TAMPERE UNIVERSITY OF TECHNOLOGY

Juha Arvio

Augmenting IP blocks for verification and optimization

Master of Science Thesis

Examiners: Dr. Tech. Erno Salminen
Prof. Timo D. Hämäläinen
Subject approved by Faculty of
Computing and Electrical Engineering
Council meeting on 04.04.2012

Tiivistelmä

TAMPEREEN TEKNILLINEN YLIOPISTO

Sähkötekniikan koulutusohjelma

Arvio, Juha: Augmenting IP blocks for verification and optimization

Pääaine: Digitaali- ja tietokonetekniikka

Diplomityö, 79 sivua

Tarkastajat: Tek. Toht. Erno Salminen, Prof. Timo D. Hämäläinen

Avainsanat: Metadata, verifiointi, optimointi

Helmikuu 2012

Digitaalisten IP-lohkojen verifiointi on aina ollut haasteellista. Modelsimin kaltaisilla softasimulaattoreilla voidaan verifioida aika perusteellisesti yksinkertaiset lohkot, joihin löytyy suoraviivaiset testisyötteet. Valitettavasti monimutkaisten järjestelmäpiirien simulointi softasimulaattorilla voi kestää päiviä ellei viikkoja. Lisäksi jokaiselle lohkolle, tarkasteltavassa järjestelmäpiirissä, on löydyttävä toimiva simulointimalli. Nykyaikaisia ohjelmoitavia FPGA-piirejä voidaan kylläkin monitoroida reaaliaikaisesti Alteran Signal Tap II:n tapaisilla työkaluilla, mutta tämän kaltaisilla työkaluilla voidaan monitoroida vain pieniä määriä signaaleja lyhyellä aikavälillä.

IP-informaatiorekisterit (IIR) luotiin ylittämään nämä esteet. Näitä rekistereitä käytetään tallentamaan tietoa jota saadaan IP-lohkoista sekä järjestelmäpiireistä kokonaisuutena. Tämä tieto voi olla joko staattista tai dynaamista, toisin sanoen se on luotu joko ennen järjestelmäpiirin käyttämistä oikealla alustalla tai sen aikana. Tätä tietoa voidaan käyttää moneen tarkoitukseen, kuten yksittäisten IP-lohkojen ja kokonaisten järjestelmäpiirien verifiointiin.

Tämän työn esimerkkitapauksessa on kolme osaa joissa tutkitaan tarkemmin kolmea näistä tarkoituksista Terasicin toisen sukupolven kehitys- ja opetuskäyttöön tarkoitetulla FPGA-kortilla. Tähän fyysiseen alustaan integroitiin kaksi järjestelmää. Ensimmäiseen kaksiuloitteiseen grafiikkajärjestelmään lisättiin informaatiorekistereitä jotka keräsivät siitä tietoa. Toinen järjestelmä taas keräsi tämän tiedon ja välitti sen eteenpäin.

Rekistereiden staattista käyttöä identifioimiseen tutkittiin esimerkkitapauksen ensimmäisessä osassa. Toinen ja kolmas osa hyödynnettiin rekistereiden dynaamisen puolen tutkimiseen tarkastelemalla niiden käyttöä verifioinnissa ja optimoinnissa. Jokainen näistä eri puolista paljastui hyväksi lisäavuksi digitaalisten piirien suunnitteluun.

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Electrical Engineering
Arvio, Juha: Augmenting IP blocks for verification and optimization
Major: Digital- and Computer Systems
Master of Science Thesis, 79 pages
Examiners: Dr. Tech Erno Salminen, Prof. Timo D. Hämäläinen
Keywords: Metadata, verification, optimization

February 2012

The verification of digital intellectual property (IP) blocks has always been a challenge. Simple IP blocks with straightforward test inputs, can be quite thoroughly verified with software simulators such as Modelsim. But the verification of a complex System-on-Chip (SoC) on a software simulator can last days or even weeks, and that assumes that every IP on the SoC has a working simulation model. Although modern programmable chips can be monitored in real time with tools like Altera's Signaltap II, they still only offer monitoring capabilities for a limited amount of signals and for a limited amount of time.

To overcome this deficiency, IP information registers (IIR) were developed for this thesis. These registers are used to store information pertaining to the IPs and the SoC as a whole. The information can be static or dynamic, ie. generated before or during run-time . The information itself can be used for many different purposes along with the verification of single IPs or whole SoCs.

The case study in this thesis has three parts where three of those purposes are examined with Terasic's second generation development and education (DE2) board. This physical platform was fitted with two systems, a 2D graphics system embedded with information registers and a system to monitor the first one using these registers.

The first part examined the identification aspects with static information whereas the second and third part examined the dynamic aspects of the information registers with their verification and optimization capabilities. Each of these aspects was deemed to offer a good service for developers designing digital circuits.

Preface

This Master of Science thesis was written at the Department of Computer Systems, Tampere University of Technology. The work for this thesis was partly done for the Funbase project which was funded by TEKES, and spanned from spring 2010 to spring 2012.

I would like to thank all my co-workers for providing a good working environment. Specifically two persons gave me valuable advice on completing this thesis. These two were Dr. Tech. Erno Salminen and Dr. Tech Timo Hämäläinen.

Tampere, February 28, 2012

Juha Arvio

Multiojankatu 28 D 29

33850 Tampere

Finland

Contents

List of figures

List of tables

Abbreviations

2D	Two dimensional
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
CRT	Cathode Ray Tube
DAC	Digital to Analog Converter
eCos	embedded Configurable operating system
fifo	first in, first out
FPGA	Field Programmable Gate Array
Gbps	Giga bits per second
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HD	High Definition
HDL	Hardware Design Language
HIBI	Heterogeneous IP Block Interconnect
HW	Hardware
ID	Identification
IDE	Integrated Development Environment
I/O	Input / Output
IP	Intellectual Property
IIR	IP Information Registers
JTAG	Joint Test Action Group
LUT	Look-Up table
Mutex	Mutual exclusion
OS	Operating System
PC	Personal Computer
PPU	Pixel Processing Unit
RAM	Random Access Memory
RTL	Register-Transfer Level
RTOS	Real Time Operating System
SDRAM	Synchronous Dynamic Random Access Memory
SRAM	Static Random Access Memory
SoC	System-on-Chip
SUT	System-Under-Test
SOPC	System-On-Programmable-Chip
SW	Software
Tcl	Tool Command Language
TTA	Transport Triggered Architecture

TUT	Tampere University of Technology
USB	Universal Serial Bus
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extensible Mark-up Language

1. Introduction

In the early years of semiconductor design, companies developed digital logic for their internal use. The design of these intellectual property (IP) blocks was widely protected and they were rarely licensed to third parties. But as the complexity of semiconductor chips has dramatically increased, the development of digital IP blocks has become ever so challenging. This has lead in the past decade into the increase in the reuse of these IP blocks [gsa].

Additionally, these blocks are more and more used to construct customised chips that include many different IP blocks in them. These constructs are referred to as System-on-Chips (SoC), because of the way they implement a fully working system on a chip as opposed to constructing the system of individual components [nul06]. A simplified illustraton of a SoC can be seen in Figure 1.1.

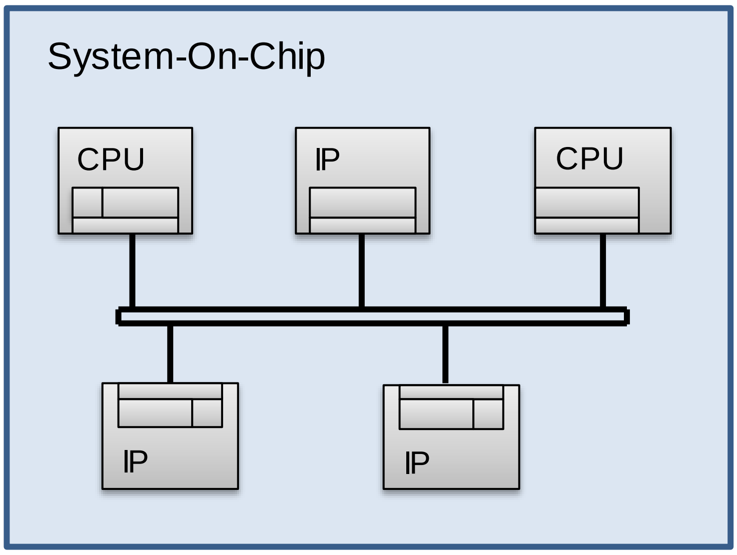


Figure 1.1 - Simplified structure of a SoC

The design of a hardware (HW) IP block, such as a microprocessor or a hardware accelerator, always includes the verification of its design. The importance of verification is steadily increasing as the complexity of the IP blocks and systems increase [keut00, keat02]. The first step to verify the correct behaviour of a design is to simulate it in a test bench on a program like Modelsim [model12]. The main hardware description languages (HDL) supported by Modelsim are verilog [ver95] and very high-speed integrated circuit HDL (VHDL) [vhd94].

However, there is also a need to test the IP block in the target platform which may not be feasible with software (SW) based simulation. For example, certain clocking and power-saving modes are hard to verify in simulation. Moreover, simulation models of external components, such as memories and network interfaces, might be missing. Additionally, simulation of a basic hardware platform with one or more SoCs can take days or even weeks to complete. This makes the iteration process of the verification almost impossible. To overcome these problems, the designer needs a way to construct a test platform before the application specific integrated circuit (ASIC) is produced of the SoC.

In 1984, Altera introduced the first reprogrammable logic device to the world called the EP300 which was a major improvement for prototyping [alt12]. Since then, the transistor counts of digital devices have increased significantly. From the 21st century, the chips have been large enough so that hardware designers have had the opportunity not only to synthesize single IP blocks but even whole systems on a chip. Nowadays, it’s common practise to test IP blocks and systems with a field programmable gate array (FPGA) or an array of them. Figure 1.2 presents a simplified structure of an FPGA chip.

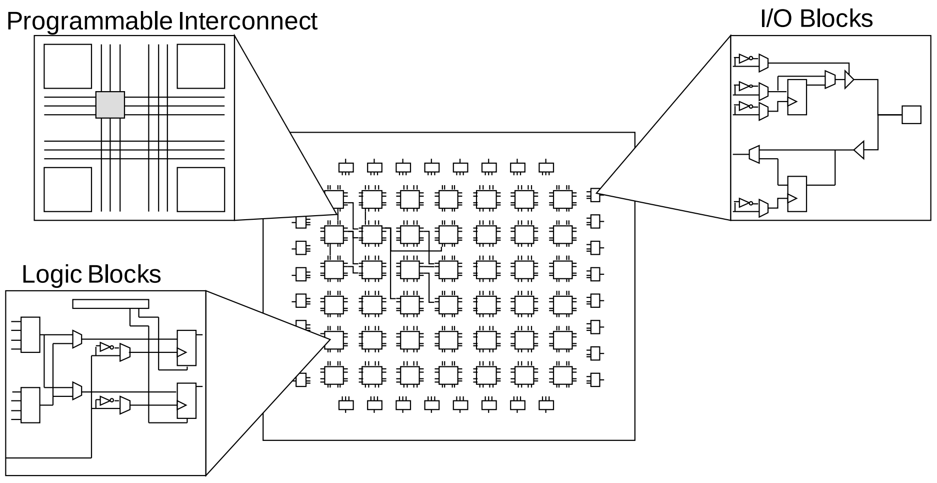


Figure 1.2 - Structure of an FPGA chip

The goal of this thesis is to utilize these modern programmable devices in SoC design. Moreover, IP components are augmented with special *information registers* to identify, verify and optimize their functionality and the system as a whole. These registers are examined thoroughly in chapters four and six.

The work presented in this thesis is carried out during a project called Function Based Platform (FunBase) [fun12]. It is a project that has been created specially from the need of the small to middle sized companies to design and create their own hardware platforms with the same kind of modern design flow that is used to create software.

The objective of the project is to develop a design flow and tools which enable the creation of an FPGA based product much faster and with less effort than before. The design flow also ensures that the development costs are low enough for the small and middle sized companies. In addition, the design flow helps companies with little or no expertise in hardware to create their own systems [kam11, sal11].

It's essential that a company's IP block is packetized so that it can be effortlessly sold and integrated as a part of another system. A company can also purchase IPs from other vendors to be used in their own systems. The project also aims to develop a physical platform to which end user defined functions can be created from modular software and hardware components. Further information of the FunBase project and its design flow can be read in the next chapter.

This thesis is partitioned into seven chapters. The first one is this brief introduction. The second chapter delves a little into the concept of metadata. Chapter 3 explores the tools and methods used in SoC design flow. Chapter 4 studies the concept of IP information registers. Basic concepts of computer graphics are described in chapter 5, and the practical usages IP information registers are explored in chapter 6. Finally, chapter 7 makes conclusions of the usability of these registers.

2. Metadata

The term *metadata* refers here to general information about IP components in addition to its source code. This information includes for example, interface specifications, documentation, lists of needed files, tool environment and so on. Most of it is created at design-time but some parts are defined at runtime.

2.1. Requirements from/to IP vendors

Before the actual design of a target system can be started, the requirements for the product must be obtained and accurately defined according to the needs of the customer. After these requirements are known, requirements for the system and its functions can be defined.

The underlying components or IPs which then carry out these functions can be selected from these requirements. The possible vendors for these IPs are then narrowed down to the best ones.

The requirements for the IPs include the required functionality, performance, cost, size etc.

2.2. Design time information

Design time information is information about the implemented system and its components. IP-XACT is a standardized format for capturing it [ipxact10].

Usually, part of the needed IP blocks exist already before the system design starts. Some of these are IPs developed previously in the company and a part is obtained from third party IP vendors.

The components or IPs have information about them that is needed for integrating them into the target system. This design time information includes the *vendor, library, name and version* (VLNV) of the IP. Information which is important for the integration of the hardware IP to the system include the signal widths and their names, maximum synthesizable speed for a specific FPGA chip or ASIC process, size and power usage of the IP on the specific FPGA etc.

This design time information is not used to verify their functionality but to integrate the IP to the target system.

2.3. Run time information

Run time information is retrieved from the target system by monitoring it somehow.

Physical logic analyzers can be used to do the monitoring. This is however limited to the signals that can be physically probed by the analyzer and any of the internal signals of the chips containing the digital circuits cannot be directly monitored.

Fortunately Altera provides a way to record the signals within a FPGA chip with their SignalTap II Logic Analyzer tool [sig11]. This real time monitoring tool has limited time length for the snapshot(s) of the signal traffic inside the FPGA and also a limited amount of signals that can be monitored at the same time. A typical snapshot has a few dozen signals captured for few thousand cycles. This comes from the simple fact that FPGA chips have limited amount of internal memory and logic elements and a good part of this resource will already be consumed by the actual SoC synthesized to the FPGA. The physical system which has the FPGA has to also be connected directly to a PC so this method is limited to be used in the developer’s work space.

3. System-on-Chip design flow

Large companies have enough designers to create hardware systems with the old way of designing digital circuits. The IP blocks which are designed this way are often ad-hoc in nature and are only used in the original system for which they were created. However, with a little more planning these digital IPs can be reused in later systems.

Software designers have long been using object oriented programming with different software layers. This has reduced the development costs because software can be reused in the future assuming that the documentation of the functionality is good.

3.1. FunBase SoC design flow

At the heart of the FunBase SoC design flow is the idea that the system specification is divided into different *functional blocks* rather than the actual hardware or software blocks [keut02, san07]. This abstraction provides the flexibility for the developers to design systems which can be later mapped into many different physical platforms. For example, the functionality to calculate a residual image from two input images can be mapped into a software, a hardware or a mixed hardware/software implementation based on the performance and resource usage requirements.

As FunBase is a project in progress there will be later revisions of the design flow but the general idea will remain the same. This thesis describes the first version of the design flow.

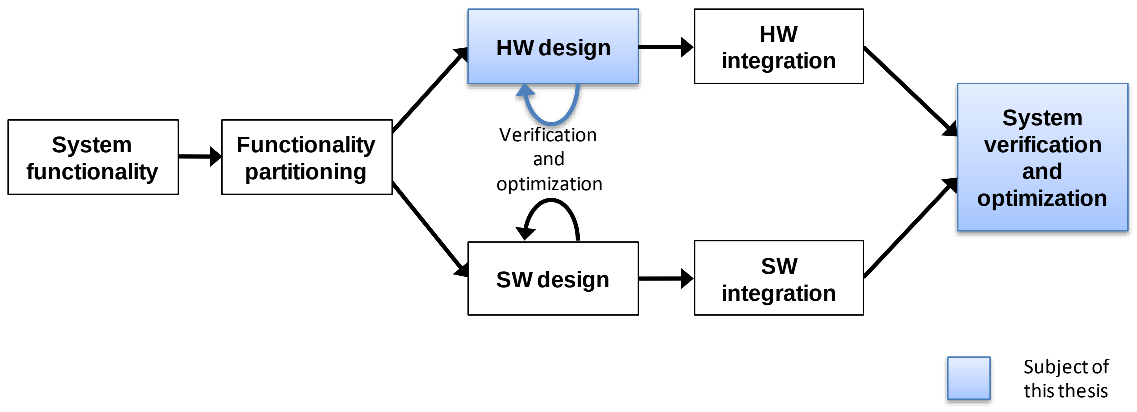

System functionality
Functionality partitioning
HW design
HW integration
Verification and optimization
SW design
SW integration
System verification and optimization
Subject of this thesis

Figure 3.1 - FunBase SoC design flow

The FunBase SoC design flow can be separated in to five parts as illustrated in Figure 3.1. The system's and its components' functionality is defined first. After this the functionalities are partitioned into different hardware and software components. Next, the required hardware and the software components are designed which also includes the verification and optimization of these individual components. After this, the hardware and software are integrated to the system with the FunBase tools and lastly the whole system is verified and optimized. This thesis focuses in improving the processes of hardware and system verification and optimization.

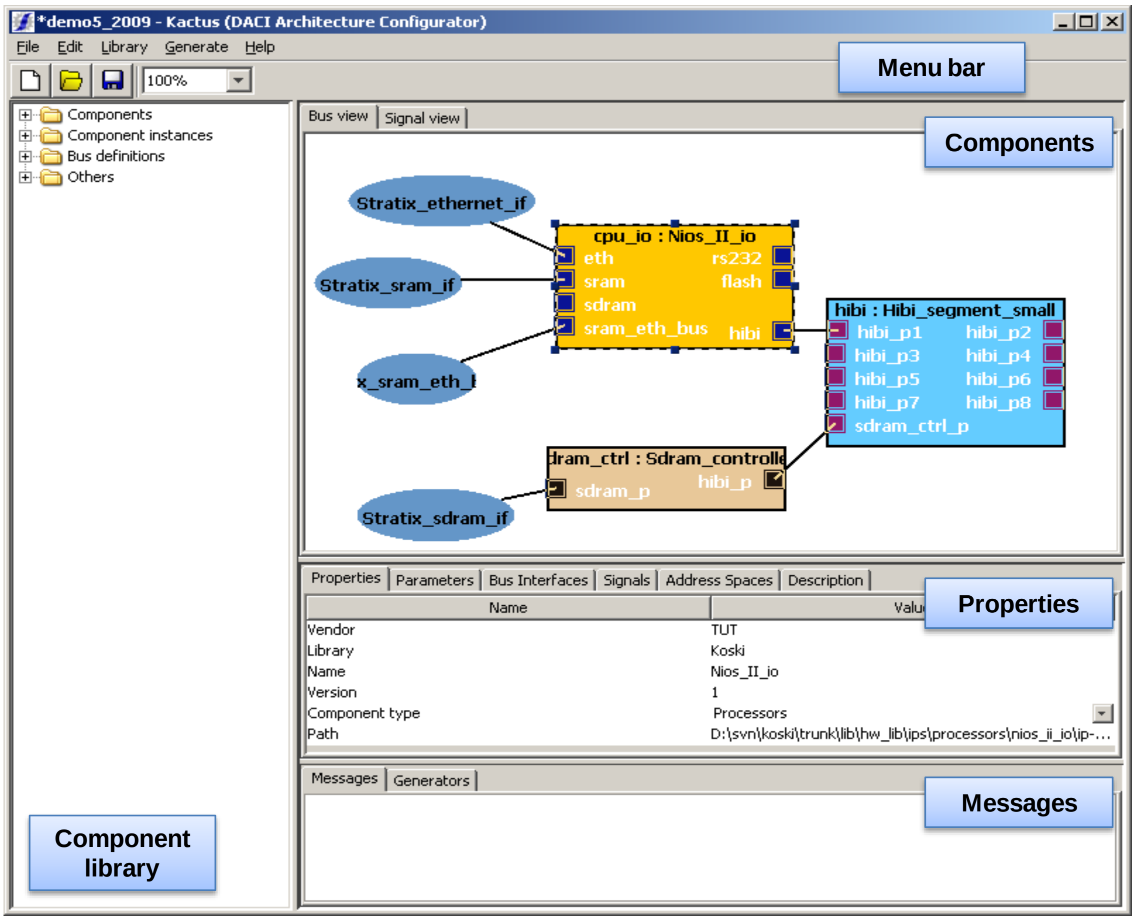
3.1.1. Design tools

The design tools used in the FunBase flow include software which were originally developed at the Tampere University of Technology (TUT), within Altera corporation and the open source community. The tools originating from TUT will be further developed as the project continues.

These tools include Kactus, Library manager, Component Editor, KoskiGUI and several generators created for KoskiGUI [kos09, kam11]. Most of these tools were written in programming languages that either use a virtual machine or run time interpretation to execute the code. This ensures the software's easy portability to several different operating systems including Windows, Linux and Unix. For example the graphical interfaces for the tools were written in Java and the generators run in KoskiGUI were written in scripting languages like Tcl and Python.

Several tools from Altera are also used. These include Quartus II, SOPC builder, FPGA programmer, nios2-downloader and nios2-terminal among others [asoft12].

Kactus is a graphical tool written in Java that is used to integrate components to a target hardware platform. It also generates the structural top level description of the hardware platform which is later used in KoskiGUI. The hardware platform consists of components and their connections described in IP-XACT format [ipxact10]. The generated design file for the platform is also in IP-XACT. The user interface for the Kactus tool consists of five sections which can be seen in Figure 3.2. These sections include the menu bar, component library, components, properties and messages section.

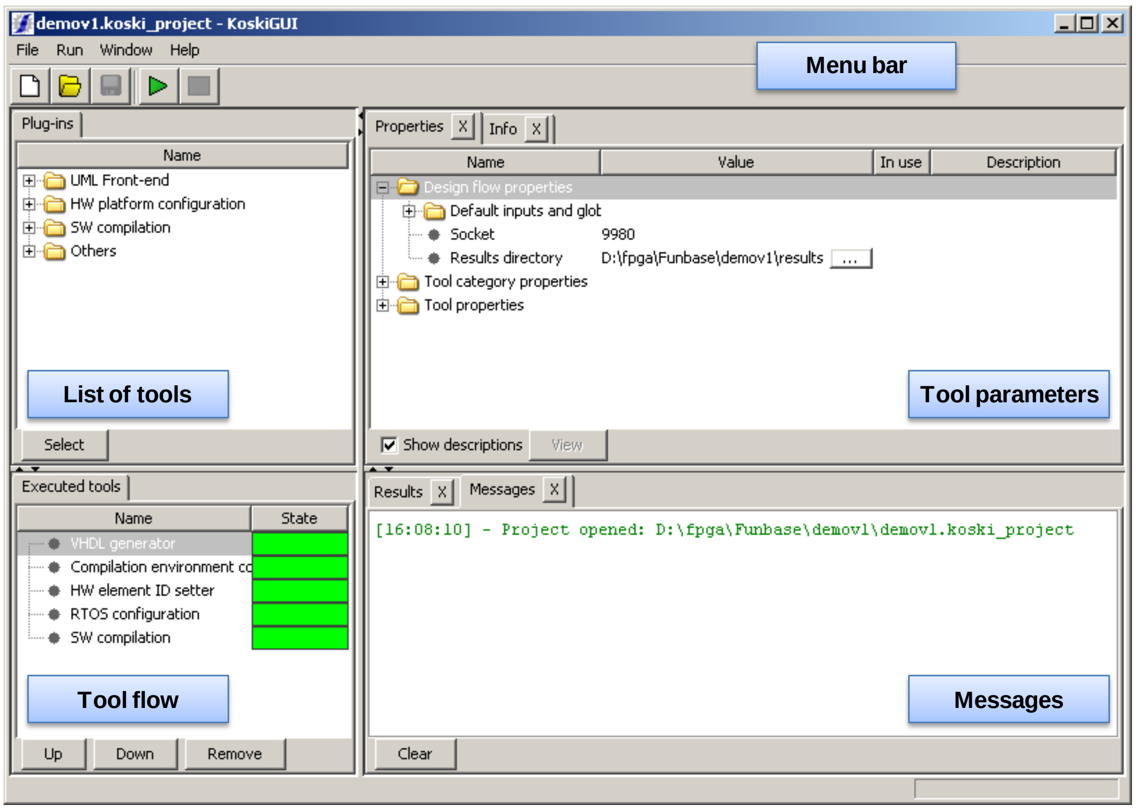


Name	Value
Vendor	TUT
Library	Koski
Name	Nios_II_io
Version	1
Component type	Processors
Path	D:\svn\koski\trunk\lib\hw_lib\ips\processors\nios_ii_io\ip-...

Figure 3.2 - Kactus tool

The menu bar is used to create and open projects, configure the program, start the generation of the hardware platform, etc. Components and connections can be selected from the component library and dragged to the components section to create a new architecture. Properties, parameters and other options related to the components can be set in the properties section and any messages related to the generation of the architecture can be seen in the messages section.

The IP-XACT descriptions for the IP blocks can be created with the Component editor which is launched from the KoskiGUI. It is also possible to create the descriptions with the Eclipse IP-XACT plugin or any other extensible mark-up language (XML) editor including basic text editors. A figure showing KoskiGUI and its main sections can be seen in Figure 3.3.



Name	Value	In use	Description
Design flow properties			
Default inputs and glot			
Socket	9980		
Results directory	D:\fpga\Funbase\demov1\results		
Tool category properties			
Tool properties			

Name	State
VHDL generator	🟩
Compilation environment co	🟩
HW element ID setter	🟩
RTOS configuration	🟩
SW compilation	🟩

```
[16:08:10] - Project opened: D:\fpga\Funbase\demov1\demov1.koski_project
```

Figure 3.3 - KoskiGUI tool

KoskiGUI is the tool where the automated generation of hardware description language (HDL) files and compilation of the software code is done. It has five tools which are used for VHDL generation, configuring the compilation environment, setting the HW element identifications (ID), real time operating system (RTOS) configuration and SW compilation.

The IP library is governed with a separate tool called Library Manager. The developer can easily add, remove and modify the IPs in the library with it. This tool is not mandatory but it simplifies and speeds up the design process.

Altera’s Quartus II is used to create the blank synthesis project which is later needed by the KoskiGUI. In the first version of the FunBase design flow, Quartus II is also used to synthesize the hardware system to the target FPGA. Other synthesis tools may also be used in later revisions of the design flow. Nios2-download is used in the prototype phase of the FunBase flow to download the software to the Nios II processors on the FPGA. Nios2-terminal is used to debug the system via JTAG-UARTs.

3.1.2. Hardware integration flow

First step in the hardware integration flow, which can be seen in Figure 3.4, is the gathering of all of the necessary hardware IPs to the IP library. These IPs can be added to the library with the Library Manager. If an IP does not exist in the library it must be created and packetized so that it can be added to the library. It is also possible to purchase already made IPs and packetize them if they were not done according to the FunBase flow.

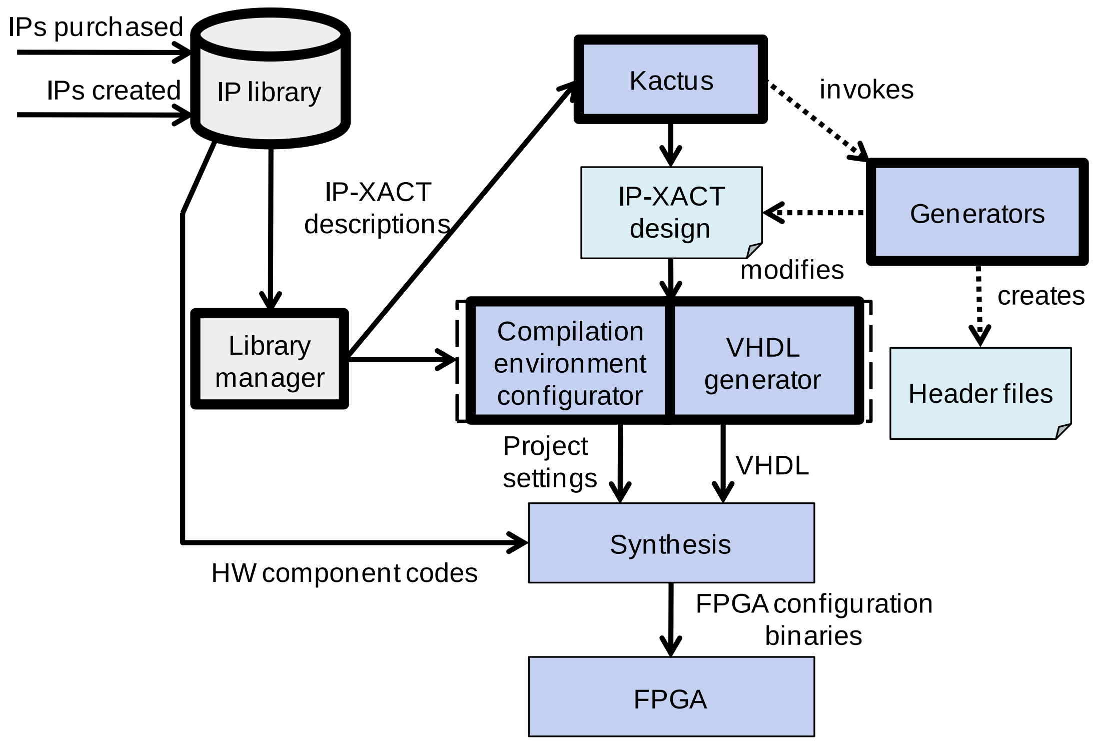


Figure 3.4 - Hardware integration flow

The main part of packetizing a hardware IP block is to create an IP-XACT description for it. The description includes the vendor name, library name, name of the component and the version. VHDL signal widths must be assigned manually in the first version of the SoC design tools. Bus interfaces are also created to map the VHDL signal names to the logical ones. After the IP-XACT XML file is created it can also be validated separately by the XML validator. This is also done by the VHDL generator in the KoskiGUI but it is better to validate the file as soon as possible.

After the IP is packetized, it is added to the IP library. To create a library component with the Library Manager the developer needs to set the ID, name and the path for the component. The hardware codes along with the IP-XACT definition must be added to the component. Software drivers can also be added.

When all of the required IP blocks are in the library, the developer proceeds to describe the HW platform with the Kactus tool. It has a very intuitive graphical interface where the developer simply adds the IP blocks and the connectors between them to describe the system. After the system is done, it will generate the necessary IP-XACT design needed by KoskiGUI.

In the next step of the integration flow, Quartus II is used to create a blank synthesis project. After this, KoskiGUI is used to run the necessary generators. These generators include the VHDL generator which as the name implies generates the necessary VHDL files required by the synthesis. More specifically it generates all of the VHDL files needed to connect the IP blocks in to a system. Compilation environment configurator and HW element ID setter also modify the VHDL files as is required by the functionality of the system.

The final step in the hardware integration flow is the synthesizing and fitting of the VHDL files to the target FPGA. This is done with the Quartus II tool from Altera.

3.1.3. Software integration flow

The software integration flow has fewer phases which use FunBase specific tools as the software industry has long had good tools to automate the design flow of software. A figure illustrating the flow can be seen in Figure 3.5. The software and hardware integration flows also have some overlapping like the adding of the software drivers for the hardware IP blocks in Library manager.

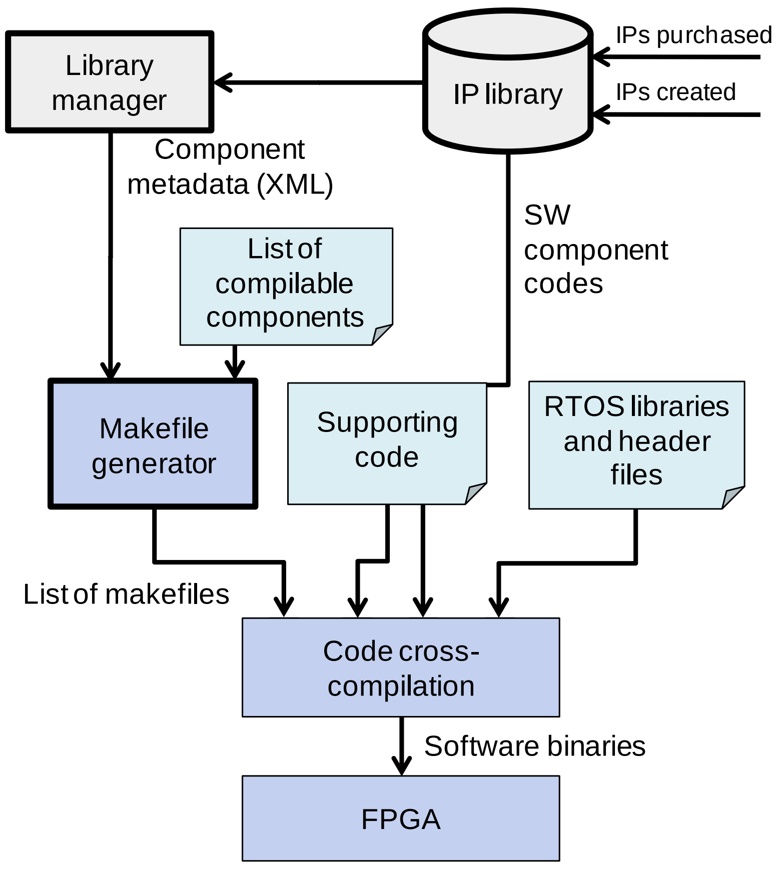


Figure 3.5 - Software integration flow

Version one of the FunBase SoC design flow can use three kinds of processing environments in the target system. These include a Nios II processor [nios11] with an eCos RTOS [ecos11], a transport triggered architecture (TTA) processor [tta11] with no RTOS and a personal computer (PC) environment with an Intel x86 compatible processor running Windows. The PC environment is only used as a part of the target systems in this early stage of the SoC flow but as the FunBase project continues it will be replaced with an embedded processor board with an Intel Atom processor running a Linux RTOS.

Actually, the eCos RTOS will not either be used in later revisions of the SoC flow as there is not a need to have multiple processors running a fully fledged RTOS. The removal of the eCos RTOS decreases unnecessary software overhead and thus the performance of the processor(s) is increased. Another improvement is achieved as the software run on the processor has smaller size.

The software integration flow starts the same as the hardware one with the gathering of the IP blocks needed by the target system. These are then integrated into larger software blocks which are run by the different processors in the system.

After this the RTOS environments are configured by the RTOS configuration tool in the KoskiGUI. And finally all of the software is compiled by the SW compilation tool.

3.2. FunBase hardware platform

Microteam [micro12] which is one of the FunBase partner companies designed a baseboard which can be used as a physical platform for the FunBase SoC design flow. The core of the FunBase baseboard is an Arria II GX FPGA chip manufactured by Altera [arria11]. It is a mid-range FPGA designed for transceiver applications and offers up to 3.75 Gbps of input/output (I/O) bandwidth. As can be seen from Figure 3.6 the board has extensive amount of high speed interfaces to meet the requirements of a broad range of systems. Smaller and less complex boards can and will be later designed based on this baseboard.

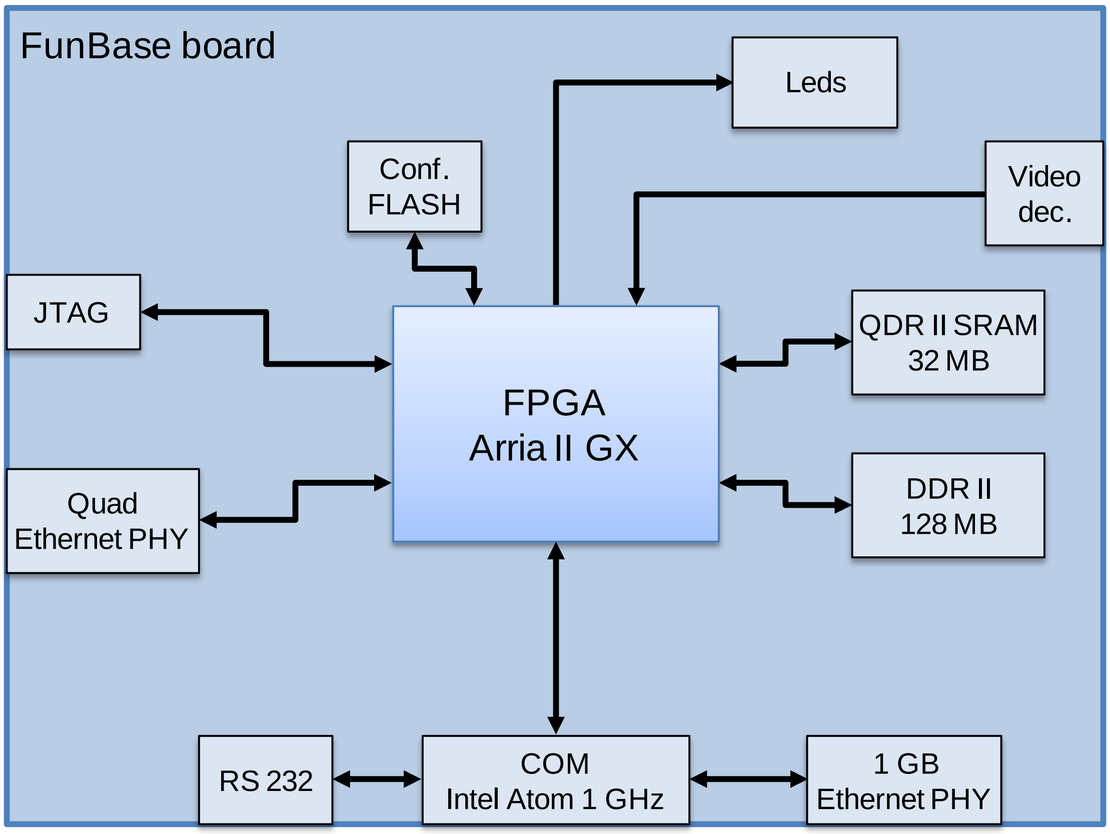
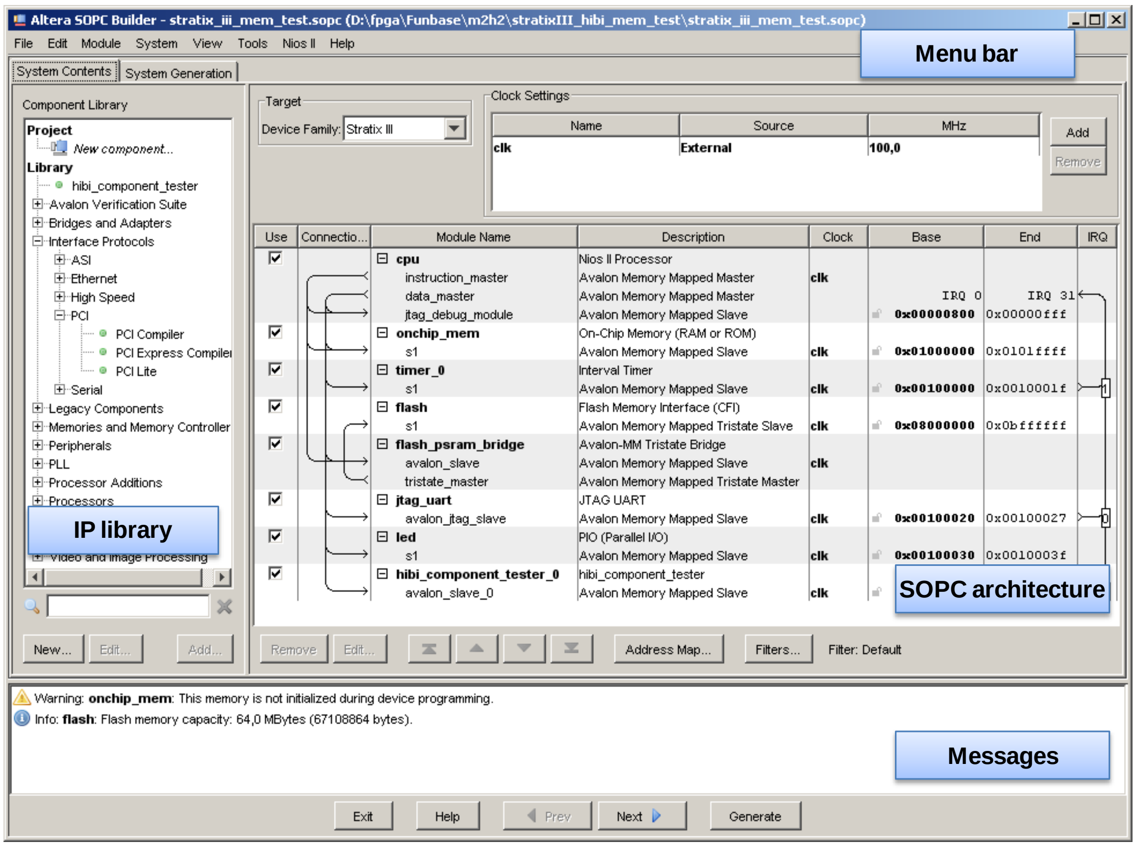


Figure 3.6 - FunBase hardware platform

3.3. Altera tools used in FunBase

As the FPGA chips used at TUT are primarily manufactured by Altera there are several tools made by them that are used in the FunBase design flow. Many of them integrate easily to the flow but as can be read in the next section of this thesis there is one that does not.

SOPC builder is a hardware design tool which is used to create SOPC sub-systems. These sub-systems comprise of one or multiple Nios II processors and the accompanying peripherals used by the processor(s). In short, SOPC builder can be described as a tool to create soft-core microcontrollers for Altera's FPGAs. The graphical user interface (GUI) of the SOPC builder can be seen in Figure 3.7. A SOPC sub-system is created by adding IP blocks from the IP library section to the SOPC architecture view. In this view the blocks can be configured and mapped into memory regions seen by the CPU(s). The HDL and some configuration files for the system is created by clicking the generate button at the bottom of the GUI. Messages related to the generation and design of the system can be seen in the messages section.



Device Family: Stratix III

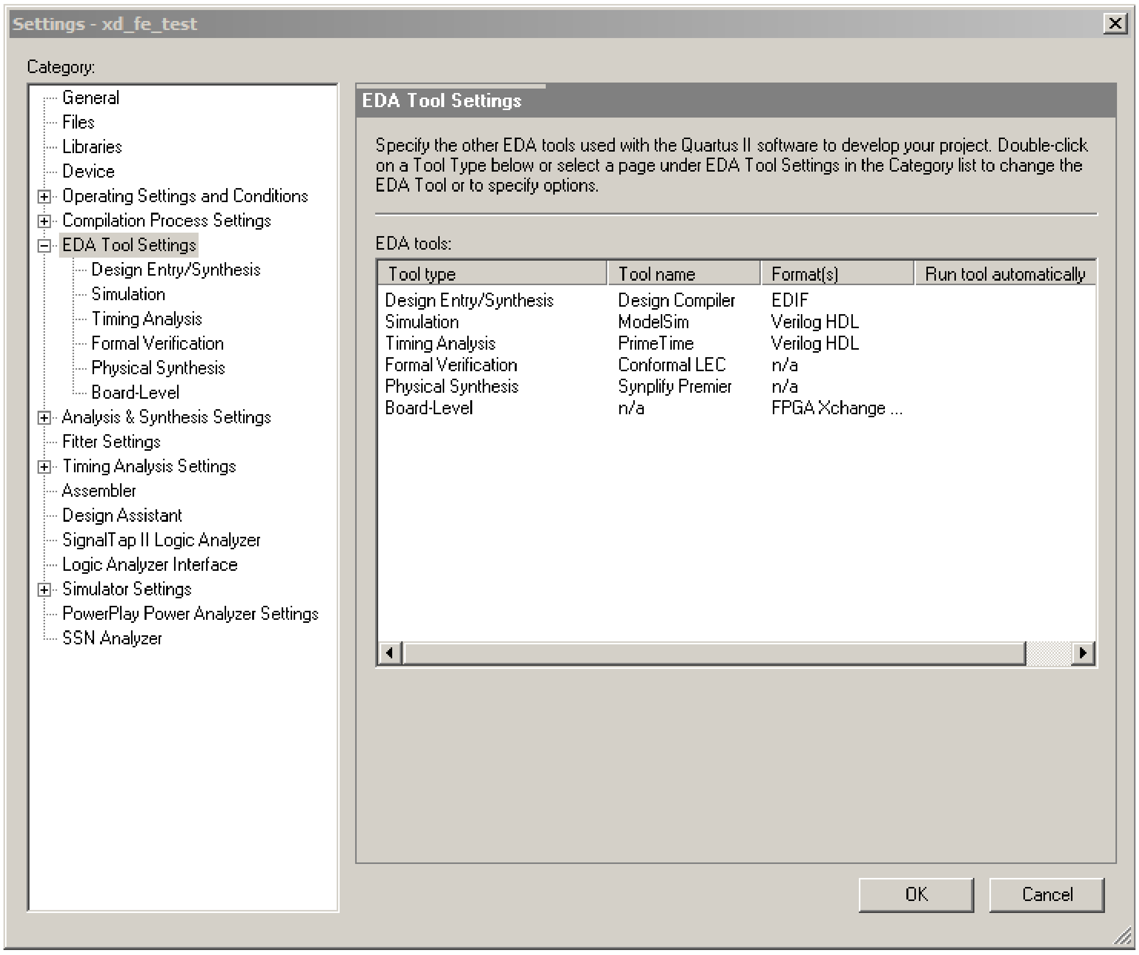
Name	Source	MHz
clk	External	100,0

Use	Connectio...	Module Name	Description	Clock	Base	End	IRQ
✓		cpu	Nios II Processor				
		instruction_master	Avalon Memory Mapped Master	clk			
		data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Memory Mapped Slave		0x00000800	0x00000fff	
✓		onchip_mem	On-Chip Memory (RAM or ROM)				
		s1	Avalon Memory Mapped Slave	clk	0x01000000	0x0101ffff	
✓		timer_0	Interval Timer				
		s1	Avalon Memory Mapped Slave	clk	0x00100000	0x0010001f	1
✓		flash	Flash Memory Interface (CFI)				
		s1	Avalon Memory Mapped Tristate Slave	clk	0x08000000	0x0bffffff	
✓		flash_psram_bridge	Avalon-MM Tristate Bridge				
		avalon_slave	Avalon Memory Mapped Slave	clk			
		tristate_master	Avalon Memory Mapped Tristate Master				
✓		jtag_uart	JTAG UART				
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk	0x00100020	0x00100027	0
✓		led	PIO (Parallel I/O)				
		s1	Avalon Memory Mapped Slave	clk	0x00100030	0x0010003f	
✓		hibi_component_tester_0	hibi_component_tester				
		avalon_slave_0	Avalon Memory Mapped Slave	clk			

Warning: **onchip_mem**: This memory is not initialized during device programming.
Info: **flash**: Flash memory capacity: 64,0 MBytes (67108864 bytes).

Figure 3.7 - SOPC builder

Quartus II is mostly used to synthesize and program the hardware design of a SoC to an FPGA. However, it is also a hardware design tool and it can be compared with the KoskiGUI tool excluding the software functionality. Like KoskiGUI, it can be used to integrate different tools for the SoC design flow. These tools can be either Altera's own or third-party tools. A dialog showing the third party EDA tool configuration in Quartus II is shown in Figure 3.8.



Settings - xd_fe_test

Category:

- General
- Files
- Libraries
- Device
- Operating Settings and Conditions
- Compilation Process Settings
- EDA Tool Settings
  - Design Entry/Synthesis
  - Simulation
  - Timing Analysis
  - Formal Verification
  - Physical Synthesis
  - Board-Level
- Analysis & Synthesis Settings
- Fitter Settings
- Timing Analysis Settings
- Assembler
- Design Assistant
- SignalTap II Logic Analyzer
- Logic Analyzer Interface
- Simulator Settings
- PowerPlay Power Analyzer Settings
- SSN Analyzer

EDA Tool Settings

Specify the other EDA tools used with the Quartus II software to develop your project. Double-click on a Tool Type below or select a page under EDA Tool Settings in the Category list to change the EDA Tool or to specify options.

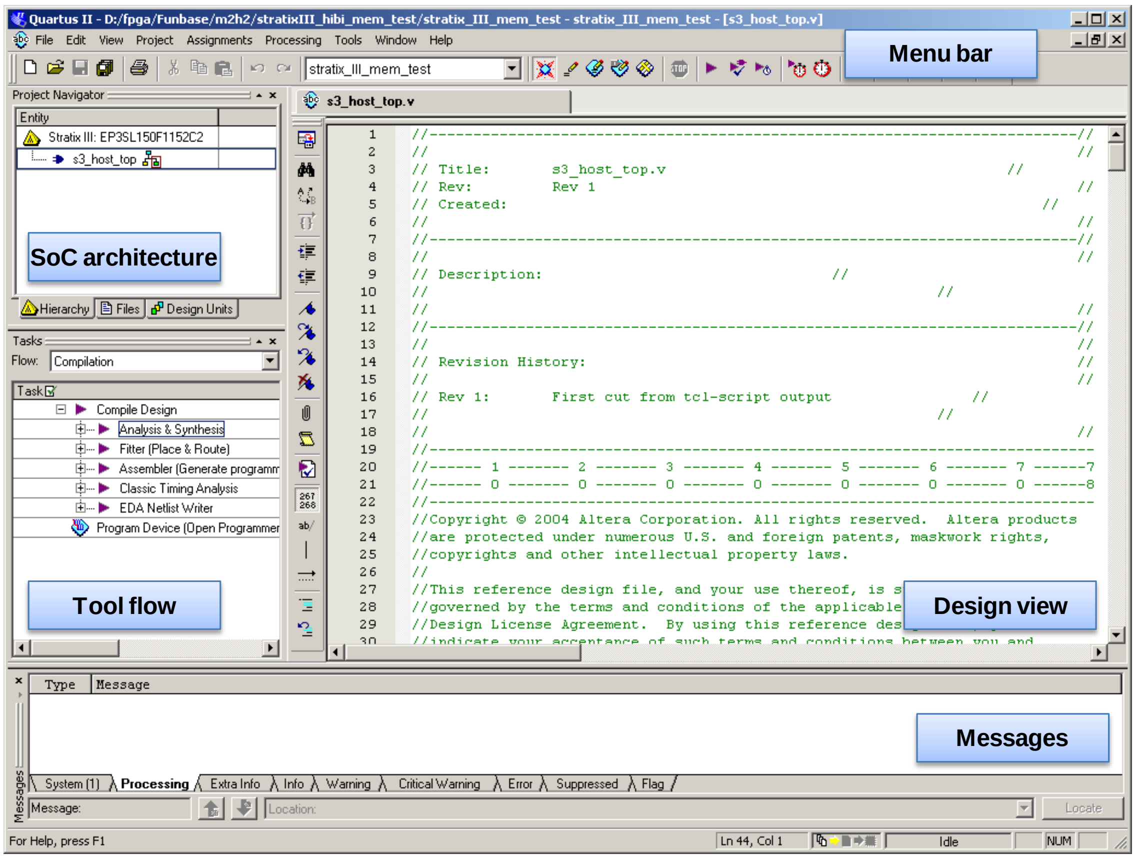
EDA tools:

Tool type	Tool name	Format(s)	Run tool automatically
Design Entry/Synthesis	Design Compiler	EDIF	
Simulation	ModelSim	Verilog HDL	
Timing Analysis	PrimeTime	Verilog HDL	
Formal Verification	Conformal LEC	n/a	
Physical Synthesis	Synplify Premier	n/a	
Board-Level	n/a	FPGA Xchange ...	

OK Cancel

Figure 3.8 - Quartus II EDA tool configuration

Tools launched from Quartus II are used to integrate, design and verify the architecture of a single SoC that is synthesized to an FPGA chip. It can also be used to create the top level design of the SoC either with a graphical block diagram file or a text based HDL file. SOPC sub-system(s) can be added to this top level design along with other IP blocks. An illustration showing Quartus II and its main sections can be seen in Figure 3.9.


Quartus II - D:/fpga/Funbase/m2h2/stratixIII_hibi_mem_test/stratix_III_mem_test - stratix_III_mem_test - [s3_host_top.v]
File Edit View Project Assignments Processing Tools Window Help
stratix_III_mem_test
Menu bar
Project Navigator
Entity
Stratix III: EP3SL150F1152C2
s3_host_top
SoC architecture
Hierarchy Files Design Units
Tasks
Flow: Compilation
Task
Compile Design
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate programm
Classic Timing Analysis
EDA Netlist Writer
Program Device (Open Programmer
Tool flow
s3_host_top.v
Design view
Type Message
Messages
System (1) Processing Extra Info Info Warning Critical Warning Error Suppressed Flag
Message:
Location:
Locate
For Help, press F1
Ln 44, Col 1
Idle
NUM

```
//-------------------------------------------------------------------------------//
//                                                                               //
// Title:        s3_host_top.v                                               //
// Rev:          Rev 1                                                           //
// Created:                                                                    //
//                                                                               //
//-------------------------------------------------------------------------------//
//                                                                               //
// Description:                                      //
//                                                           //
//                                                                               //
//-------------------------------------------------------------------------------//
//                                                                               //
// Revision History:                                                             //
//                                                                               //
// Rev 1:        First cut from tcl-script output                          //
//                                                           //
//                                                                               //
//--------------------------------------------------------------------------------
//------ 1 ------- 2 ------- 3 ------- 4 ------- 5 ------- 6 ------- 7 ------7
//------ 0 ------- 0 ------- 0 ------- 0 ------- 0 ------- 0 ------- 0 ------8
//--------------------------------------------------------------------------------
//Copyright © 2004 Altera Corporation. All rights reserved.  Altera products
//are protected under numerous U.S. and foreign patents, maskwork rights,
//copyrights and other intellectual property laws.
//
//This reference design file, and your use thereof, is s
//governed by the terms and conditions of the applicable
//Design License Agreement.  By using this reference des
//indicate your acceptance of such terms and conditions between you and
```

Figure 3.9 - Quartus II

Command line tools like nios2-download and nios2-terminal are used to debug the software on the Nios II processor(s).

Altera's SOPC builder is used to manually create SOPC systems which include one Nios II processor and IP blocks used by the processor. At the time of writing this thesis there have been four of these SOPC systems created for FunBase SoCs. These include three systems specifically made for Altera's Stratix II development board and one for the lower end DE2 development board. In these systems, a Nios II processor is used for fast processing with minimal amount of supporting IPs, a similar setup but with faster memory for the processor, a system with a slower clocked processor but more external communication interfaces and finally a similar system for the DE2 than the previous one. As can be clearly seen, this type of generation of multiple SOPC systems with each one differing little from the other leads to the result that each SOPC system has to be manually created just for one specific platform and this does not fit with the FunBase SoC design flow. Improvements to this SOPC builder integration issue are explored in the next sub-chapter.

3.4. Third party tools used in FunBase

Initially, Nios II processors are used to run software within an eCos RTOS. This RTOS is configured and the software library components are generated manually with the *ecosconfigtool* [ecos11]. The eCos RTOS has to be configured and generated for each of the different SOPC systems. This leads to a similar problem as was earlier described with the SOPC system generation. Fortunately for the FunBase project, it has been decided that the eCos RTOS is not really needed and will be replaced with a software model without a fully fledged RTOS.

In the future, the Nios II integrated development environment (IDE) is used to create the software library files. UCos II RTOS could also be used. Nios II IDE can be controlled by command line tools so integrating it to the FunBase SoC flow is fairly easy.

3.5. Third party IPs used in FunBase

These IPs include memory controllers, timer units, jtag-uarts, etc. Researchers at TUT have done some IPs which are used in the SOPC environment but they can also be used outside it with small modifications.

All of the IPs provided by Altera require some kind of generation. A part of the IPs can be generated outside the SOPC environment with the Megawizard tool included in Quartus II but the rest have to be generated into a SOPC system.
The SOPC IP blocks are manually designed by using the graphical user interface (GUI) of the SOPC builder.

3.6. Improving the integration of Altera tools to FunBase

Since the tools for the Altera corporation and the FunBase flow were developed separately, there is bound to be some overlapping between them. The best example of this overlap is Altera’s SOPC builder tool.

The SOPC systems created with the SOPC builder tool cannot be modified directly by the first generation FunBase tools. Hence, they are treated as any other IP block the internal structure of which cannot be changed with the FunBase tools. It can be said that they have even less adaptability than most IP blocks because they do not even have parameters to configure like many IP blocks.

These two reasons make the integration of IP blocks with an Avalon switch fabric interface require more knowledge about Altera's tools along with more work. To improve the situation, the tools have to be better integrated as a part of the FunBase SoC design flow.

To better integrate the SOPC tool to the FunBase flow, SOPC blocks have to be customizable with the Kactus tool. An example illustration of how the presentation of a SOPC block would be changed in a future version of Kactus is given in Figure 3.10. The grey CPU block indicates the situation as of now and the blue one is how the block would look like in the new version of Kactus with the internal architecture of the SOPC block hidden. The big transparent block shows what the SOPC block would look like with the internal architecture revealed. By enabling the Kactus tool to customize the SOPC blocks, any unnecessary components within the blocks can be removed. This is also shown in the figure below.

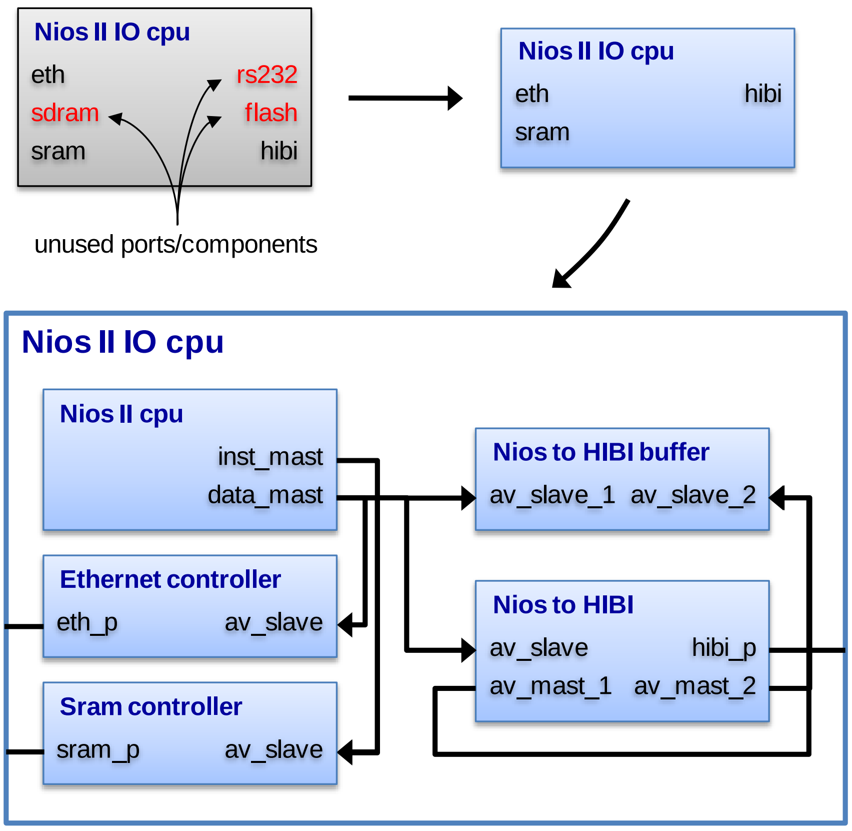


Figure 3.10 - Improving the integration of the SOPC builder to the Kactus tool

There are basically three different approaches for implementing this customization and improving the integration. The first one includes generating a large SOPC system which contains all the IPs that cannot be generated with the Megawizard in Quartus II and using these IP blocks with a Wishbone bus or similar. Similarly in the second approach one large SOPC system is designed but from which the Kactus tool will disable all the unnecessary IPs and then generate this customized system. The third approach would be to create a generator for the

SOPC system configuration file so that the Kactus tool could build any SOPC system which can be created with the actual SOPC builder.

3.6.1. SOPC builder integration approach one

Many of the IPs used in SOPC systems can only be generated within one. To use them outside they must be first generated into a SOPC system. After this the generated HDL file for the particular IP can be used to instantiate the IP outside the SOPC system it came from.

In this approach IPs and all their variations, which can only be generated with the SOPC builder, are generated into HDL files as was earlier described. These IPs are then packetized the same way any other IP is packetized for the FunBase SoC design flow.

This way of dealing with the SOPC builder integration problem would unfortunately create other problems and limitations that would have to be dealt with. These IPs would have an Avalon interface but no Avalon bus and the required arbitrators. Fortunately the Avalon interface is very similar to the Wishbone interface so the IP blocks could be connected to a Wishbone bus in a pretty straightforward manner. Wrappers could also be made to interface them with a Heterogeneous IP Block Interconnection (HIBI) bus [sal01].

There would be one major limitation with this approach. The IPs could not be configured like they can be with the SOPC builder so that multiple variations would have to be made to compensate for this. For example, many variations of the Nios II processor with different cache sizes and other properties would have to be generated beforehand. Memory controller variations for each different type of external memory would also have to be made. Considerable time would also have to be committed to packetize all these components.

For all these reasons, this approach would not obviously fit with the targets of the FunBase SoC design flow.

3.6.2. SOPC builder integration approach two

The second approach adds a new feature to the Kactus tool. The developer could add a configurable SOPC system to the Kactus project and decide what components would be generated into it.

To enable the creation of this new feature, one large SOPC system would initially be designed but only one variation of each of the IPs would be placed in the system. This system would then be treated as a starting point from which a modified SOPC system would be generated according to the developer’s decisions.

After the developer had made the SOPC system along with the components outside it, the Kactus tool would then modify the base SOPC system configuration file and initiate the generation of this modified SOPC system. The modification would simply include three kinds of operations. First the Kactus tool could disable any unnecessary components, secondly it could make copies of the components and finally it could change the configuration parameters of the components. As a result, Kactus could make any SOPC system which included the components found in the base SOPC system.

In this approach the Avalon bus would not have to be replaced and more importantly excessive time used for generating and packetizing the IPs could be avoided. Considerable work would still have to be done so that Kactus could make the required modifications to the SOPC system’s configuration file.

3.6.3. SOPC builder integration approach three

The last approach adds the ability to fully create a SOPC system configuration file to Kactus. This would include the most work of all of the three different approaches and it would not necessarily offer significant advantages over the second approach.

3.6.4. SOPC builder integration conclusion

As a conclusion, it can be said that the second approach is the one to go for. If it would not be sufficient for some reason, the work put to adding the SOPC system configuration file modification function to the Kactus tool could be continued and full generation of the configuration file could be achieved.

3.7. FunBase case study

To demonstrate the main capability of the FunBase SoC design flow an example system is created. As was earlier stated in this chapter, the main idea of the FunBase SoC flow is to treat parts of a system as functions rather than actual hardware or software components. This way the functions can be later mapped to the actual components based on the requirements and limitations of the target platform.

3.7.1. Image residual with different hardware architectures

This case study deals with a system capable of producing a residual image of two input images. The residual of two images shows the differences between them. An illustration of this functionality is given Figure 3.11.

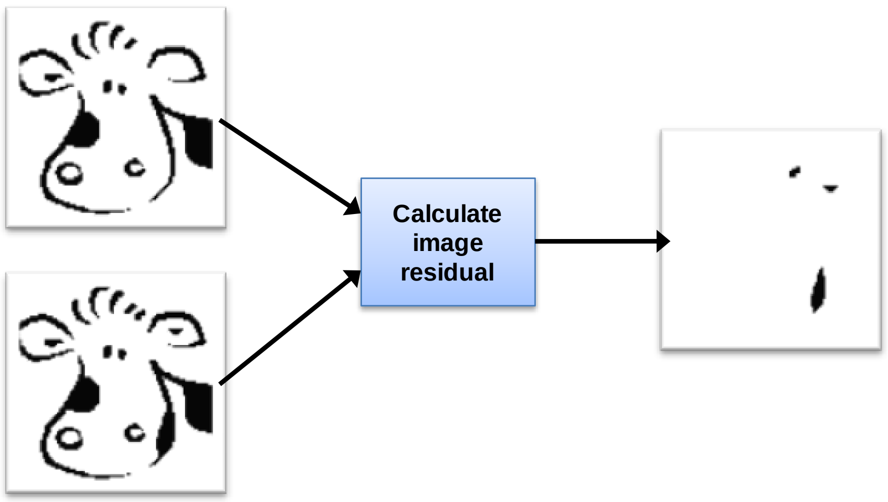


Figure 3.11 - Calculation of a residual from two images

Four different hardware architecture variations are created to demonstrate the function abstraction capability of the FunBase SoC design. The structure of these architectures is depicted in Figure 3.12.

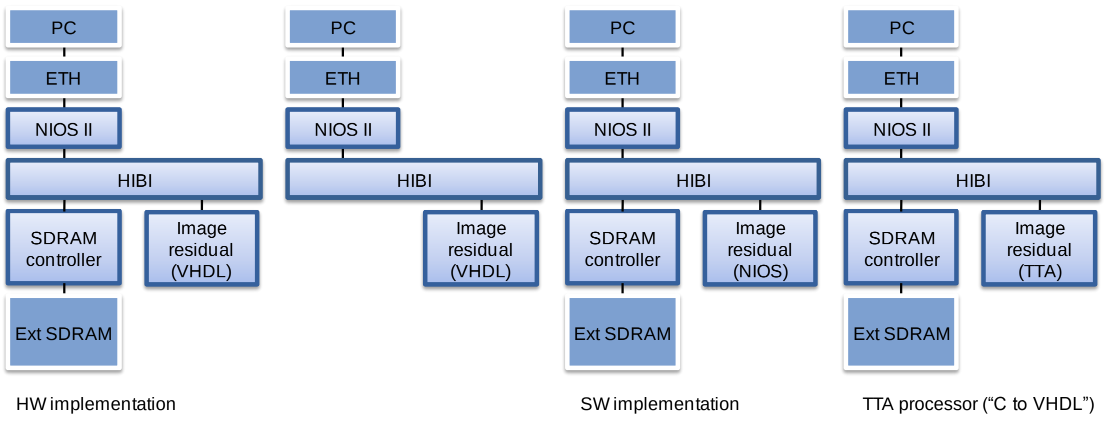


Figure 3.12 - Image residual with different hardware architectures

In these architectures, the image residual function stays unchanged as the underlying implementation changes. The function is implemented with two hardware only, one software only and a mixed hardware/software implementation. In the hardware only implementations, the image residual function is performed with a hardware component with and without using an external SDRAM chip. The software implementation uses a Nios II processor with SDRAM and the mixed implementation uses a TTA processor with SDRAM. This specific TTA processor was developed at TUT and is generated from C software source files into a hardware accelerated version of the software.

4. IP information registers

As was already discussed in the introduction for this thesis, traditional software simulation offers the possibility to examine signals through relative long time periods but extra effort is needed to create the simulation models for the whole range of hardware components. If the simulation models for the components behave the same way as their physical counterparts, the register-transfer level (RTL) simulation provides a view to the system's signals on a clock cycle level. But as the system is simulated on this level, the time needed to carry out a sufficiently long simulation can be measured in days or even weeks. This of course makes it practically impossible to iterate the verification process with these kinds of simulations, as was already noted.

The other way currently to verify the behavior of a system is to use some kind of hardware probing. Altera provides a tool for this called the SignalTap II Logic Analyzer. With this tool, signals can be probed directly from an FPGA chip at runtime. There are though a couple of critical disadvantages with this method. Only a small group of signals over a short time period can be examined at a time due to the finite logic cell and routing resources of the FPGA chip. Also if the design of a SoC consumes nearly all of these resources it will be impossible to use this kind of probing.

It would be good to have the means to record events and information from a physical prototype through long time periods [sal11].

To achieve this goal, the specification for these information registers, which are embedded to IP blocks, were developed for this thesis. These registers are added alongside the pre-existing ones and are filled with important information from the blocks and the system at run time. This information is used to check the IP blocks and the system for possible error events which may significantly aid the verification process. Information is also used to optimize the performance. Additionally the registers can be used to identify IP blocks and their place on a CPU's address map. An illustration of what the architecture of an IP block with embedded IP information registers (IIR) looks like is given in Figure 4.1. The bus interface is on the left. Information registers are distinct from the regular ones in this example and hence, separate logic for interface sharing (arbitrator) is needed.

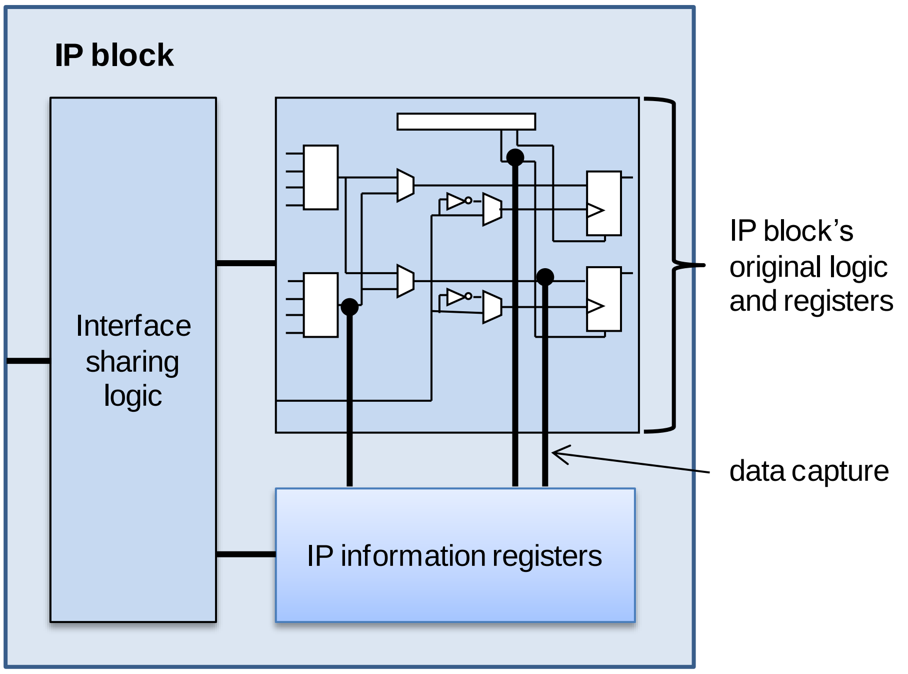


Figure 4.1 - IP block with embedded information registers

A simplified test setup using the information registers is depicted in Figure 4.2. The setup includes a System-Under-Test (SUT), a data gathering system attached to it and a PC to analyze the data.

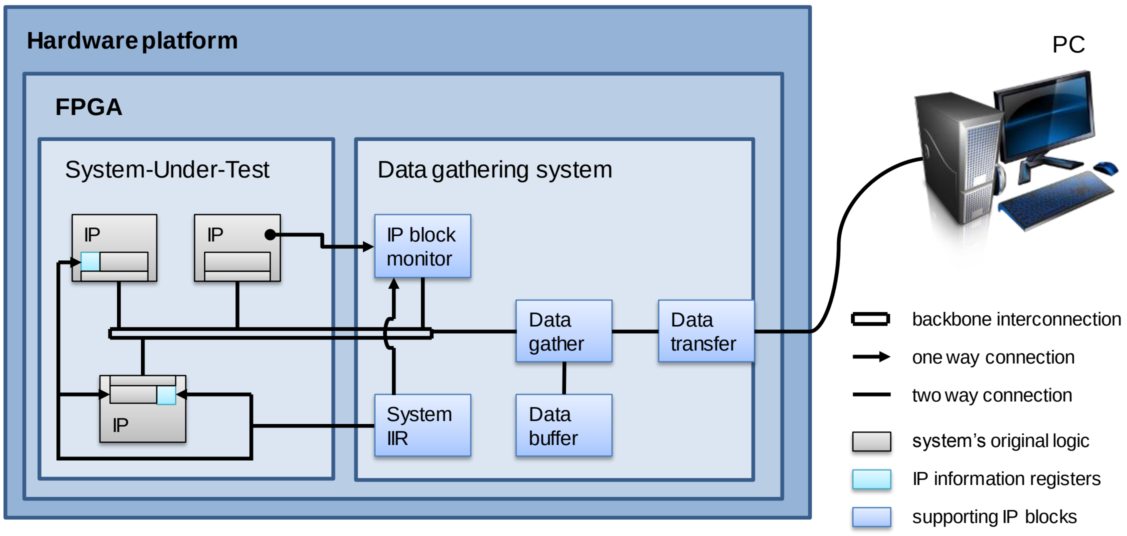


Figure 4.2 - A test setup with IP information registers

4.1. Verification and optimization flow

The flow for the verification and optimization process using the IP information registers is depicted in Figure 4.3. In this flow, a set of usage cases are defined from the requirements specification for the system. Possible optimization areas for the system and the IP blocks are explored and defined from the usage cases. And if any problem areas can be found they are also defined. After this, concrete plans for test runs for the system are specified.

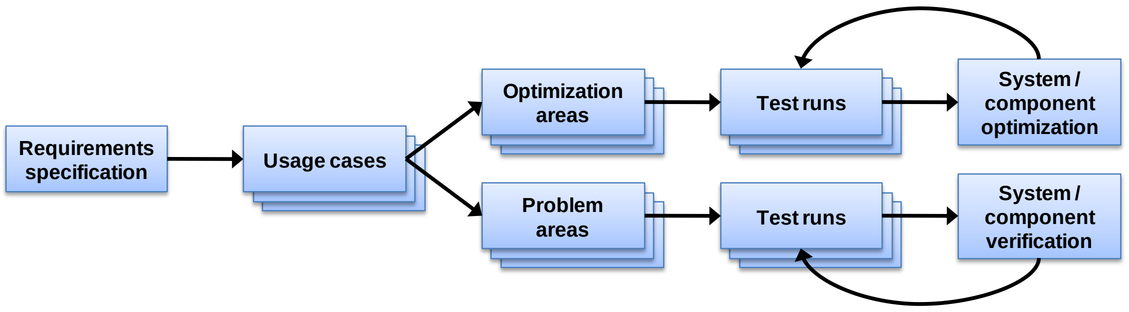

Requirements specification
Usage cases
Optimization areas
Problem areas
Test runs
Test runs
System / component optimization
System / component verification

Figure 4.3 - Verification and optimization flow with IP information registers

After the test run is initiated, the data gathering system periodically accesses the IP blocks and reads the contents of their information registers. After this, the gathered data is stored in a temporary data storage unit and sent to the developer's PC for analysis after the test run has completed.

Any parts not up to the original specification or any areas for optimization are addressed by the developer. After the possible corrections and optimizations are made, the same test run is repeated to analyze the outcome of these changes. This process is then iterated as many times it is necessary to achieve the wanted results.

4.2. Resources

One needs some extra logic and routing resources from an FPGA chip to instantiate the required components for information gathering. These components can be divided into IP information registers and supporting IP blocks. These registers and blocks can be seen in the earlier Figure 4.2. As can be seen from the figure, the data gathering unit is attached to the same backbone interconnection that connects the SUT's IP blocks. This way there is no need for an additional bus to be used to retrieve the data from the information registers.

External IP block monitors can also be attached to the backbone bus to monitor IP blocks with no embedded information registers.

A version of the test setup with a dedicated IP information bus could be later designed to gather information which requires continuous monitoring along with a high data bandwidth.

4.2.1. Register and interconnection types

The term register is used loosely in this thesis as there can be multiple physical registers or even on-chip memories behind them. A register is only used to describe an entry point for the underlying memory.

IP information registers come in three main types which include regular, fifo and special header registers. These registers are described in Table 4.1.

Table 4.1 - IP information register types

Register type	R/W/C	Description
Regular	R/W/C	A single word
Fifo	R	The number of words is known and only words containing the actual values are stored
IIR header	R	The words are stored normally and a word with all bits set to one indicates the end of the register

Regular registers can be subjected to three kinds of operations which include reading (R), writing (W) and clearing (C) the information on the register. A register can be either writable or clearable but not both. A register's content is cleared by writing any word to the register. No specific word is defined to reduce the amount of needed logic.

Fifo registers are associated with the logging capabilities in certain information registers. These fifo registers act as read only sources for the data stored by the logging logic. Finally the read only IIR header register has special functionality which allows it to change its contents during sequential accesses. IIR header registers are further described in the next sub-chapter.

The register map of an IIR enabled block is divided into three different parts as can be seen in Figure 4.4. These parts include the IP block's original registers, the general information registers and the optional information registers. The structure and functionality of these registers are described in later sub-chapters.

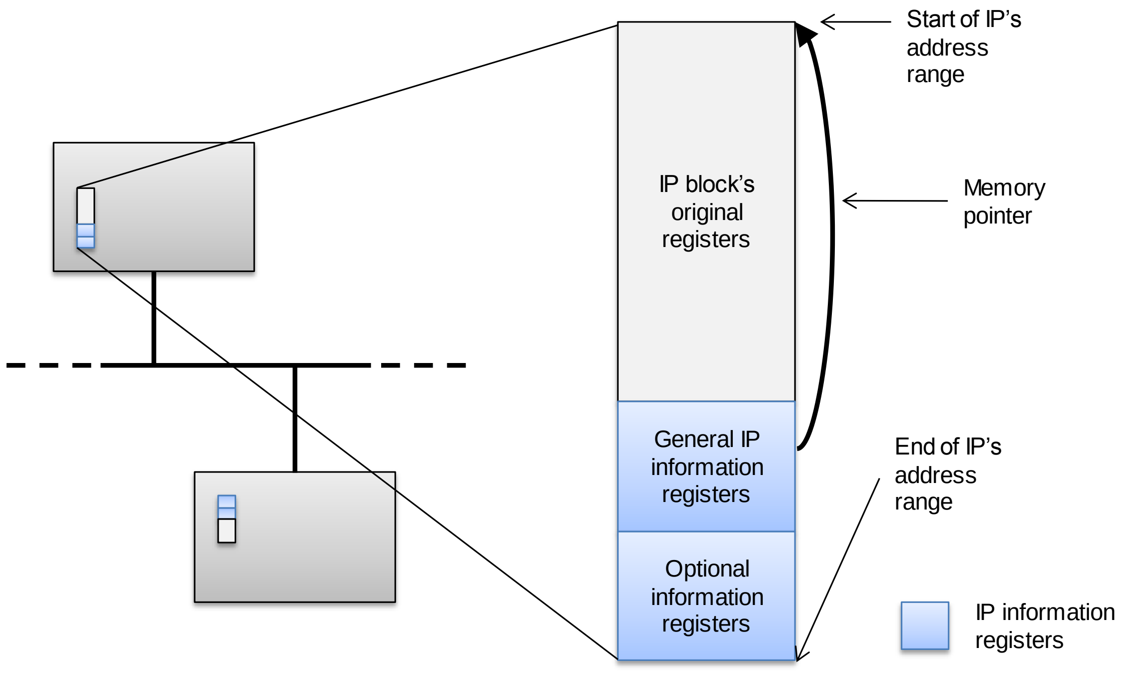

Start of IP's address range
IP block's original registers
Memory pointer
General IP information registers
End of IP's address range
Optional information registers
IP information registers

Figure 4.4 - IP information registers in IP block's memory space

4.2.2. General information registers

The general information registers contain registers to identify the parent IP block along with other miscellaneous data and functionality.

To comply with the specification, an IP block has to reserve a continuous address space for at least the general information registers which are described in Table 4.2. Offsets are given as 32-bit words. This address space can reside anywhere within the IP block's greater address space.

Table 4.2 - General information register layout

Offset	R/W/C	Name	Description
0x00	R	IIR1 header	A 32-bit header that inverts its byte ordering on reads
0x01	R	IIR type	0: internal IIR, 1: external IIR
0x02	R	IP reg. offset / address	Memory pointer in Figure 4.4
0x03	R/W	IP reset	Can be used to reset the IP-block
0x04	R	Instance number	Number to distinguish different instances of IIR blocks
0x05	R/W	Mutex	Used in multi-master systems
0x06	R	VLNV	Four ascii strings separated by null bytes
…	R	...	
0xXX	R	VLNV	
0xXX	R	Extra information	Optional information, fill with zeros if not used

The arbitrary placement of this address space is enabled by the IIR1 header register which is a special read only register used to identify the address space. This register returns a 32-bit ascii string reading “IIR1” on first access, and a “1RII” string on second access. Successive accesses will repeat the same alternating pattern. Using this unique characteristic, a processor can scan its full address space looking for IIR enabled IP blocks.

The second register contains three bits of information about the IIRs and the parent block. If the first bit on this register is one, the IIR block is external but otherwise it resides inside the parent IP. Secondly, the parent IP has accessible registers if the second bit is one. For example a Nios II CPU has not got any registers accessible outside and therefore its external IIR block has this bit set to zero. Lastly, if the third bit of this register is set to one the parent IP can be set to a reset state and the reset register on the IIR block is enabled. The SRAM controller used in this thesis cannot be set to a reset state and has this bit set to zero.

The information on the previous register determines how the third register is interpreted and it either has an IP register offset or an address. If the IIR block is internal this register provides the offset pointing to the parent block’s registers otherwise it provides a direct address to the registers. The direct address has to be set manually before synthesis. If the IP block has not got any accessible registers this register is set to zero.

An IIR enabled IP block has usually two reset signals. A system reset which comes from outside of the IP and an internal IP reset signal coming from the IP reset register. The fourth register is this reset register which can set the parent IP block to a reset state for example to recover from an error event. The reset is active high and it should normally be set to low at system reset.

If the IIR block of an IP is accessed by more than one component on the system, a multiple access mutex has to be implemented on that block. The mutex is the fifth register and it has to be used when accessing registers which operation has to be uniform and to ensure that when one block (e.g. a Nios II CPU) is accessing the registers no other component will interfere with the access.

The VLNV registers have four ASCII strings of basic information concerning the IP block and its creator. The strings are null terminated and include the vendor name, the library name, the name of the component as well as the component's version.

Lastly, the extra information registers are optional and can be used to store additional information pertaining to the parent IP or the IIR block. The string on the registers is null terminated and has to be filled with zeros if not in use. Useful information on these registers could for example be the design date of the parent IP.

4.2.3. Optional information registers

The optional information registers reside right after the general information registers and accommodate registers specific to the IP block. These registers mainly provide functionality to verify and optimize the IP block and the system. Other useful functions like usage statistics are described in Table 4.3.

Table 4.3 - Optional information register usages

Usage	Example
Verification	Log to examine CPU crashes
	Number of faulty write or read accesses done to IP block
	Clock cycles without write or read accesses done to IP block
Optimization	Log to optimize shared memory usage
Statistics	Number of write or read accesses done to IP block
	Number of cycles/frames from reset
	Bytes written to frame/line buffer in a frame/period of frames (eg. fill rate)
	Log to store unused frame/line buffer cycles

The log registers which can be present in the optional information registers are described further in chapter 6.

4.3. Supporting IP blocks

There are five different kinds of supporting IP blocks defined in this specification: the data gathering unit, data transfer unit, data buffer, system IIR and the external IP block monitor. These were previously illustrated in Figure 4.2.

4.3.1. Data gathering unit

To transfer the data from the IP information registers to the developer, a data gathering mechanism has to be implemented. There are basically two different approaches to implementing this mechanism.

Both approaches include a data gathering unit on the FPGA and a connection from this unit to the PC. The first approach sends data continuously to the PC whereas in the second approach the data is written to a data buffer during the monitoring and sent to the PC after the monitoring ends. The second approach might be the only viable one if the data bandwidth for the gathered information is too large to be sent in real time. Unfortunately this approach is not always practical in systems meant for the end user market.

The data gathering unit can either be a processor or a custom data gathering component.

4.3.2. Data transfer unit

To transfer the data to the PC and the developer, a connection is naturally required for it. Additionally to interface this connection to the FPGA, a controller for the connection has to be included. These two blocks form the data transfer unit.

Depending on the bandwidth and other requirements for the connection, some of the following connections can be considered: a USB connection with a virtual JTAG UART, a USB connection with a proprietary transfer protocol, a serial RS232 connection or an ethernet connection. JTAG UARTS and RS232 connections are traditionally quite slow (in the order of 100 kbs) [tex02] and therefore are not sufficient for many systems. A USB connection on the other hand can vary from the slow 1.0 standard (1.5 Mbs) to the ultra high speed standard of 3.0 (4.8 Gbs). As the 3.0 standard is quite new and not mature, the transfer speed of a USB connection is practically limited to the speed of the 2.0 standard (480 Mbs) [usb11]. An ethernet connection can have a maximum speed varying from 10 Mbs to 10 Gbs. Though ethernet controllers for the basic customer market support only up to 1 Gbs speed [eth08].

4.3.3. Temporary data memory

The temporary data memory comprises of an actual physical memory and its controller. The unit is used if the gathered data is sent to the PC after the monitoring is done. Of course the system must have a memory with extra space and bandwidth on it. Considering the latter two requirements, this will not usually be plausible to be used on a product that is used by the end user.

This unit can be really useful when a system is verified or optimized on a development platform and the collected information requires high data bandwidth. It subsequently removes the need for a high bandwidth connection to the PC because the data does not need to be transferred in real time.

4.3.4. System IIR

A system IIR is a special IIR block that has information pertaining to the system and also provides services to regular IIR blocks. In the first version of the IIR specification, it provides a service with its embedded system counter.

The system counter is very basic in its functionality but it still offers one of the main functions of the IP information registers which is the ability to store timestamps in the log registers. It is constructed of only one 32 or 64-bit wide counter. This counter is set to zero when the system reset is active and incremented by one on every clock cycle when the system reset is inactive. The counter is then connected to all of the IIR blocks that have log register(s).

Additionally the VLNV registers on the system IIR can be used to discern if a program run on the system's CPU was meant for the system, as is done on the first part of the usage case in chapter six. This functionality can be especially useful for developers dealing with FPGAs who can accidentally use either a wrong FPGA configuration or a wrong program.

4.3.5. IP monitor

An additional IP monitor is needed for each IP block that does not have information registers and cannot be modified for some reason. For example a Nios II processor can only be monitored by an external IP monitor because the HDL of the Nios II processor is encrypted and therefore not modifiable.

This kind of an IP monitor implements all the required information registers. The VLNV registers of such a monitor should be filled with information associated with the monitored IP block.

5. Basics of computer graphics

One needs to know the basics of computer graphics to understand how IP information registers can be used to optimize a 2D graphics system in the case study for this thesis. There are many different kinds of methods and hardware designed to generate computer graphics, but there is a set of basic concepts behind them [ake08, eck01].

This chapter describes a part of these concepts along with the basic functionality of computer monitors and the graphical elements used in line buffer based graphical processing units (GPU).

Every graphics system consists of five main components. These include the graphics memory, blitter, scene memory, image buffer and the digital to analog converter (DAC). These components are shown in Figure 5.1. The contents of the graphics memory and image buffers are only partially visible to save space.

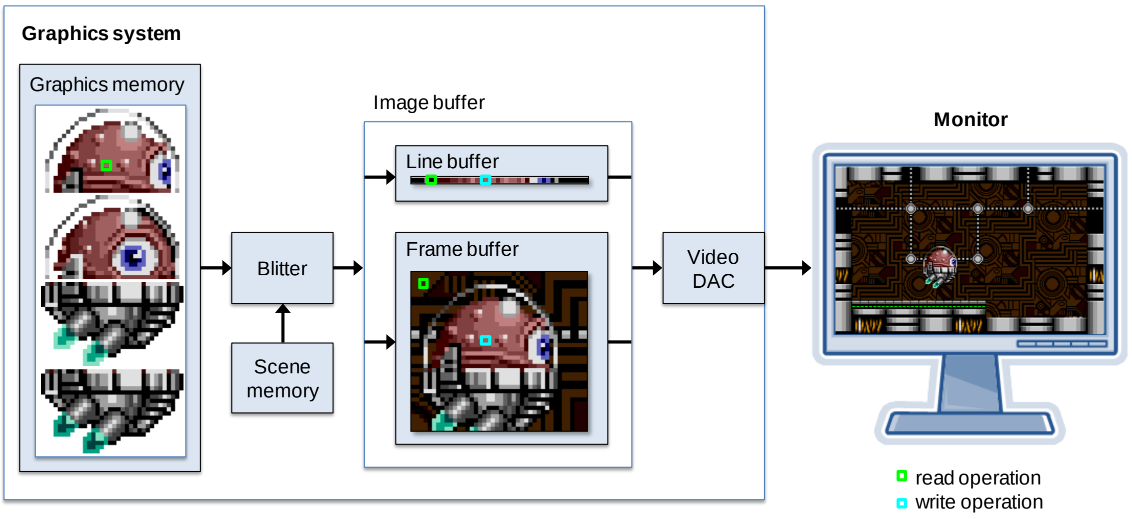


Figure 5.1 - Image generation methods

5.1. Color information

Every pixel on an image buffer has to be described as a binary number. The color of the pixel can either be directly read from this number or can be derived from this number using a color palette [eck01]. Many of the devices intended for customer markets from the 1970s to the start of the 1990s recorded the pixels' color information as index number tied to a color palette. Figure 5.2 shows 4-bit color palette with 16 different colors.

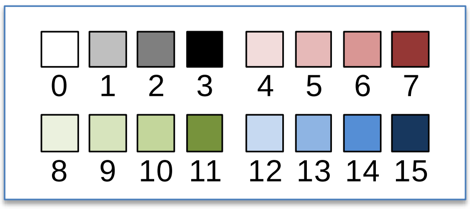


Figure 5.2 - Example of a 4-bit color palette

The colors of pixels are usually coded in RGB format where every color consists from a combination of three base colors (red, green and blue). If every one of these base colors is described with 8 bits, the pixels colours consists of a combined 24 bits which can define 16 million of different colors. This color format is in wide use in modern graphics systems.

5.2. Image buffers

An image buffer is an area of memory where a processor or a GPU can store a part or the whole displayed image. The GPU reads this image buffer in the pace the processor has instructed and sends the color information to the DAC. The DAC then converts the information into an analog signal that is relayed to the monitor.

Digital circuits that generate pixel based 2D images can be divided into two categories based on which kind of an image buffer it uses for the generated image. As can earlier be seen in Figure 5.1, these buffers include the line and the frame buffer [adv84]. Both of these buffers have advantages and disadvantages associated with them.

Japanese game company Namco developed the first line buffer based graphics system for the arcade game Galaxian in 1980 [adv84]. The 8 and 16-bit videogame consoles used mainly this technology until the early 1990s. This buffer type is still used in some modern video signal processors like the YGV629 [yam08] from Yamaha and MB88F332 [fuj09] 'Indigo' from Fujitsu. It needs less memory than the frame buffer and can be implemented in onchip-memories, but it also makes the generation of the image harder and more restricted.

A frame buffer which stores the entire image, makes the image generation easier and less restricted than with a line buffer. But unlike the line buffer it cannot be practically implemented in a SoC and needs an external memory. This makes it the more expensive approach.

Nowadays memories are fairly cheap and modern day GPUs use mainly frame buffers to generate the displayed image. More specifically they usually use double buffering to achieve a clean looking moving image. A GPU implementing double buffering writes the next displayed image to one buffer while the image currently being displayed is read from the other buffer. These two buffers are then switched as the next image is started to be displayed. The same kind of double buffering can also be used with line buffers.

Image size directly defines the needed line or frame buffer size. Table 5.1 shows the double buffered line and frame buffer sizes for four common video standards [vga, atsc08, ebu10].

Table 5.1 - Line and frame buffer sizes

Video standard	Image dimensions	bytes/pixel	line buffer size* (bytes)	frame buffer size* (bytes)	proportional frame buffer size
VGA	640*480	4	5 120	2 457 600	480
SVGA	800*600	4	6 400	3 840 000	600
HD 720	1280*720	4	10 240	7 372 800	720
HD 1080	1920*1080	4	15 360	16 588 800	1 080

* double buffered

For example a VGA image is comprised of 0.3 mega pixels in total. And if one pixel is described with three color values and one transparency value with each value requiring eight bits of storage, a double buffered frame buffer would need 2400 KB of space. A double buffered line buffer would in the other hand require only 5 KB.

5.3. 2D graphics elements

Hardware accelerated 2D graphics systems use mainly two different kinds of graphics elements. These are the sprites and the tile arrays.

5.3.1. Sprites

Sprites are rectangular shaped images or parts of them, which the GPU writes into the image buffer based on the instructions made by the controlling processor [eck01]. Old videogame consoles usually had sprites which size was limited [nes04, snes04, gen98]. Their size was considerably smaller than in the modern 3D GPUs, although their dimensions were similarly powers of two in size. The size for example could have been 8x8 or 64x64. Figure 5.3 shows a character made of 23 8x8 sprites.

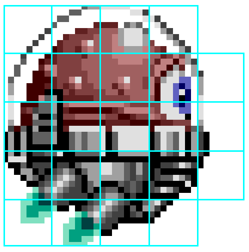


Figure 5.3 - A character made of 8x8 sprites

5.3.2. Tiles

Like sprites, tiles can for example be 8x8 pixels in size, but unlike sprites they cannot be placed in arbitrary coordinates [nes04]. They are placed in an orderly array like the one depicted in Figure 5.4.

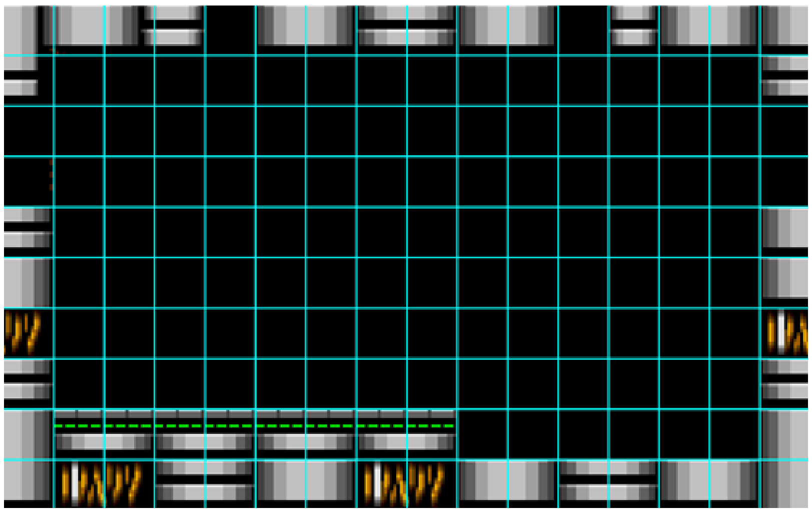


Figure 5.4 - A 16x10 tile array made of 16x16 tiles

The figure has a tile array made of 16x16 tiles in a formation that has 16 tiles in the horizontal and 10 tiles in the vertical plane. As can be seen in the figure they are commonly used as a background graphic.

5.4. Graphics memory, etc.

The graphics memory is where all the necessary graphical building blocks reside. These building blocks are used to create the image according to the instructions held in the scene memory. The instructions include information about the position, color palette and other properties of each of the used building blocks. Blitter is the component which reads graphics data from the graphics memory according to the instructions on the scene memory. This data or pixels are then written to one of the two different buffers which feeds the video DAC. The analog video signal is then transmitted to the monitor which displays the generated image.

5.5. Video signals

The image displayed on a monochrome cathode ray tube (CRT) monitor is physically drawn by sweeping an electron beam across the surface of the monitor from left to right [tpu]. An illustration how this is done is given in Figure 5.5.

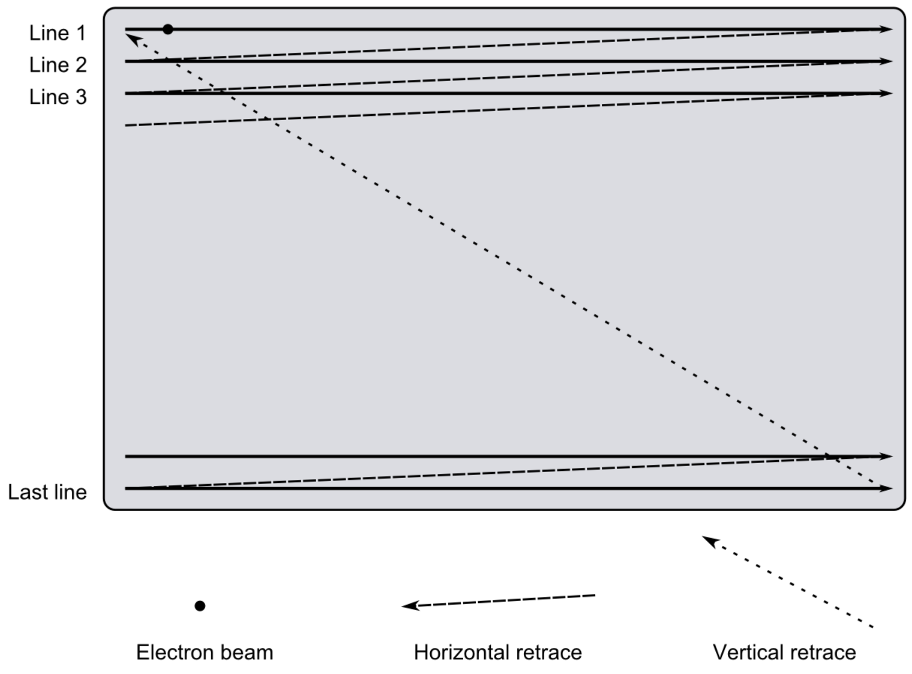


Figure 5.5 - Image displayed on a CRT monitor

When the beam sweeps the surface, it energizes specific pixels one line at a time. As the end of a line or far right-hand side of the screen is reached the beam is turned off and it is moved to the beginning of the next line. After the entire screen is swept like this, the beam is turned off for last time for that frame and it is returned to the beginning of the first line on the screen. This is then repeated for multiple consecutive frames to form a moving image.

Three analog signals are needed to control the intensity and position of the electron beam on a monitor. These signals include the video luminance, horizontal synchronisation and vertical synchronization signals. The levels of these signals through time are represented in Figure 5.6 [jav]. The upper graph shows the luminance and horizontal synchronization signals for one vertical line whereas the lower graph shows the luminance and vertical synchronization signal for a single frame.

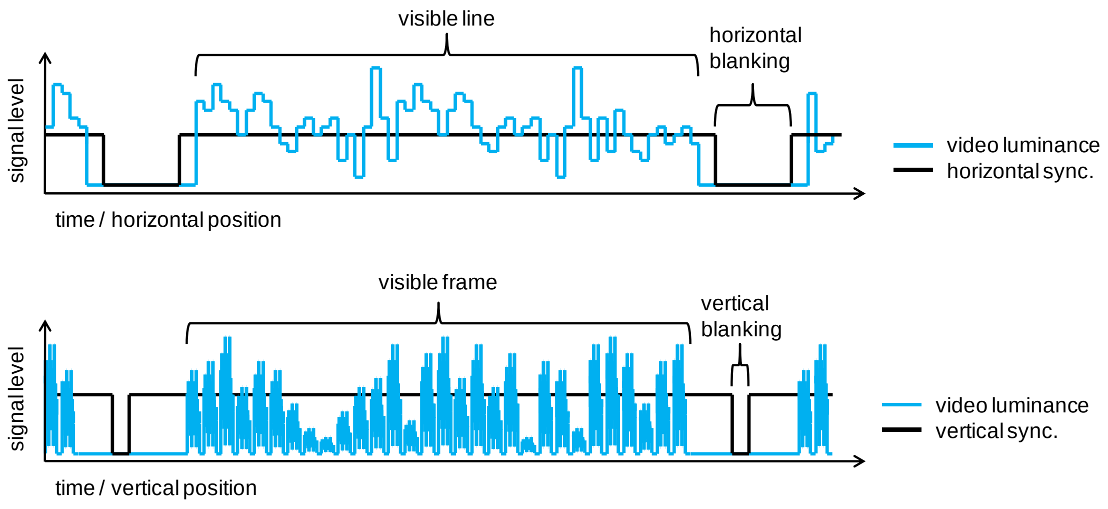


Figure 5.6 - Monochrome video signals

The video luminance signal is responsible for controlling the intensity of the beam which in turn creates the visible image. The horizontal and vertical synchronization signals control the horizontal and vertical retrace of the electron beam. More specifically the retrace of the beam happens during the horizontal and vertical blanking periods shown in the figure.

To display a color image, three signals are required for the three main colors which include the red, green and blue colors.

6. IP information register case study

This chapter is dedicated to examining the IP information registers in a practical environment. To do this, a test setup is devised with a 2D graphics system as the SUT and a data gathering system attached to it. These two systems are synthesized to the FPGA chip on the DE2 board [de2] as can be seen in Figure 6.1. The setup is used to examine three usages for the IP information registers which include the identification of IIR enabled IP blocks as well as the run-time gathering of information to verify and optimize the target system

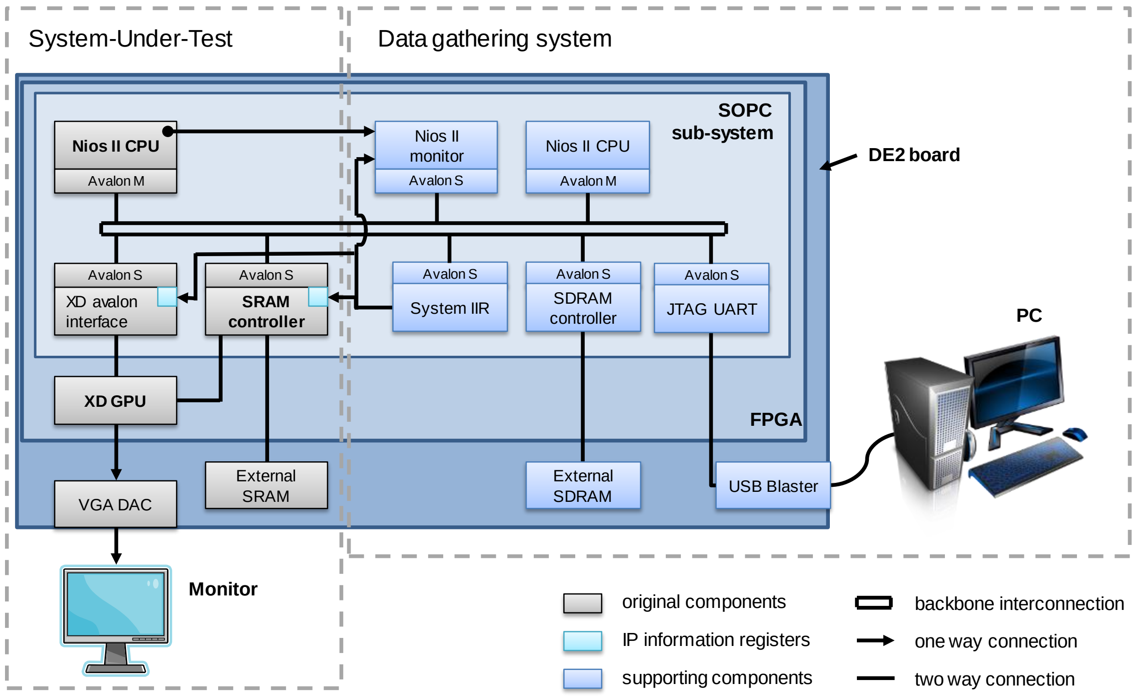


Figure 6.1 - Case study test setup

The main components of the SUT, or the 2D graphics system, are a Nios II processor, an XD GPU [arvio11], a shared SRAM controller, which is used by the processor and the GPU, and an Avalon bus to connect the components.

The data gathering system also includes a Nios II processor which is connected to a SDRAM controller and a JTAG UART. Additional supporting blocks include an external Nios II monitor which is used to probe the processor's activity on the SUT and a system IIR. Both of these systems are operated by a clock running at 100 MHz.

Before the IIR usages are examined in detail, the physical hardware and the IP blocks used in the FPGA are described in the next three sub-chapters.

6.1. Test setup hardware

6.1.1. DE2 board

Terasic's DE2 board is the main component in the test setup. It is the successor for the DE1 board. The DE2 board is a versatile and low cost development board targeted at students who study digital and computer systems and embedded programming. It can be used to illustrate fundamental concepts but also prototype advanced designs in multimedia, storage and networking applications.

At its core, it has a Cyclone II FPGA chip [cyc05] with over 33 thousand logic elements and and almost 500 thousand bits of on-chip memory. The board also incorporates four different kinds of memories which include a SRAM, a SDRAM and two flash memory chips. It also has an extensive range of digital and analog I/O connections which enable its use in various applications [de2]. The board layout and the main components can be seen in Figure 6.2.

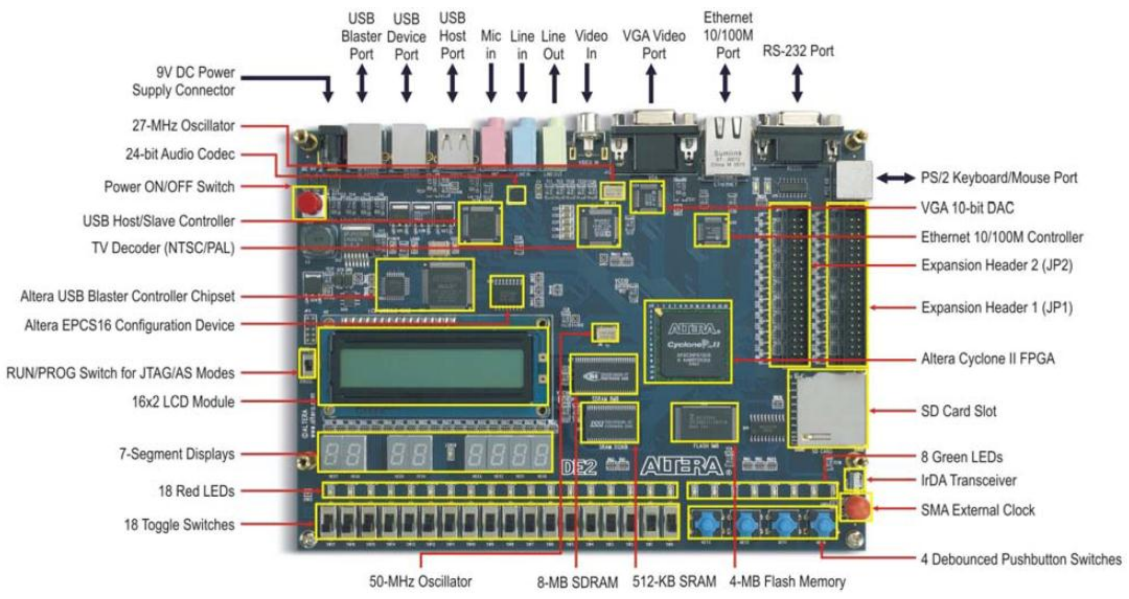


Figure 6.2 - DE2 development and education board

At the time of writing this thesis, the price of the board for commercial use was $495 and $269 for academic purposes [de2]. The lower price for academic purposes is enabled by Altera including the board in its university program.

6.1.2. Samsung SyncMaster 957MB monitor

An old 19 inch Samsung CRT monitor is connected to the DE2 board. The monitor is used to verify that the 2D graphics system is working properly while data is gathered from the IP information registers.

6.2. Altera IPs used in the test setup

6.2.1. Nios II processor

The Nios II [nios11] processor is Altera's own configurable 32-bit soft-core processor which is the successor to Altera's Nios processor. The older processor could be synthesized either as a 16 or 32-bit processor. Both of these processors are soft-core, which means they have not got a fixed netlist and can be targeted to any FPGA made by Altera. The Nios II processor is also configurable so that it can be customized to the target system by adding or removing features from it.

The Nios II processor has three cores to choose from. These include the fast, standard and the economy core from which the fast has the highest and economy the lowest performance.

Two Nios II processors are used in this case study. The first processor, which uses the fast core and has 4 kByte data and instruction caches, is used on the 2D graphics system to run a graphics demo. The second processor, which uses the economy core with no caches, is used on the data gathering unit to examine the IIR enabled IP blocks.

6.2.2. SDRAM controller

A SDRAM controller connected to an 8MB SDRAM chip [icsi00] is used in the data gathering system. It is used as program and data memory for the second processor as well as to store the data from XD GPU's log register.

6.2.3. FLASH controller

An Altera provided CFI FLASH controller connected to a 4MB FLASH chip [spa05] is used as boot memory for the two processors. It also contains the graphical elements used with the XD GPU.

6.3. IPs designed for the test setup

All of the IPs designed for the test setup are fitted with IP information registers.

6.3.1. Nios II monitor

As the name suggests, the Nios II monitor is used to probe the activity of a Nios II processor. And as was already mentioned in chapter four, the source code of the Nios II processor is encrypted. This leads to the reality that the information registers cannot be placed inside the processor and the monitor has to be external.

The register layout for the monitor itself can be seen in Table 6.1. Unused registers are marked with a dash. The monitor has all the required IP information registers as well as eight optional registers, of which three are used for creating logs of the processor's activity.

Table 6.1 - Nios II monitor information register layout

Offset	R/W/C	Name	Description
0x00	R	IIR1 header	A 32-bit header that inverts itself on reads
0x01	R	IIR type	value: 101
0x02	-	-	-
0x03	R/W	IP reset	Can be used to reset the Nios II CPU
0x04	R	Instance number	value: 0
0x05	-	-	-
0x06	R	VLNV	vendor:TUT, library:TUT, name: Nios II monitor, version: 0.2
…	R	...	
0x0C	R	VLNV	
0x0D	-	-	-
0x0E	R/W	Log register pointer	
0x0F	R/C	Reads	Number of reads done to the Nios II monitor
0x10	R/C	Writes	Number of writes done to the Nios II monitor
0x11	R/C	Faulty reads	Number of reads with faulty address
0x12	R/C	Faulty writes	Number of writes with faulty address
0x13	R/C	CPU longest wait	Longest time in cycles the CPU has waited for memory access
0x14	R/C	CPU mem stall log	Log storing the events when the CPU was stalled waiting for memory access
0x15	R/C	CPU I rd log	Log storing the events when the CPU was reading instructions from external memory
0x16	R/C	CPU D rw log	Log storing the events when the CPU was writing or reading data to or from external memory

These types of log registers are further described in the next sub-chapter.

6.3.1.1. Log registers

There are two main types for the log registers. The first one is an event based in which one event includes a timestamp and the accompanying data for the event. The accompanying data can be excluded if the event log register stores only the timestamps of an event and the event can be identified implicitly. The second one is a periodic log with a custom sized log interval and only the data is stored for the intervals. For example an error log is event based and a utilization log is periodic.

The logging functionality can be controlled by writing specific words to specific registers. The logging can be disabled, enabled, cleared and the log mode can be set. The log register commands are described in Table 6.2.

Table 6.2 - Log register commands

Code	Command	Description
0x00	disable logging	-
0x01	enable logging	-
0x02	clear the log	-
0x03	disable automatic clear	Disables automatic clearing of log contents after a full read
0x04	enable automatic clear	Enables automatic clearing of log contents after a full read
0x05	enable linear log mode	-
0x06	enable fifo log mode	-
0x07	read log memory	Next reads will read the log's contents
0x08	read log status	Next reads will read the log's status

Table 6.3 - Log status structure

b31..b11	b10..b03*	b02	b01	b00
-	log fill amount	log overflow	log auto clear	log enable

* log size = 128 words

The log mode of these registers can be set to either a linear mode or a FIFO mode. In the linear mode, the memory reserved for the log is filled from the beginning and when the end has been reached the logging is stopped. In the FIFO mode, the memory is used as a FIFO so when the whole log memory is filled, the values at the beginning of the log will be replaced with new ones.

There is a possibility that a component reading an event based log can miss data by first reading the entire log and after this manually clearing it. This happens because additional events can be written to the log during the clock cycles after the log is read and it is cleared. To ensure that no logged data is lost by clearing the log manually, an automatic clear after a full read can be enabled.

The content for an event based log is described in Table 6.4. The example logr shown in the table has eight words of data for each event and there are 32 events in the log. The contents of a periodic log described in Table 6.5 .

Table 6.4 - Example of event based log content

Event	Word	Content
0	0	timestamp 0
	1	data 0 [1/8]
	2	data 0 [2/8]
	…	…
	8	data 0 [8/8]
1	9	timestamp 1
	10	data 1 [1/8]
	…	…
	16	data 1 [8/8]
…	…	…
31	256	data 31 [8/8]
-	257	

Table 6.5 - Example of periodic log content

Period	Word	Content
0	0	data 0 [1/8]
	1	data 0 [2/8]
	…	…
	7	data 0 [8/8]
1	8	data 1 [1/8]
	…	…
	15	data 1 [8/8]
…	…	…
31	255	data 31 [8/8]
-	256	

6.3.2. XD GPU

The XD GPU is a VGA display controller and a 2D graphics processing unit (GPU) which is still early in development [arvio11]. It is inspired by old video game consoles and their pixel processing units (PPU), and has a fixed resolution of 640x480. Unlike today's GPUs, the PPUs did not have frame buffers and the graphics were generated at the same time the pixel line was displayed on the television or monitor.

XD GPU uses only two line buffers to store the generated graphics. The line buffers are used the same way as double buffered frame buffers are used. The next line of graphics is generated at the same time as the previous one is being displayed.

Information registers were added to the GPU after its design. Besides the required registers, the IIR block has registers to log the activity of the CPU's and GPU's access to the shared SRAM. This log functionality is used in the third part of the case study. The register layout for the IIR block can be seen in Table 6.6.

Table 6.6 - XD GPU information register layout

Offset	R/W/C	Name	Description
0x00	R	IIR1 header	A 32-bit header that inverts itself on reads
0x01	R	IIR type	binary value: 110
0x02	R	IP offset	value: - 0x1800
0x03	R/W	IP reset	Can be used to reset the XD GPU
0x04	R	Instance number	value: 0
0x05	-	-	-
0x06	R	VLNV	vendor: liHard, library: gfx, name: xd_gpu, version: 0.2
…	R	...	
0x0B	R	VLNV	
0x0C	-	-	-
0x0D	R	Frames from reset	Incrementing frame number starting from exit of reset state on the main FSM (not system or IP reset)
0x0E	R/C	Faulty writes	Number of writes with faulty address
0x0F	R/C	Faulty reads	Number of reads with faulty address
0x10	R/W	Frames to capture	Frames to capture on the log
0x11	R/W	Log mem. begin	Start address for the log
0x12	R/W	Log mem. end	End address for the log
0x13	R/W	Log status	bit 0: log active, bit 1: fifo overrun

6.3.3. SRAM controller

As the usage of SRAM chips is very straightforward, they really do not need fully fledged memory controllers for operation. But in the case of this 2D graphics system where two components have to have access to a 512 kB SRAM [issi01], a controller was implemented which has the logic to arbitrate the access rights to the memory.

There is a SOPC memory controller for the SRAM chip on the DE2 provided by Terasic, but this “controller” can only use the chip at 50 MHz which is half of the full 100 MHz frequency the chip can operate under. The chip can operate at full speed by adding extra timing information for the Quartus II fitter and generating a 5 ns write pulse.

Memory controllers normally do not have registers which to configure and have only one address space which spans the whole of the underlying memory. Therefore it was logical to house the information registers for the controller on a whole new Avalon slave port, while retaining the other port solely for the underlying memory. The layout for the information registers can be seen in Table 6.7.

Table 6.7 - SRAM information register layout

Offset	R/W/C	Name	Description
0x00	R	IIR1 header	A 32-bit header that inverts itself on reads
0x01	R	IIR type	binary value: 000
0x02	-	-	-
0x03	-	-	-
0x04	R	Instance number	value: 0
0x05	-	-	-
0x06	R	VLNV	vendor: liHard, library: storage, name: sram_2x_access, version: 0.2
…	R	...	
0x0E	R	VLNV	
0x0F	-	-	-
0x10	R/C	Mem. writes	Writes done to the SRAM (lower bytes of value)
0x11	R/C	Mem. writes	Writes done to the SRAM (upper bytes of value)
0x12	R/C	Mem. reads	Reads done to the SRAM (lower bytes of value)
0x13	R/C	Mem. reads	Reads done to the SRAM (upper bytes of value)

6.4. Test setup software

The test setup has four different programs, of which maximum of two are run at a time. The program run on the 2D graphics system is a simple graphics demo which uses about 50% of the fill rate on the XD GPU. The demo utilises two tile arrays and 256 moving sprites. An artificial work load was also added to the demo to induce data cache flushing and loading between frames. A screenshot of the demo can be seen in Figure 6.3.

Figure 6.3 - Screenshot of XD GPU demo

The three other programs are run by the processor on the data gathering unit and are used to gather the required data for the three different parts of the case study. Additional information about these programs can be read in the case study that is described in the next sub-chapters.

6.5. Examining IP information register usages

This case study is divided in to three parts where the main three usages of the IP information registers are examined. The first part examines the use of the IIR1 header register which can be used to identify IP blocks by scanning a CPU's address space. The second part peers into how a log register can be used to discover SRAM timing problems. And finally a group of log registers are examined which store the CPU's and the XD GPU's usage patterns of the shared SRAM, which is then used to optimize the usage of a shared memory.

The test setup for all of these parts is wired so that the Nios II monitor's IP reset register is set to one at system reset. And as this IP reset controls the reset state of the first processor, the processor is left in a reset state until the second processor frees it by writing a zero to the Nios II monitor's reset register. This leaves the CPU on the data gathering system to control the examination process, while the first CPU just provides a practical usage case which to examine.

6.5.1. Scanning for IIR enabled IP blocks

This usage part is the simplest and demonstrates the use of the identification features on the IIR blocks. The first processor is left in a reset state as a program is run on the second one which scans its full address space for IIR enabled IP blocks. After the program has scanned the address space, it confirms that the program being run matches the system it is run on. This is done by comparing the VLNV information of the system which is stored in software with the one stored in hardware. The hardware counterpart is found on the system IIR as is described in the end of chapter four. Full address space for the second processor is described in Table 6.8.

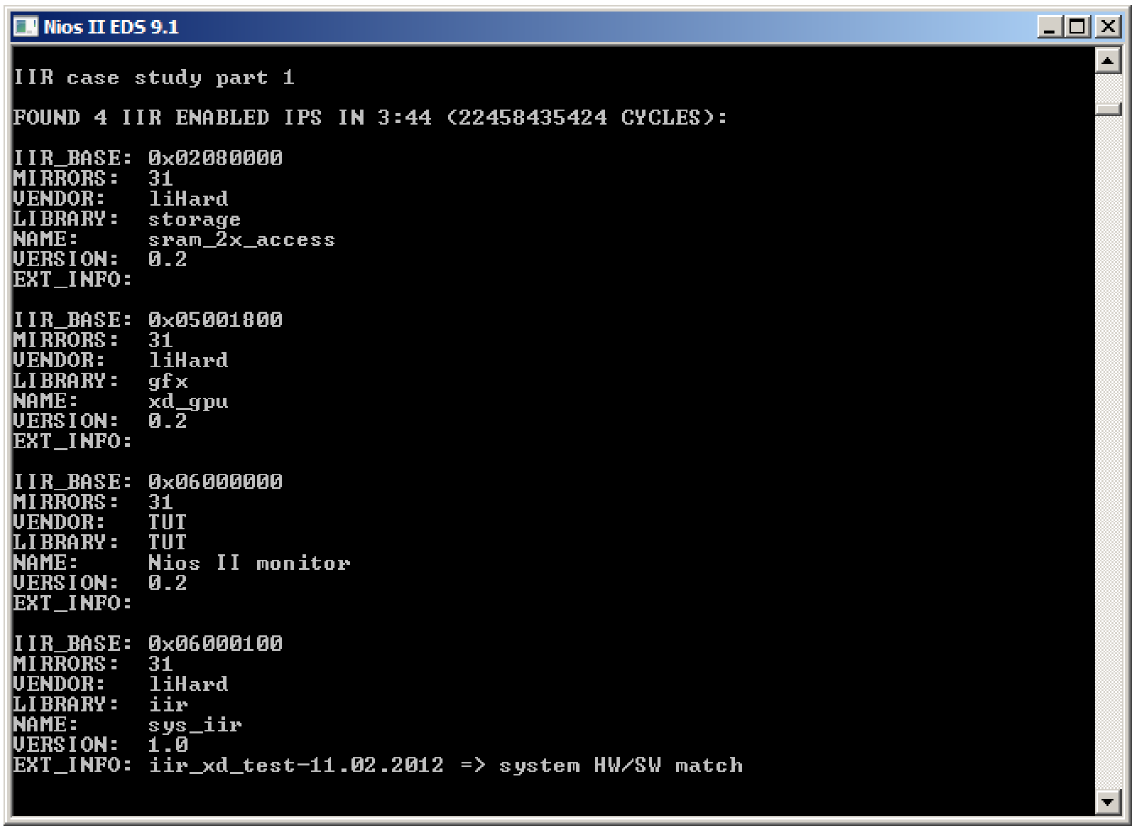
Table 6.8 - Address space for the data gathering unit's Nios II CPU

address	Name	Description
0x00001800...0x00001FFF	JTAG debug module	Used to debug programs
0x01000000…0x010001FF	onchip memory	Used as interrupt memory
0x01000200…0x0100023F	Timer	64-bit cycle accurate counter
0x01000240…0x01000247	JTAG UART	Used for debug printing
0x02080000…0x0208007F	SRAM IIR port	Port to access SRAM IIR block
0x03000000…0x037FFFFF	8MB SDRAM	Main memory for second CPU
0x04000000…0x053FFFFF	4MB FLASH memory	Boot memory for both CPUs
0x05000000…0x05003FFF	XD GPU	2D GPU
0x06000000…0x060000FF	Nios II monitor	External IIR block for first CPU
0x06000100…0x0600017F	System IIR	Holds information for the system

As this address space is scanned, one would except to find four IIR blocks. These are the ones in the XD GPU, the SRAM controller, the Nios II monitor and the system IIR. But Altera's Avalon bus mirrors some addresses, and as a result each of the four IIR blocks are found in the address space 32 times. To counter this, the program searching for the blocks keeps track of their instance numbers and discards any mirrors.

The address space for a Nios II processor is 2GB in size. The scan for this particular processor lasts about 3 minutes and 44 seconds, which in doubly is affected by the large address space and the fact that slow components like the FLASH memory are mirrored several times.

A short printout of the program can be seen in Figure 6.4.



```
IIR case study part 1

FOUND 4 IIR ENABLED IPS IN 3:44 (22458435424 CYCLES):

IIR_BASE: 0x02080000
MIRRORS:  31
VENDOR:   liHard
LIBRARY:  storage
NAME:     sram_2x_access
VERSION:  0.2
EXT_INFO:

IIR_BASE: 0x05001800
MIRRORS:  31
VENDOR:   liHard
LIBRARY:  gfx
NAME:     xd_gpu
VERSION:  0.2
EXT_INFO:

IIR_BASE: 0x06000000
MIRRORS:  31
VENDOR:   TUT
LIBRARY:  TUT
NAME:     Nios II monitor
VERSION:  0.2
EXT_INFO:

IIR_BASE: 0x06000100
MIRRORS:  31
VENDOR:   liHard
LIBRARY:  iir
NAME:     sys_iir
VERSION:  1.0
EXT_INFO: iir_xd_test-11.02.2012 => system HW/SW match
```

Figure 6.4 - Case study part 1 printout

This usage part clearly shows that even with an interconnect like Avalon, which by its design mirrors parts of its address space, can be successfully scanned for IIR blocks without crashing the scanning program.

6.5.2. Determining memory related timing problems

As the verification of the different individual components in the system has been earlier made by simulating them on test benches in ModelSim, the system as a whole still has to be verified. This verification can be aided by adding IP information registers which monitor and log the events in the system and its components at run time. These logs can be later analysed for any error events.

For example, the operation of a SRAM chip has to be monitored on an actual physical system in a real environment so that possible timing related problems can be sorted out. If there are timing problems with the SRAM and some of the accesses done by the CPU are not completed correctly, the program run by the CPU will most likely crash. The developer can observe the crash by seeing that the output for example coming from a JTAG UART or, like in the 2D graphics system, from the monitor connected to the DE2 board has stopped or that it is corrupted. The crash event can be seen in greater detail by using information registers to monitor certain signals on the FPGA.

If the CPU has crashed, the program execution might jump to a random memory location which does not contain any code. If this happens, it can safely be said that the occurrence of this abnormality was likely caused by a memory error. To see if a soft core CPU is reading instructions from an incorrect memory address, the program counter of the CPU must be inspected. But as the HDL source code of the Nios II processor is encrypted, the only way to directly monitor the operation of the program counter is by using the SignalTap II logic analyzer found in the Quartus II software. As this analyzer is run outside of the FPGA, the program counter address has to be monitored indirectly outside of the CPU. Fortunately the SOPC system's HDL file is not encrypted, so the filling of the instruction cache and therefore the program counter addresses can be seen by monitoring the instruction master port's read address.

Additionally, as the information register to monitor this cannot be placed on the Nios II CPU, an external monitor block for the CPU was made. The use of this monitor is further explored in the next sub-chapter of this thesis.

There are also other means to monitor the operation of the CPU. Initially, there were errors in SRAM timing which crashed the CPU. The crash was detected by monitoring cache fillings and flushes which should go on and on all the time. If the filling and flushing has stopped it can be safely said that the program has crashed. A watchdog counter can be implemented in an information register that counts up when there is not any filling or flushing of the caches and it can be reset to zero when there is. So if the counter reaches a predetermined high value, it can be said that the program has crashed and therefore there can be a problem with the SRAM timings. This predetermined value can be chosen based on the external memory usage pattern of the cache.

The filling and flushing of the caches must be determined by monitoring the data and instruction master ports read and write signals. If no direct memory accesses to the SRAM are made, the only accesses through these two master ports are the filling and flushing of the caches.

6.5.3. Verifying SRAM timings

The second part of this case study examines the use of the aforementioned log registers which store the access pattern and addresses of a processor's instruction port. In this part, both of the test setup's processors are used. The one on the 2D graphics system is used to run a graphics demo, while the second processor is used to monitor the first one with the Nios II monitor.

Certain assignments on the Quartus project synthesising the system were disabled to induce timing problems with the first processor's access to the SRAM. This was done to examine this particular usage case relating to timing problems.

An illustration visualizing the sample points used with the Nios II monitor is presented in Figure 6.5. The simplified illustration shows the internal structure of the Nios II CPU, the SRAM controller and the external monitor. The signals between the sample points and the monitor were able to be created simply by editing the generated top level verilog file of the SOPC sub-system.

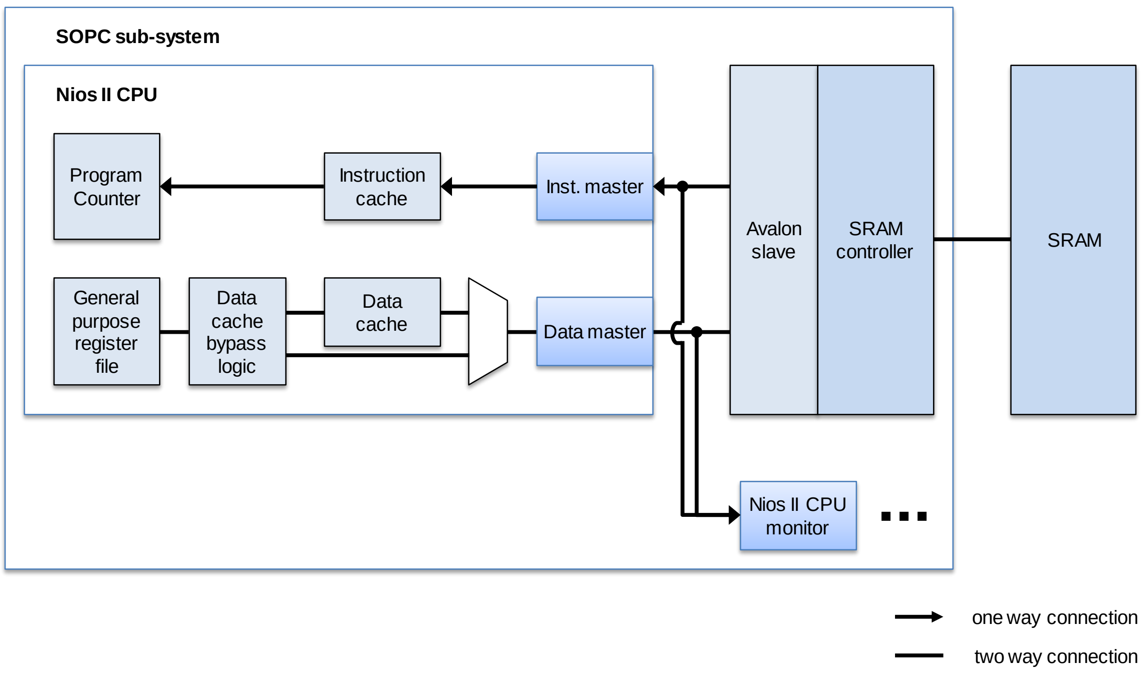


Figure 6.5 - Monitoring the Nios II CPU

When the second CPU is started, it first enables the log register for monitoring and after that releases the first CPU to execute its program. But as the SRAM is induced with artificial timing problems, the graphics demo visibly crashes. This can be examined in further detail by inspecting the log data. A part of the log at the crash time is presented in Table 6.9.

Table 6.9 - Log data during crash

cycles from reset	cpu inst. read	cpu inst. addr
21 387 192	1	0x20111B4
21 387 194	1	0x20111B8
21 387 196	1	0x20111BC
21 387 198	0	0x20111A0
21 387 202	1	0x0000004
21 387 203	1	0x0000008
21 387 204	1	0x000000C
21 387 205	1	0x0000010
21 387 206	1	0x0000014
21 387 207	1	0x0000018
21 387 208	1	0x000001C
21 387 209	1	0x0000000
21 387 210	0	0x0000004
21 387 245	1	0x0000020
21 387 246	1	0x0000024
21 387 247	1	0x0000028
21 387 248	1	0x000002C
21 387 249	1	0x0000030
21 387 250	1	0x0000034
21 387 251	1	0x0000038
21 387 252	1	0x000003C

The address range from the SRAM, which is used by the CPU, starts from 0x02000000 and ends at 0x0202FFFF. Additionally, the instruction section of the program (.text) spans from 0x02000000 to 0x02011CC0.

Based on the latter address range, it can be seen from the log that the CPU was filling its instruction cache correctly until 21 387 196 cycles from system reset. But after 21 387 202 cycles from reset it started filling the cache from an address range that does not contain any instructions for the CPU.

Based on this observation, it can be said that the CPU possibly read invalid data around this time and its execution went off track.

6.5.4. Sharing an external memory

Similarly to modern CPUs, the Nios II processor has the option to include a data and an instruction cache. The functionality of these caches involves storing frequently used instructions and data in the caches so that the CPU does not always need to access the slow external memory. Consequently, the CPU is not using the external memory all the time and another component can use this extra bandwidth. In fact, the first version of the GPU must be granted access to the external memory every time it requests it. The problem with this arrangement is that while the GPU can operate at maximum speed, the CPU has to wait extensive time periods to get access.

A simplified illustration showing an example of the accesses by these two components can be seen in Figure 6.6. Instead of showing a one dimensional array of timestamps and the corresponding values, the figure below is constructed to show how the accesses relate to the GPU's image shown on the monitor. The earliest accesses are on the upper left corner and the timeline continues the same way an electron beam travels trough a screen surface, which was previously shown in Figure 5.5. The area in time where the visible frame is located, is marked with a dark blue background and ranges from 0 to 639 lines in the horizontal axis and 0 to 479 on the vertical axis. Likewise, the horizontal blanking period is marked with a light blue background while the vertical blanking period with a white background. The GPU can fill its line buffers during the visible frame and the horizontal blanking period.

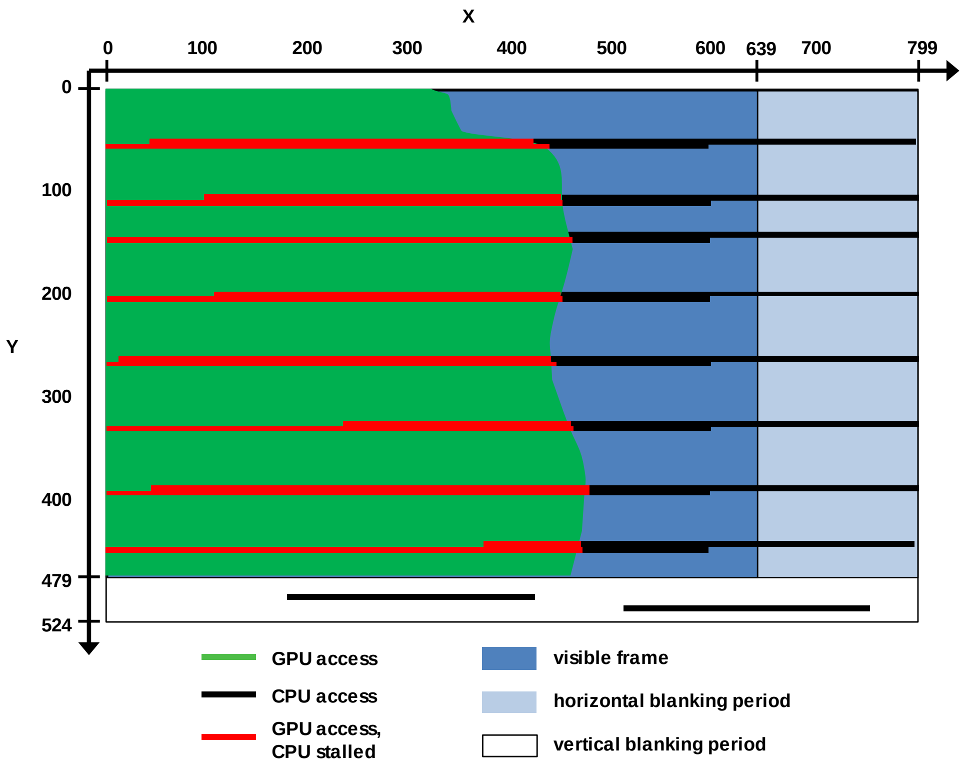


Figure 6.6 - GPU and CPU memory usage pattern, GPU has top priority

The CPU's accesses, shown in the figure with black stripes, consist of mainly the CPU filling the caches from or flushing them to the external memory. Some control data designed for the XD GPU is also written directly to the memory by the CPU.

The CPU's and the GPU's accesses to the external memory can be mapped using IP information registers. This functionality is examined in the next sub-chapter.

6.5.5. Acquiring usage patterns for the shared SRAM

This final part examines the possibilities of IP information registers to be used for optimizing IP blocks. As with the second part, the 2D graphics system is monitored by the CPU on the data gathering system. Although this time, the usage of the shared SRAM by the 2D graphics system's CPU and GPU is examined.

The memory usage patterns of the two components are recorded by event log registers present on the XD GPU. A simplified illustration showing how these information registers are connected to the surrounding system is depicted in Figure 6.7.

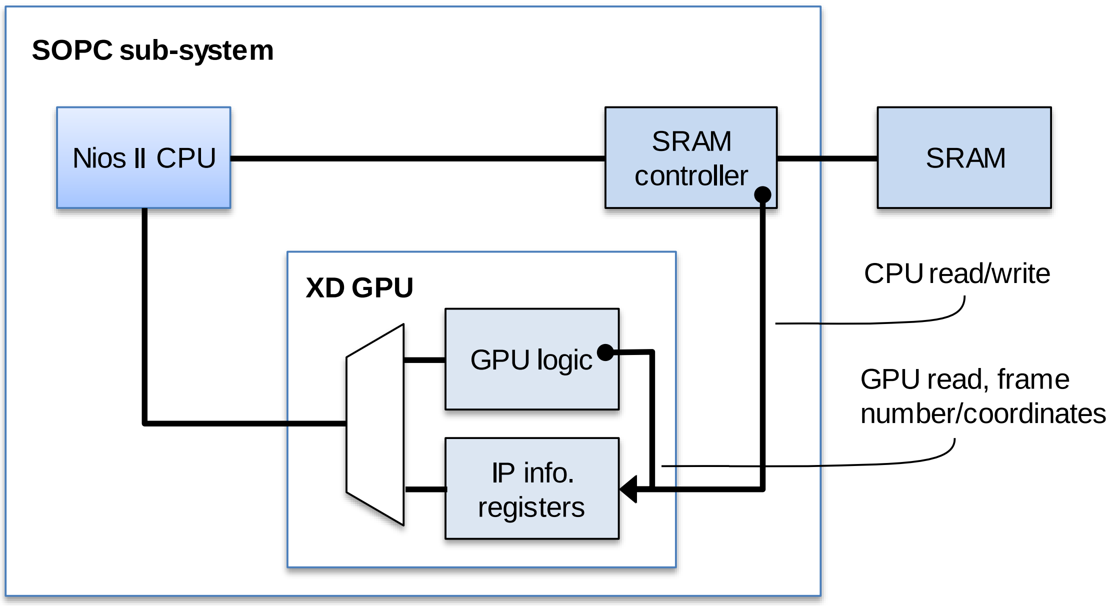


Figure 6.7 - XD GPU IP information register connections

Each of the events is stored in a 32-bit word. The data on the word consist of a timestamp of the event along with bits which tell whether the CPU or GPU was accessing the external memory at the time. Unlike in the previous part, the timestamp is acquired from the GPU's VGA signal generator and is stored either as a frame number or as the coordinates of the frame. The frame number is stored only at the beginning of a frame and otherwise the coordinates are stored. These two variant event formats can be seen in tables 6.10 and 6.11. No addresses are stored for the CPU's and the GPU's accesses. The GPU active bit is high when the GPU reads from the SRAM while the active bit for the CPU is high when the CPU reads from or writes to the SRAM. The GPU active bit is acquired from the GPU's internal logic, whereas the active bit for the CPU is acquired from the SRAM controller.

Table 6.10 - Shared SRAM log event format for frame start

bit	31	30..22	21..02	1	0
description	frame begin	-	frame number	cpu active	gpu active

Table 6.11 - Shared SRAM log event format for intra frame events

bit	31	30..24	23..13	12..11	10..02	1	0
description	frame begin	-	screen x	screen x cycle	fill y	cpu active	gpu active

Unlike in the previous part, the first CPU is released for operation at the start of the experiment and the monitoring CPU waits for two seconds before proceeding. The wait ensures that the XD demo run by the first processor is in full swing when the second CPU starts the monitoring.

The second CPU configures the logging functionality by writing the start and end addresses for the log memory and the logging starts when the amount of frames to capture is written to the specific register. Now as the logging logic is active, it checks for changes in the aforementioned activity signals. When the signals change, ie. an event occurs, the logging logic writes an event word into an internal fifo. The output of this fifo is constantly written to the specified address range on the external log memory.

The monitoring CPU then sits in a loop waiting for the activity bit on the log's status register to go down. After this happens, the CPU reads the entirety of the log and outputs a graphical presentation of the acquired data. The first full frame of the data can be seen on the next page in Figure 6.8.

This figure is divided into three rectangular areas, exactly like the previous example figure. As can be seen from the red color, the CPU on the 2D graphics system is stalled for prolonged time periods waiting for access to the external memory. This is the flushing and loading of the data cache that was described earlier in this chapter. But as the figure shows, there is spare time to be given to the CPU at these utilisation levels.

Other aspects of the XD GPU's operation can also be observed from the figure. Such as the regular spikes in GPU access times every 16 lines, which is induced by the filling of the tile buffers for the two tile arrays. It also can be seen that the timing for writing the coordinates of the sprites for the next frame is perfect. This is seen at the bottom end of the GPU filling area. Additionally, the GPU's brief access to the external memory before the next frame can be seen at the bottom of the figure. This is the time the GPU reads information pertaining to the next frame.

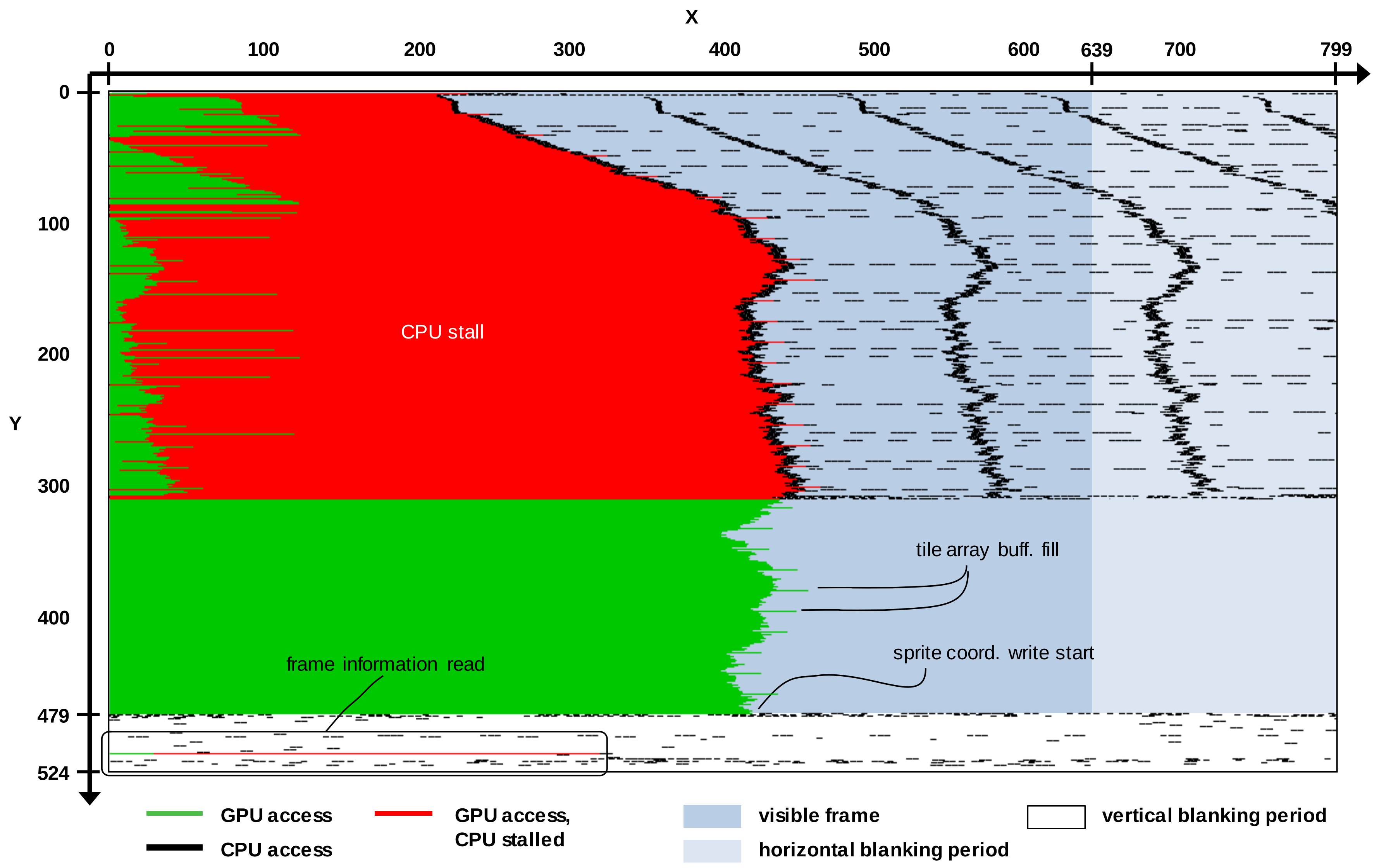


Figure 6.8 - CPU and GPU memory usage pattern, GPU has top priority

6.5.6. Optimizing the use of the shared SRAM

Most of the CPU's stalling time, which is shown for the frame on the previous page, can be avoided. For example, an internal memory containing 480 values could be embedded to the XD GPU. Additional logic would store the access time that was left unused for each of the visible lines for one frame. This information would then be used within the GPU's main FSM to determine the amount of cycles which could be given to the CPU on a given line. An example illustration showing how this would look like with the access times is given in Figure 6.9.

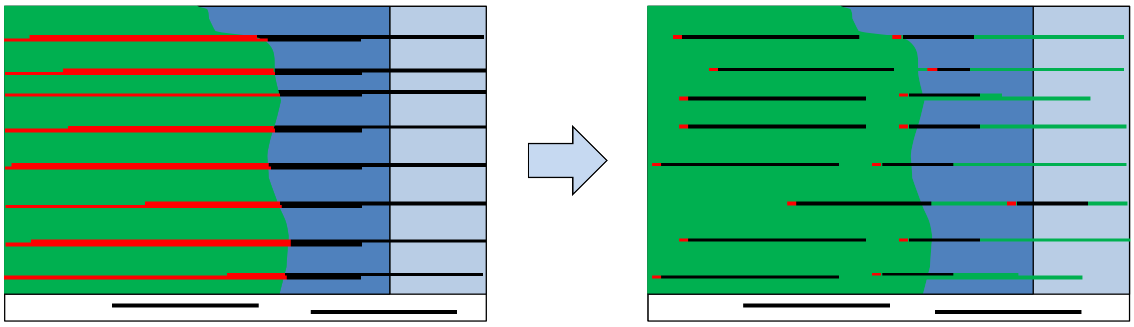


Figure 6.9 - Arbitrating access times

The left portion of the figure shows a simplified example how the access patterns look like with no access time arbitration by the GPU. After the additional arbitration logic is added, the patterns would look like the right portion of the figure. Notice that the CPU stalls (red) have reduced significantly.

For additional optimization, the CPU could be given access rights to the aforementioned memory so that it could grant itself more access time. This would be a good tool for the CPU in certain circumstances. If the CPU for example knew that the next frame differed from the last one and was significantly less taxing to the GPU, it could give itself more of the access time. The CPU could also grant itself more access time, if it needed it for processing power regardless of the graphics.

7. Conclusions

Based on the case study on this thesis, it is clear that IP information registers can provide many good services. A short summary of the case study and its parts is presented in Table 7.1. The table shows each of the parts' purpose, the number of IIR blocks used for that purpose as well as the logic elements consumed by the information registers for that purpose. The logic element counts in this last chapter were acquired from a single synthesis instance and will wary a little from different instances with different synthesis parameters.

Table 7.1 - Case study parts

Part	Purpose	IIR blocks	IIR le:s
1	IP recognition during boot, VLNV registers, SW and HW matching	4	1056
2	Log CPU's instruction address to detect timing errors	1	380
3	Log CPU's and GPU's access patterns to optimize memory usage	1	255

As can be seen from the table, the first part proved that the IIR1 header register can be used to identify IIR enabled IP blocks within a CPU's address space, as well as the system IIR can be used to verify that the correct software was used with the proper hardware. The latter of these functions serves to reduce the amount of things that can go wrong when developing IPs and SoCs.

The second part involved helping to verify the timing parameters for the SRAM by monitoring the 2D graphics system's CPU's instruction reading patterns. These patterns were specifically monitored during the programs crash to determine if the crash was caused by wrong or insufficient timing parameters.

The final part of the case study involved monitoring the CPU’s and the GPU’s access pattern to the shared SRAM. The log registers embedded to the GPU provided a detailed view for the accesses and a good way to utilize this knowledge for optimization was discussed. This part was so successful that the proposed optimization will be implemented in the next version of the XD GPU. The log registers will also be enhanced to include more data pertaining to the GPU’s functions relative to the access patterns.

7.1. HDL files

Although information registers provide a good service for the developer, they are relative simple to implement. This can be seen in the amount of code which had to be written for implementing the IIR blocks. The exact number of lightly commentated lines coded for these blocks can be seen in Table 7.2.

Table 7.2 - IIR block lines of code

File name	Lines of code
nios2_monitor.v	757
event_log.v	156
sram_2x_access.v	241
system_iir.v	166
iir_xd_gpu.v	470
TOTAL	**1790**

To put the IIR block in the XD GPU into perspective, Table 7.3 shows the code lines for the different source files used to construct the GPU. The IIR block stands at 470 lines of code or about 16% of total amount in the XD GPU.

Table 7.3 - XD GPU lines of code

File name	Lines of code	Percent of total
x2d_gpu.v	1623	56,1
x2d_gpu.h	37	1,3
x2d_gpu_if.v	55	1,9
vga_sync_gen.v	147	5,1
line_buf.v	203	7,0
tile_buf.v	44	1,5
alpha_blend_4x.v	53	1,8
alpha_blend_channel.v	38	1,3
alpha_blend_pixel.v	98	3,4
clut.v	74	2,6
dp_ram.v	51	1,8
iir_xd_gpu.v	470	16,2
TOTAL	**2893**	**100**

New IIR blocks can be constructed from older ones, which further reduces the work amount required for their implementation. Especially the general information registers are very similar in all of the IIR blocks reducing required effort.

7.2. Hardware resource usage

The logic resources consumed for the two systems and their IP blocks can be seen in Table 7.4. The total was 11557 logic cells, 56 onchip memories and 32 onchip multipliers. Average logic cell interconnect usage was at 15%.

Table 7.4 - IP block resource usage

Name	Logic cells	Onchip memory	Onchip multipliers (9x9)
Nios II CPU 0 core	2998	20	4
CPU 0 data master	297	-	-
CPU 0 inst. Master	132	-	-
CPU 0 interrupt memory	1	1	-
CPU 0 timer (32-bit)	154	-	-
Nios II CPU 1 core	783	2	-
CPU 1 data master	282	-	-
CPU 1 inst. Master	70	-	-
CPU 1 interrupt memory	1	1	-
CPU 1 timer (64-bit)	283	-	-
SDRAM controller	475	-	-
SRAM controller - orig. logic	70	-	-
SRAM controller - IIR block	300	-	-
JTAG UART	167	2	-
Key parallel input	10	-	-
Switch parallel input	18	-	-
Green led parallel output	16	-	-
Red led parallel output	20	-	-
Nios II monitor	681	2	-
XD GPU - original logic	3227	27	28
XD GPU - IIR block	585	1	-
System IIR	222	-	-
TOTAL	**11557**	**56**	**32**

The four IIR blocks in the system consumed a combined 1788 logic cells, which is about 5,4% of the available logic cells on the chip. Three onchip memories were also used in the IIR blocks. Additionally 2061 logic cells or about 6,2% was used for the CPU and its peripheral components on the monitoring system. The IIR blocks, the monitoring CPU and etc. contribute to 3849 logic cells or about 11,6% of the available resources. Of course the first and last part of the case study can be done with only one CPU along with the additional SDRAM memory. So the extra amount of logic cells stands at 2536 (7,6%) for these two parts and 3849 for the second one.

An illustration showing the resource usage for the FPGA on the DE2 board can be seen in Figure 7.1. The left part shows the resource usage for the IIR blocks highlighted in blue, whereas the right part does the same for the monitoring CPU and its peripheral components.

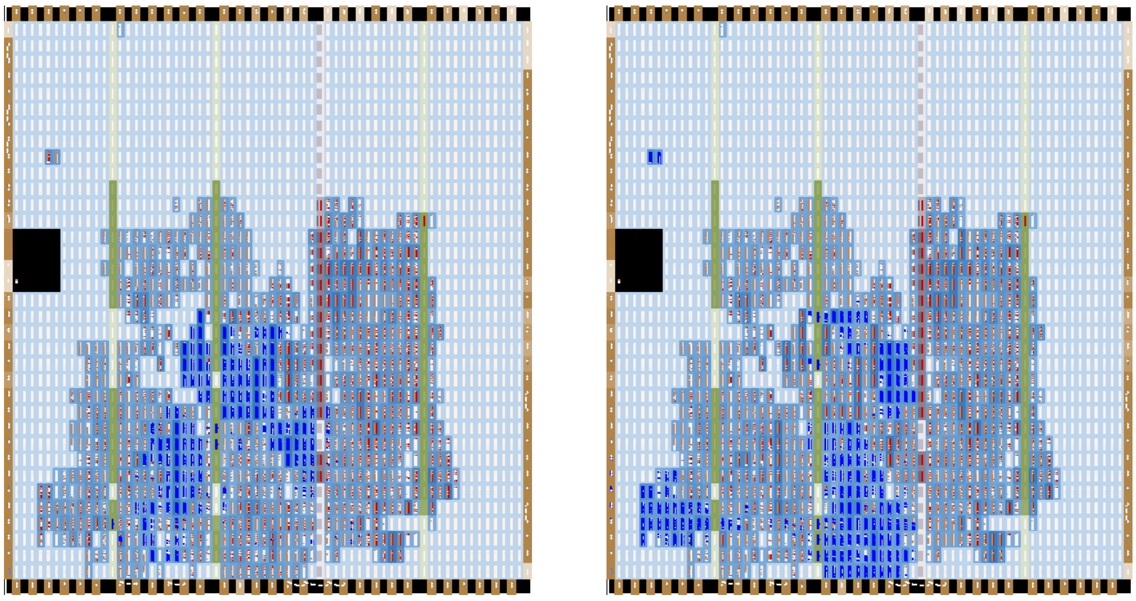


Figure 7.1 - Resource usage for the IIR blocks on the left, monitoring CPU, etc. on the right

For comparison, a performance monitor created by Lancaster et al. consumes 1655 look-up tables (LUT) and 1260 registers from a 98304 LUT/register FPGA [lan09]. This can be compared to the performance monitoring present in the XD GPU's IIR block, which consumes 300 LUTs and 276 registers.

7.3. Software files

The software part of the case study was fairly straightforward, which was reflected in amount of lines written for the Nios II processors. The amount of code written for the software source files can be seen in Table 7.5.

Table 7.5 - Software lines of code

file name	lines of code
main.c (XD demo)	260
main.c (IIR part 1)	59
main.c (IIR part 2)	71
main.c (IIR part 3)	71
ip_info_reg.h	228
ip_info_reg.c	516
misc.h	129
misc.c	261
TOTAL	**1595**

As can be seen from the table, the main.c files for the different IIR usage parts have only a small amount of code. This is due to the fact that the functions for using the information registers reside in the ip_info_reg.h and .c files. The functions ensure that the future use of the information registers is simple and straightforward, and any developer using the registers has not got much work to do to utilize them properly.

References

[atsc08] ATSC Standard: Video System Characteristics of AVC in the ATSC Digital Television System Document A/72 Part 1, Advanced Television Systems Committee Inc., 2008.

[ake08] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Realtime rendering 3rd edition, book, A K Peters Ltd, 2008, pp. 829-877.

[arria11] Arria II device handbook, Altera Corporation, [online], available: http://www.altera.com/literature/hb/arria-ii-gx/arria-ii-gx_handbook.pdf

[cyc05] Cyclone II Device Handbook, Altera Corporation, 2005.

[de2] DE2 Development and Education Board, Altera Corporation, [online], available: http://www.altera.com/education/univ/materials/boards/de2/unv-de2-board.html

[asoft12] Design software, Altera Corporation, [online], available: http://www.altera.com/products/software/sfw-index.jsp

[nios11] Nios II Processor Reference Handbook, Altera Corporation, May 2011, [online], available: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[sig11] Quartus II Handbook v11.1.0 Volume 3 Chapter 13, Altera Corporation, November 2011, pp. 1-72

[alt12] Altera Corporation's Technology Milestones, Altera corporation, [online], available http://www.altera.com/corporate/about_us/history/abt-history.html

[arvio11] J. Arvio, Tampere University of Technology, Bachelor of Science thesis, 2011, [in Finnish].

[tta11] Jani Boutellier, Transport-triggered processors, slideset, Computer Science and Engineering Laboratory, 2011, [online], available: http://tce.cs.tut.fi/slides/tta_uo.ppt

[gen98] Barry Cantin, John Hokanson Jr, Ken Arromdee, Sega Genesis/Mega drive faq version 1.5, 1998, [online], available: http://www.gamefaqs.com/genesis/916377-genesis/faqs/2756

[nes04] Patrick Diskin, Nintendo Entertainment System Documentation Version 1.0, Nintendo Entertainment System Architecture version 1.4, 2004, [online], available: http://nesdev.parodius.com/NESDoc.pdf

[ebu10] EBU – TECH 3299: High Definition(HD) Image Formats for Television Production, EBU, 2010.

[eck01] R. Eckert, Computer Graphics, course slides, Binghamton U., N.Y., 2001, [online], available:http://www.cs.binghamton.edu/~reckert/enginet.html

[ecos11] eCos User Guide, Free Software Foundation Inc., 2011, [online], available: http://ecos.sourceware.org/docs-latest/user-guide/ecos-user-guide.html

[vga] Video Signal Standards and Conversion Page, Epanorama.net, [online], available: http://www.epanorama.net/links/videosignal.html

[gaj00] D.D. Gajski, A.C.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, P. Bricaud, Essential issues for IP reuse, ASP-DAC, Jan. 2000, pp. 37-42.

[gsa] GSA's IP Subcommittee's Industry Baseline Working Group, Understanding the semiconductor Intellectual property (sip) business process, handbook, Global Semiconductor Alliance, [online], available: http://www.gsaglobal.org/resources/tools/docs/Handbook_Understanding_SIP_BusinessProcess.pdf

[ver95] IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE, 1995.

[vhd94] ANSI/IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual, IEEE, Piscataway, New Jersey, 1994.

[eth08] IEEE 802.3-2008, IEEE Standard for Information technology-Specific requirements - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications, IEEE, 2008.

[ipxact10] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows, IEEE Std 1685-2009, 2010, 360 pages.

[tpu] The cathode-ray tube (CRT), Integrated Publishing, [online], available: http://www.tpub.com/neets/book6/21e.htm.

[icsi00] IS42S16400 1M Words x 16 Bits x 4 Banks (64-Mbit) Synchronous Dynamic RAM, datasheet, Integrated Circuit Solution Inc, 2000.

[issi01] IS61LV25616 256k x 16 high speed asynchronous cmos static ram with 3.3v supply, datasheet, Integrated Silicon Solution Inc, 2001.

[kam11] Antti Kamppi, Lauri Matilainen, Joni-Matti Määttä, Erno Salminen, Timo D. Hämäläinen, Marko Hännikäinen, "Kactus2: Environment for Embedded Product Development Using IP-XACT and MCAPI", The 14th Euromicro Conference on Digital System Design (DSD), Oulu, Finland, August 31, 2011 - September 2, 2011, pp. 262-265.

[keat02] M. Keating and P. Bricaud, Reuse Methodology Manual, 3rd ed. Kluwer Academic Publishers, 2002.

[keut00] K. Keutzer, S. Malik, R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, “System-level design: Orthogonalization of concerns and platform-based design,”

IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, Dec. 2000.

[kos09] Tapio Koskinen, Metadata-based automated configuration of system-on-chip, Master of Science thesis, Tampere University of Techonology, 2009.

[lan09] Joseph M. Lancaster, Jeremy D. Buhler, and Roger D. Chamberlain, Efficient runtime performance monitoring of FPGA-based applications, SOCC 2009 IEEE International, 2009, pp. 23-28.

[micro12] Microteam Oy web page, [online], available: http://www.microteam.fi/

[model12] ModelSim – Advanced Simulation and debugging, Mentor Graphics, [online], available: http://model.com/

[nul06] Linda Null, Julia Lobur, The essentials of computer organization and architecture, Jones & Bartlett Learning, 2006, pp. 511.

[usb11] Universal Serial Bus Specification Revision 3.0, USB Implementers Forum, 2011, pp. 1.1.

[sal01] Erno Salminen, Interface design for multiple processors in a system-on-chip video encoder, Master of Science thesis, Tampere University of Techonology, 2001.

[sal11] Erno Salminen, Timo D. Hämäläinen, Marko Hännikäinen, "Applying IP-XACT for product data management", International Symposium on System-on-Chip, Tampere, Finland, October 31, 2011 - November 2, 2011, pp. 86-91.

[san07] A. Sangiovanni-Vincentelli, “Quo vadis, SLD? reasoning about the trends and challenges of system level design,” Proc. IEEE, vol. 95, no. 3, pp. 467–506, Mar. 2007.

[spa05] S29AL032D 32 Megabit CMOS 3.0 Volt-only Flash Memory, Spansion LLC, 2005.

[fun12] Funbase main page, Department of computer systems, Tampere University of Technology, [online], available: http://funbase.cs.tut.fi/

[tex02] Interface circuits for TIA/EIA-232, Design notes, Texas Instruments Inc, 2002.

[jav] Javier Valcarce, VGA Video Signal Format and Timing Specifications, Javier Valcarce's Personal Website, [online], available: http://www.javiervalcarce.eu/wiki/VGA_Video_Signal_Format_and_Timing_Specifications

[wag04] F.R. Wagner et al., Strategies for the integration of hardware and software IP components in embedded systems-on-chip, Integration, the VLSI Journal, September 2004, Vol. 37, Iss. 4, pp. 223-252.

[snes04] Jason Williams, SNES faq version 1.05, 2004, [online], available: http://www.gamefaqs.com/snes/916396-snes/faqs/31726

[adv84] Marshall C. Yovits, Thomas A. Defanti, Advances in Computers, volume 23, book, Academic Press Inc, 1984, pp. 116-117.