += 2;
        }

go_home:
    return result;
    }


// +------------------------------
// | Read a byte from the string pointerpointer wp
// | in hex. Also, add the byte into g.checksum.
// |
static unsigned int r_fetch_next_hex_byte(char **wp)
    {
    char *w = *wp;
    unsigned int value = 0;

    if(*w != 0)
        value = r_hex_char_to_value(*w++);
    if(*w != 0)
        value = value * 16 + r_hex_char_to_value(*w++);

    *wp = w;
    g.checksum += value;

    return value;
    }

static unsigned int r_fetch_multibyte(char **wp,int byte_count)
    {
```

```
    unsigned int value = 0;

    while(byte_count--)
        value = value * 256 + r_fetch_next_hex_byte(wp); // big endian

    return value;
    }




int do_1_command(void)
    {
        extern unsigned int start_1_sbi;
        int returnCode = 0;

        returnCode = configure_pld((unsigned int *)&start_1_sbi);
        return 0;
    }

int do_2_command(void)
    {
        extern unsigned int start_2_sbi;
        int returnCode = 0;

        returnCode = configure_pld((unsigned int *)&start_2_sbi);
        return 0;
    }

int do_3_command(void)
    {
        extern unsigned int start_3_sbi;
        int returnCode = 0;

        returnCode = configure_pld((unsigned int *)&start_3_sbi);
        return 0;
    }

int do_4_command(void)
    {
        extern unsigned int start_4_sbi;
        int returnCode = 0;

        returnCode = configure_pld((unsigned int *)&start_4_sbi);
        return 0;
    }

int do_5_command(void)
    {
        extern unsigned int start_5_sbi;
        int returnCode = 0;

        returnCode = configure_pld((unsigned int *)&start_5_sbi);
        return 0;
    }
```

```
// +-----------------------------------------
// | S-Record is
// | S<type><length><address><data...><checksum>
// |
// | Type is 0 -- starting record
// |       1-3 -- 16/24/32 bit address data record
// |       4-6 -- misc symbol records we ignore
// |       7-9 -- 32/24/16 bit ending record (Germs treats as GO record)

int do_s_command(char *command)
    {
    char record_type;
    unsigned int record_length;
    unsigned int record_address;
    unsigned int record_address_width; // in bytes: 2, 3, or 4
    int i;

    g.checksum = 0; // reset the checksum

    if(*command)
        record_type = *command++;

    record_length = r_fetch_next_hex_byte(&command);

    switch (record_type)
        {
        case '1': // 16-bit-address data record
            record_address_width = 2;
read_s_record_data:
            record_address = r_fetch_multibyte(&command,record_address_width);
            record_length -= (record_address_width + 1); // no count addr or checksum
            for(i = 0; i < record_length; i++)
                {
                unsigned int value;

                value = r_fetch_next_hex_byte(&command);
                r_stash_byte(record_address + i + g.relocation_offset,value);
                }

            // lastly, checksum on S-Record. Should set g.checksum to 0xff.
            r_fetch_next_hex_byte(&command);
            if(g.checksum != 0xff)
                r_send_string("^"); // checksum error caret
            break;

        case '2': // 24-bit address data record
            record_address_width = 3;
            goto read_s_record_data;

        case '3': // 32-bit address data record
            record_address_width = 4;
            goto read_s_record_data;

        case '7': // 32 bit start-address record
```

```
            record_address_width = 4;
read_s_record_start_address:
            record_address = r_fetch_multibyte(&command,record_address_width);
            // checksum
            r_fetch_next_hex_byte(&command);
            if(g.checksum != 0xff)
                r_send_string("^"); // checksum error caret
            else
                r_goto(record_address);
            break;

        case '8': // 24 bit start-address record
            record_address_width = 3;
            goto read_s_record_start_address;

        case '9': // 16 bit start-address record
            record_address_width = 2;
            goto read_s_record_start_address;
        } /* switch */
    return 0;
    }

int do_colon_command(char *command)
    {
    unsigned int record_type;
    unsigned int data_bytes_count;
    unsigned int record_address;
    int i;

    g.checksum = 0; // reset the checksum

    data_bytes_count = r_fetch_next_hex_byte(&command);

    record_address = r_fetch_multibyte(&command,2);
    record_type = r_fetch_next_hex_byte(&command);

    switch(record_type)
        {
        case 0: // normal data read
            record_address += g.ihex_upper_address;


            for(i = 0; i < data_bytes_count; i++)
                {
                unsigned int value;

                value = r_fetch_next_hex_byte(&command);
                r_stash_byte(record_address + i + g.relocation_offset,value);
                }

            // lastly, checksum on I-Hex Record. Should set g.checksum to 0x00.
            r_fetch_next_hex_byte(&command);
            if(g.checksum != 0xff)
                r_send_string("^"); // checksum error caret
            break;
```

```
        case 4: // upper address bits record:
            g.ihex_upper_address = r_fetch_multibyte(&command,2) << 16;
            break;

        } /* switch */

    return 0;
    }

int do_show_info(void)
    {
    r_send_string("\n\n");
    r_send_string("Welcome to DAMP Monitor\n");
    sprintf(g.b,"CPU Architecture: %s\n",nm_cpu_architecture_string);
    r_send_string(g.b);

    sprintf(g.b,"System Name: %s\n",nm_system_name_string);
    r_send_string(g.b);

    sprintf(g.b,"System ID: %s\n",nm_monitor_string);
    r_send_string(g.b);

    r_send_string("\n\n");

    return 0;
    }

int do_help(void)
    {
    r_send_string("\n");
    r_send_string("# The following commands are available:\n");
    r_send_string("# <SBI number>              -- programs sbi-file\n");
    r_send_string("# v                         -- initialize video_system\n");
    r_send_string("# g <addr>                  -- execute at addr\n");
    r_send_string("# e <addr>                  -- erase flash memory at addr\n");
    r_send_string("# r <addr1>-<addr2>         -- relocate S-records from addr1 to addr2\n");
    r_send_string("# m <addr>                  -- show memory at addr\n");
    r_send_string("# m <addr1>-<addr2          -- show memory range from addr1 to addr2\n");
    r_send_string("# m <addr>:<val1> <val2>... -- write halfwords at addr\n");
    r_send_string("# m <addr1>-<addr2>:<val>   -- fill range with halfword val\n");
    r_send_string("# s<S-record data>          -- write S-record to memory\n");
    r_send_string("# :<Intel hex data>         -- write Intel hex record to memory\n");
    r_send_string("# v                         -- show processor version\n");
    r_send_string("# help                      -- show this help\n");
    r_send_string("\n");
    return 0;
    }

int main(void)
    {
    int result;
    char command[k_line_buffer_length];

    do_show_info();
```

```
    while(1)
        {
        r_send_string("+");

        result = r_get_line(command,k_line_buffer_length);
        if(result < 0)
            do_show_info();
        else
            result = do_germs_command(command);
        }
    }

// end of file
```

config_logic.c

```
#include <stdio.h>
#include "ARM_Stripe.h"
#include "config_logic.h"
#include "errors.h"
#include "timer.h"
#define CONFIG_TIMEOUT (0x1000000) //Number of clocks on AHB2 before configuration  timeout


int configure_pld(unsigned int *sbi_base_address)
{
    int return_code = OK;
    unsigned int *sbi_data_base, *sbi_data_end;
    unsigned int *register_base = ( unsigned int *)EXC_REGISTERS_BASE;

    sbi_data_base = (*(sbi_base_address + 2))/4 + sbi_base_address;
        //The coffset is located at the second location after the base of the file
        //The coffset give the offset in bytes.  Had to divide by for for word alignment
        //The coffset plus the base address of the chosen EBI will give the absolute address
        //Where the SBI data starts.
    sbi_data_end = sbi_data_base + (*(sbi_base_address + 3))/4;
        //The size of the SBI data is given at the third location after the base of the file
        //Set a pointer to the end of the file
    *CONFIG_UNLOCK(EXC_PLD_CONFIG00_BASE) = MAGIC_UNLOCK_VALUE;
        //Unlock the configuration logic

        //Check to make sure that the configuration logic is unlocked
    if(*CONFIG_CONTROL(EXC_PLD_CONFIG00_BASE) & CONFIG_CONTROL_LK_MSK)
    {
        return_code = ERROR_PLD_CONFIG_LOCKED;
        return return_code;
    }
    *CONFIG_CLOCK(EXC_PLD_CONFIG00_BASE) = DIVISOR_FOR_CONFIG_CLOCK(
                                    CONFIG_CLOCK_SPEED,EXC_AHB2_CLK_FREQUENCY) + 1;
                                     //Had to add 1 to this because of rounding error
                                     //I should go back and make this more robust and so that
                                     //I don't have to add 1.  Need to look into a modulus function
    if(*(sbi_base_address + 1) != *(register_base + 2))  //Check for the device IDCODE
    {
```

```
        return_code = ERROR_INVALID_PLD_IDCODE;
        return return_code;
    }

    *CONFIG_CONTROL(EXC_PLD_CONFIG00_BASE) = CONFIG_CONTROL_CO_MSK ;
    //Set the CO bit in the Configuration Control Register
    //This enables the Configuration
    while(*CONFIG_CONTROL(EXC_PLD_CONFIG00_BASE) & CONFIG_CONTROL_B_MSK);
    //After the CO bit is set, the Busy bit is automatically set
    //Need to wait for it to be cleared before we move on
    return_code = start_timer();        //start timer to check for configuration timeout.
    if(return_code != OK)
      return return_code;
    while(sbi_data_base <= sbi_data_end)        //Configure PLD
    {
        *CONFIG_DATA(EXC_PLD_CONFIG00_BASE) = *sbi_data_base;

            //Check the busy bit
        while(*CONFIG_CONTROL(EXC_PLD_CONFIG00_BASE) & CONFIG_CONTROL_B_MSK);
        sbi_data_base++;                    //Increment pointer to EBI information
        if(read_timer()> CONFIG_TIMEOUT)            //check for time out
          {
            return_code = ERROR_PLD_CONFIG_TIMEOUT;
            return return_code;
          }
    }
           //Wait until configuration is complete before moving on.
    while(*CONFIG_CONTROL(EXC_PLD_CONFIG00_BASE) & CONFIG_CONTROL_CO_MSK)
      {
        if(read_timer()> CONFIG_TIMEOUT)                //check for timeout
          {
            return_code = ERROR_PLD_CONFIG_TIMEOUT;
            return return_code;
          }
      }

     return_code = stop_timer();                    //stop timer


return return_code;
}
```

config_logic.h

```
#ifndef __CONFIG_LOGIC_H
#define __CONFIG_LOGIC_H

/*
 * Register definitions for the UART
 */
int configure_pld(unsigned int *sbi_base_address);

/*
 * Copyright (c) Altera Corporation 2002.
```

```
 * All rights reserved.
 */

#define CONFIG_CLOCK_SPEED 16000000 //Want to configure as fast as possible

#define CONFIG_CONTROL(base_addr) (CONFIG_LOGIC_TYPE (base_addr  + 0x00 ))
#define CONFIG_CONTROL_LK_MSK (0x01)
#define CONFIG_CONTROL_LK_OFST (0)
#define CONFIG_CONTROL_CO_MSK (0x02)
#define CONFIG_CONTROL_CO_OFST (1)
#define CONFIG_CONTROL_B_MSK (0x04)
#define CONFIG_CONTROL_B_OFST (2)
#define CONFIG_CONTROL_PC_MSK (0x08)
#define CONFIG_CONTROL_PC_OFST (3)
#define CONFIG_CONTROL_E_MSK (0x10)
#define CONFIG_CONTROL_E_OFST (4)
#define CONFIG_CONTROL_ES_MSK  (0xE0)
#define CONFIG_CONTROL_ES_OFST (5)


#define CONFIG_CLOCK(base_addr) (CONFIG_LOGIC_TYPE (base_addr  + 0x04 ))
#define CONFIG_CLOCK_MSK (0xFFFF)
#define CONFIG_CLOCK_OFST (0)


#define CONFIG_DATA(base_addr) (CONFIG_LOGIC_TYPE (base_addr  + 0x08 ))
#define CONFIG_DATA_MSK (0xFFFFFFFF)
#define CONFIGU_DATA_OFST (0)

#define CONFIG_UNLOCK(base_addr) (CONFIG_LOGIC_TYPE (base_addr  + 0x0C ))
#define CONFIG_UNLOCK_MSK (0xFFFFFFFF)
#define CONFIGU_UNLOCK_OFST (0)
#define MAGIC_UNLOCK_VALUE 0x554E4C4B

#define CONFIG_LOGIC_TYPE (volatile unsigned int*)
#define DIVISOR_FOR_CONFIG_CLOCK(fout,ahb2_clk) ((ahb2_clk) /(fout << 1))
                //Come back and revisit this.  I think this should be 1



#endif /* __CONFIG_LOGIC_H */
```

erros.c

```
/*****************************************************************************
;
;               Filename: errors.c
;
;This code prints out different error messages that apply to different conditions
;that could take place while running this application.  It serves as just a place holder
;for where some kind of corrective action could take place to fix the problems that are
;reported
;
;*****************************************************************************/
```

```
#include <stdio.h>
#include "errors.h"


void print_error_message(int error_code)
{
  printf("\r\n***************Error*****************\n");
  switch(error_code)
    {
    case ERROR_PLD_CONFIG_LOCKED:
      printf("\rPLD Configuration Failure:Unable to unlock configuration logic\n");
      break;
    case ERROR_INVALID_PLD_IDCODE:
      printf("\rPLD Configuration Failure:Invalid Device ID(IDCODE)\n");
      printf("\rSBI data maybe invalid\n");
      break;
    case ERROR_PLD_CONFIG_TIMEOUT:
      printf("\rPLD Configuration Failure:Configuration timeout\n");
      break;
    case ERROR_TIMER_OVERFLOW :
      printf("\rTimer Failure: The requested delay is to large\n");
      printf("\rPlease enter a smaller value requested in delay funtion call\n");
      break;
     case ERROR_SUBTRACTION :
      printf("\rALU Failure: ALU only supports unsigned operations\n");
      printf("\rOperand2 must be smaller than Operand1 for Subtraction operation\n");
      break;
      case ERROR_TIMER_STARTED :
      printf("\rTimer Failure: The timer is running and you attempted to start it.\n");
      printf("\rTimer should be stopped before it is started again\n");
      break;
      case ERROR_TIMER_STOPPED :
      printf("\rTimer Failure: The timer is stopped and you attempted to stop it.\n");
      printf("\rTimer should be started before you stop it\n");
      break;
    }
  printf("\r\n**************System Must be Reset*************\n");
}
```

erros.h

```
#define OK              (0)
#define ERROR_PLD_CONFIG_LOCKED    (1)
#define ERROR_INVALID_PLD_IDCODE   (2)
#define ERROR_PLD_CONFIG_TIMEOUT   (3)
#define ERROR_TIMER_OVERFLOW       (4)
#define ERROR_SUBTRACTION          (5)
#define ERROR_TIMER_STARTED    (6)
#define ERROR_TIMER_STOPPED    (7)
void print_error_message(int error_code);
```

timer.c

```
/*****************************************************************************
```

```
#include <stdio.h>
#include "ARM_Stripe.h"
#include "timer00.h"
#include "timer.h"
#include "errors.h"

int delay(unsigned int milliseconds)
{
  long long  delay_in_clocks;
  unsigned int timer_value;
  int return_code = OK;

  delay_in_clocks = ((EXC_AHB2_CLK_FREQUENCY)/1000) * milliseconds;
  if(delay_in_clocks > TIMER_LIMIT)
  {
      return_code = ERROR_TIMER_OVERFLOW;
      return return_code;
  }
  return_code = start_timer();
  do
  {
    timer_value = read_timer();
  }while(timer_value < delay_in_clocks);

  return_code = stop_timer();
  return return_code;

}

int start_timer(void)
{
  int return_code = OK;
  if(*TIMER0_SR(EXC_TIMER00_BASE) && TIMER0_CR_S_MSK)
  {   return_code = ERROR_TIMER_STARTED;
                //timer should be stopped before attempting to start

  }
  else
    {
      *TIMER0_LIMIT(EXC_TIMER00_BASE)=TIMER_LIMIT;
      *TIMER0_CR(EXC_TIMER00_BASE)= TIMER0_CR_S_MSK | TIMER0_SR_MODE_FREE;
    }

  return return_code;
}

unsigned int read_timer(void)
{
  unsigned int value;
  value = *TIMER0_READ(EXC_TIMER00_BASE);
  return value;
}

int stop_timer(void)
{
```

```
  int return_code = OK;


  if(!(*TIMER0_SR(EXC_TIMER00_BASE) && TIMER0_CR_S_MSK))
     return_code = ERROR_TIMER_STOPPED;
                              //Timer should be started before attempting to stop it
                              //Attempting to stop the timer should not have an effect
                              //on the program, but the check is in place to make sure
                              //the correct sequences occurs.


  else
      *TIMER0_CR(EXC_TIMER00_BASE) = (*TIMER0_CR(EXC_TIMER00_BASE) & (~TIMER0_CR_S_MSK));


  return return_code;
}
```

timer.h

```
#ifndef TIMER_H
#define TIMER_H

#define TIMER_LIMIT (0xFFFFFFFF)

#define TIMER00_TYPE (volatile unsigned int*)
int delay(unsigned int milliseconds);
int start_timer(void);
int stop_timer(void);
unsigned int read_timer(void);
#endif /* TIMER_H */
```

timer00.h

```
#ifndef __TIMER00_H
#define __TIMER00_H

#define TIMER0_CR(base_addr) (TIMER00_TYPE (base_addr  + 0x00 ))
#define TIMER0_CR_MODE_MSK (0x3)
#define TIMER0_CR_MODE_OFST (0x0)
#define TIMER0_CR_MODE_FREE (0x0)
#define TIMER0_CR_MODE_ONE_SHOT (0x1)
#define TIMER0_CR_MODE_INTERVAL (0x2)
#define TIMER0_CR_IE_MSK  (0x4)
#define TIMER0_CR_IE_OFST (0x2)
#define TIMER0_CR_CI_MSK  (0x8)
#define TIMER0_CR_CI_OFST (0x3)
#define TIMER0_CR_S_MSK (0x10)
#define TIMER0_CR_S_OFST (0x4)

#define TIMER0_SR(base_addr) (TIMER00_TYPE (base_addr  + 0x00 ))
#define TIMER0_SR_MODE_MSK (0x3)
#define TIMER0_SR_MODE_OFST (0x0)
#define TIMER0_SR_MODE_FREE (0x0)
```

```
#define TIMER0_SR_MODE_ONE_SHOT (0x1)
#define TIMER0_SR_MODE_INTERVAL (0x2)
#define TIMER0_SR_IE_MSK  (0x4)
#define TIMER0_SR_IE_OFST (0x2)
#define TIMER0_SR_CI_MSK  (0x8)
#define TIMER0_SR_CI_OFST (0x3)
#define TIMER0_SR_S_MSK (0x10)
#define TIMER0_SR_S_OFST (0x4)

#define TIMER0_PRESCALE(base_addr) (TIMER00_TYPE (base_addr  + 0x10 ))
#define TIMER0_LIMIT(base_addr) (TIMER00_TYPE (base_addr  + 0x20 ))
#define TIMER0_READ(base_addr) (TIMER00_TYPE (base_addr  + 0x30 ))


#define TIMER1_CR(base_addr) (TIMER00_TYPE (base_addr  + 0x40 ))
#define TIMER1_CR_MODE_MSK (0x3)
#define TIMER1_CR_MODE_OFST (0x0)
#define TIMER1_CR_MODE_FREE (0x0)
#define TIMER1_CR_MODE_ONE_SHOT (0x1)
#define TIMER1_CR_MODE_INTERVAL (0x2)
#define TIMER1_CR_IE_MSK  (0x4)
#define TIMER1_CR_IE_OFST (0x2)
#define TIMER1_CR_CI_MSK  (0x8)
#define TIMER1_CR_CI_OFST (0x3)
#define TIMER1_CR_S_MSK (0x10)
#define TIMER1_CR_S_OFST (0x4)

#define TIMER1_SR(base_addr) (TIMER00_TYPE (base_addr  + 0x40 ))
#define TIMER1_SR_MODE_MSK (0x3)
#define TIMER1_SR_MODE_OFST (0x0)
#define TIMER1_SR_MODE_FREE (0x0)
#define TIMER1_SR_MODE_ONE_SHOT (0x1)
#define TIMER1_SR_MODE_INTERVAL (0x2)
#define TIMER1_SR_IE_MSK  (0x4)
#define TIMER1_SR_IE_OFST (0x2)
#define TIMER1_SR_CI_MSK  (0x8)
#define TIMER1_SR_CI_OFST (0x3)
#define TIMER1_SR_S_MSK (0x10)
#define TIMER1_SR_S_OFST (0x4)

#define TIMER1_PRESCALE(base_addr) (TIMER00_TYPE (base_addr  + 0x50 ))
#define TIMER1_LIMIT(base_addr) (TIMER00_TYPE (base_addr  + 0x60 ))
#define TIMER1_READ(base_addr) (TIMER00_TYPE (base_addr  + 0x70 ))



#endif /* __TIMER00_H */
```

Curriculum Vitae

W. Zwart was born in Gouda, the Netherlands, on September 23 1976. He attended the 'RSG De Drie Waarden' high school in Schoonhoven, from which he graduated in 1995. In the year 1996, he was admitted to the Electrical Engineering faculty of the Delft University of Technology in the Netherlands.

After receiving his Bachelor of Science degree, he joined the Computer Engineering laboratory, led by professor Stamatis Vassiliadis, to start his MSc graduation project under the supervision of Dr. Sorin Cotofana and Ir. Georgi Gaydadjiev. His thesis is titled **Testing and redesigning the Delft Altera-based Multimedia Platform**. His research interests include: computer architecture, embedded systems design and FPGA-based design.