

# Institutionen för systemteknik

## Department of Electrical Engineering

### Examensarbete

## Design and Development of a Versatile Hardware Test Platform

Examensarbete utfört i Computer Engineering  
vid Tekniska högskolan vid Linköpings universitet  
av

**Eneas Puertas Kreusch**

LiTH-ISY-EX--2012/XXXX--SE

Linköping 2012



**Linköpings universitet**  
**TEKNISKA HÖGSKOLAN**



# **Design and Development of a Versatile Hardware Test Platform**

Examensarbete utfört i Computer Engineering  
vid Tekniska högskolan vid Linköpings universitet  
av

**Eneas Puertas Kreusch**


LiTH-ISY-EX--2012/XXXX--SE

Handledare: **Per Karlström**  
ISY, Linköpings universitet

Examinator: **Per Karlström**  
ISY, Linköpings universitet

Linköping, 23 augusti 2012



	<b>Avdelning, Institution</b> Division, Department  Computer Engineering Department of Electrical Engineering SE-581 83 Linköping	<b>Datum</b> Date  2012-08-23
---	--	--

<b>Språk</b> Language  <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English  <input type="checkbox"/> _____	<b>Rapporttyp</b> Report category  <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	<b>ISBN</b> _____  <b>ISRN</b> LiTH-ISY-EX--2012/XXXX--SE  <table style="width: 100%;"> <tr> <td style="width: 70%;"><b>Serietitel och serienummer</b></td> <td style="width: 30%;"><b>ISSN</b></td> </tr> <tr> <td>Title of series, numbering</td> <td>_____</td> </tr> </table>	<b>Serietitel och serienummer</b>	<b>ISSN</b>	Title of series, numbering	_____
<b>Serietitel och serienummer</b>	<b>ISSN</b>					
Title of series, numbering	_____					
<b>URL för elektronisk version</b>  <div style="height: 40px; border: 1px solid black;"></div>						

<b>Titel</b> Title	Design and Development of a Versatile Hardware Test Platform
<b>Författare</b> Author	Eneas Puertas Kreusch

<b>Sammanfattning</b> Abstract	<p>Nowadays in the whole process around the design of an integrated circuit, verification and testing are the most time consuming tasks. Verify simple ICs is not a very complex task, but as the complexity increases, this critical task takes more and more time allowing for easily three or four times design time.</p> <p>Test platforms plays an important role in the whole process. Versatility is probably the most important feature of all. If the test platform can be re-used to test different ICs, the effect of the initial investment on its development in the cost of the verification process can be reduced.</p> <p>This project will be focused on the design and development of a versatile test platform.</p>
-----------------------------------	--

<b>Nyckelord</b> Keywords	versatile test platform, hardware, SoC design, test
------------------------------	---



## **Abstract**

Nowadays in the whole process around the design of an integrated circuit, verification and testing are the most time consuming tasks. Verify simple ICs is not a very complex task, but as the complexity increases, this critical task takes more and more time allowing for easily three or four times design time.

Test platforms plays an important role in the whole process. Versatility is probably the most important feature of all. If the test platform can be re-used to test different ICs, the effect of the initial investment on its development in the cost of the verification process can be reduced.

This project will be focused on the design and development of a versatile test platform.





## Acknowledgments

I would like to thank first to Per Karlström for given me the chance of doing my last year project with him. Second, and not for that less important, I would like to thank my family. Even though they did not understand any of the things I was telling them, they listened patiently and offered me words of support and motivation. And finally, I would like to thank my friends, for all the coffees shared explaining our problems and frustrations and the good times we spent together.

*Linköping, August 2012*  
*Eneas Puertas*



---

# Contents

<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>x</b>
<b>Abbreviations</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Why Testing . . . . .	1
1.2 Types of Tests and Platforms . . . . .	2
1.3 The Need of Versatile Test Platforms . . . . .	2
<b>2 Background</b>	<b>5</b>
2.1 HDL Languages . . . . .	5
2.1.1 HDL in IC Design . . . . .	5
2.1.2 HDL in IC Simulation . . . . .	6
2.1.3 Most Popular HDLs . . . . .	6
2.2 IP Blocks . . . . .	7
2.3 Novel Generator of Accelerators and Processors (NoGap) . . . . .	7
2.4 FPGA . . . . .	7
2.5 Simulation, Synthesis and FPGA Programming Tools . . . . .	8
<b>3 Related Work</b>	<b>11</b>
<b>4 The System</b>	<b>13</b>
4.1 Choosing the Test Platform . . . . .	13
4.2 Specifications of the SoC . . . . .	15
4.3 Target FPGA Board . . . . .	16
4.4 Final SoC Specifications . . . . .	18
<b>5 Available Resources</b>	<b>21</b>
5.1 SDRAM Memory . . . . .	21
5.2 SRAM Memory . . . . .	23
5.3 Flash Memory . . . . .	24

5.4	Interconnection Bus . . . . .	24
5.4.1	The Wishbone Bus Standard . . . . .	25
<b>6</b>	<b>The SoC and its elements</b>	<b>27</b>
6.1	The CPU . . . . .	29
6.1.1	OpenRISC 1000 Specifications . . . . .	29
6.1.2	OpenRISC OR1200 . . . . .	30
6.1.3	The Implemented OR1200 . . . . .	30
6.2	Wishbone Bus . . . . .	33
6.2.1	The Arbiter . . . . .	33
6.2.2	The Decoder . . . . .	33
6.2.3	The Bus . . . . .	34
6.3	SDRAM Controller . . . . .	35
6.4	SRAM Controller . . . . .	39
6.4.1	Control & I/O Block . . . . .	40
6.4.2	Acknowledge Control Block . . . . .	40
6.5	Flash controller . . . . .	43
6.5.1	Control & I/O Block . . . . .	44
6.5.2	Acknowledge Control Block . . . . .	44
6.6	AvBus . . . . .	47
6.6.1	AvBus Interface . . . . .	47
6.7	UART . . . . .	51
6.8	Boot Monitor . . . . .	53
6.9	Clock Manager . . . . .	53
<b>7</b>	<b>Tests</b>	<b>55</b>
7.1	Simulation Tests . . . . .	55
7.1.1	Single Tests . . . . .	56
7.1.2	Group Tests . . . . .	56
7.2	Hardware Tests . . . . .	57
<b>8</b>	<b>Results</b>	<b>61</b>
<b>9</b>	<b>Conclusions</b>	<b>63</b>
<b>10</b>	<b>Future Work</b>	<b>65</b>
10.1	System Improvements . . . . .	65
10.2	Future Goals . . . . .	66
	<b>Bibliography</b>	<b>67</b>

# List of Figures

2.1	FPGA basic structure . . . . .	8
4.1	Basic test flow . . . . .	14
4.2	Test platform . . . . .	15
4.3	SoC structure . . . . .	16
4.4	I/O modules interconnection to AvBus I/O connector . . . . .	17
4.5	Memory modules interconnection to AvBus memory connector . . . . .	18
4.6	Final SoC specifications . . . . .	19
6.1	Global structure of the SoC . . . . .	27
6.2	Implemented OR1200 structure . . . . .	31
6.3	Arbiter algorithm . . . . .	33
6.4	SDRAM tri-state . . . . .	36
6.5	Wishbone address decode . . . . .	37
6.6	SDRAM memory controller structure . . . . .	38
6.7	SRAM controller structure . . . . .	39
6.8	SRAM acknowledge control flow . . . . .	41
6.9	Flash controller structure . . . . .	43
6.10	Flash acknowledge control flow . . . . .	45
6.11	AvBus interface structure . . . . .	49
7.1	Testbench basic structure . . . . .	56
7.2	Hardware test structure . . . . .	58

# List of Tables

5.1	SDRAM controller commands . . . . .	22
5.2	Master device signals . . . . .	25
5.3	Slave device signals . . . . .	26
6.1	Device address range for decoding . . . . .	34
6.2	SDRAM address range . . . . .	35
6.3	SDRAM input/output signals . . . . .	36
6.4	SDRAM configuration signals . . . . .	37
6.5	SRAM address range . . . . .	39
6.6	SRAM input/output signals . . . . .	39
6.7	SRAM delay registers default values . . . . .	41
6.8	Flash address range . . . . .	43
6.9	Flash input/output signals . . . . .	43
6.10	Flash delay registers default values . . . . .	45
6.11	AvBus I/F board input/output signals . . . . .	48
8.1	FPGA logic utilization . . . . .	62
8.2	FPGA logic distribution . . . . .	62

---

## Abbreviations

ASIC = Application Specific Integrated Circuit

ASIP = Application Specific Instruction-set Processor

CLB = Configurable Logic Block

DCM = Digital Clock Manager

DSP = Digital Signal Processor

FPGA = Field Programmable Gate Array

HDL = Hardware Description Language

IC = Integrated Circuit

IP = Intellectual Property

MMU = Memory Management Unit

NoGap = Novel Generator of Accelerators and Processors

RAM = Random Access Memory

RISC = Reduced Instruction Set Computer

SMP = Symmetric Multi-Processing

SMT = Surface Mount Technology

SoC = System on a Chip

TLB = Translation Lookaside Buffer





# 1

---

## Introduction

### 1.1 Why Testing

Despite all the improvements that design environments have suffered, designers can not rely blindly in the results of the software tools they use. Assumptions, optimizations and other considerations that software tools made can result in, sometimes little and sometimes big, differences between the original design and the result that the software brings to the designer.

With simulation tools, the designer can get an initial estimation to how his design will behave, but some situations, such as the behavior of latches in the real implementation, can not be predicted. Finding a non desired behavior in the simulation phase is an easy thing to solve, but once the design is turned into silicon, it starts to be a little bit more complicated. Finding an error after the design has been sent to the manufacturer costs a company lots of money, finding it once the product is in the market costs money and the loss of costumers. That is why testing is so important in the development of ICs.

All companies are aware of this fact and spend huge amounts of time and money to detect and solve errors in all the phases of the design. To reduce the probability of finding error in the last stages of the design (when they could cause catastrophic consequences) companies make big efforts to plan and develop testing strategies in the early stages of the design, when errors can be solved easily. The amount of time spend in testing can be nearly 3 or 4 times the time spent designing the IC. That can give us a overall view of how important testing is.

## 1.2 Types of Tests and Platforms

All designs are unique. They can have parts shared with other designs (e.g. buses or interfaces) but at the end they are still different. Despite this huge variety we can find, the tests used to verify their behavior can be separated in four main categories.

All ICs are unique, they might have elements from other IC (e.g. IP cores or basic structures) but in the end it is always different from the rest. Despite this huge variety we can find on IC, tests can be separated in four main categories.

**Low level test.** This test focus on controlling the behavior of the IC bit by bit. This technique offers good results when small details are important, specially in low complexity ICs. When the complexity of the IC to test increases, the time this tests need to be accomplished, and their complexity, increases too.

**High level tests.** To test very complex ICs, to focus on small details is not a very efficient way of doing it. All the small components that make up the IC have been tested separately and its singular behavior has been properly checked. In this cases, what needs to be test is the global behavior of all the small components connected together. Instead of having to focus on separated bits, test engineers have to check busses, registers, memories, etc.. Using C/C++, .NET or other high level languages can provide an easy, or at least less complicated, way to perform complex tests.

**Integrated testing features.** Testing resources can be integrated in the IC, for example JTAG protocols or BIST, and they provide the testing engineer the capability to test directly from the board. Sometimes the tests are predefined and integrated in the system as "self tests", this usually happens when the ICs are not too complex and only a few number of tests are needed. When the test are bigger and more complex, they are introduced externally into the test platform and once they are done, the results are extracted and interpreted.

**External testing features.** Sometimes, due to the high density of the ICs or because of area issues, testing platforms can not be included in the IC so they need to be external. One example of this situation is the self called "bed of nails" used to test ICs or PCBs by making contact with some kind of nails (connected to the extern test platform) into certain points of the system and reading or forcing values in those points. That relieves the designer to include the testing platform in the design, but at the same time needs a complete external platform to perform the tests.

## 1.3 The Need of Versatile Test Platforms

As the reader might have noticed, the selection of the test type and platform is a big limiting factor for the target ICs capable of being tested. Creating a specific test platform for each possibility is not a very wise decision. Despite having a not very high development time, its non-reusable nature is its most limiting factor. By creating versatile testing platform we can ensure a re-usability of the resources invested in its development. It is going to be more complicated to develop and probably will need more time to be completed, but

---

once is done it can be used to test any IC, which at the end will mean a save of time and money.



# 2

---

## Background

This chapter shows the different elements the reader must know or at least be familiar with, basic knowledge and useful tools and environments that will be mentioned along the different chapters and that have been used in the development of the project.

### 2.1 HDL Languages

Hardware Description Language (HDL) [19] refers to any language used for description and design of electronic circuits, most commonly digital circuits. HDLs can be used to describe the behavior of circuits and to create tests to verify them.

When describing the behavior or internal structure of a circuit, designers generally use algorithms, flow charts or mathematical expressions. HDLs are used to transform those abstract elements into something real.

#### 2.1.1 HDL in IC Design

Despite they are also used to design analog circuits, the main field of application of HDLs is the design of digital integrated circuits.

The advantage that HDLs have against other traditional programming languages, like C or C++ when designing a digital circuit, is that they have been created specifically for this task. It does not mean that traditional languages can not be used to design a digital circuit, indeed they can be used, but some special libraries need to be added, which makes the whole process a little bit more complicated than by using HDLs.

During the process of designing an IC, designing tools and environments are another important elements designers can use. Modern tools automatically check for grammar errors, error prone structures or potentially dangerous elements than could end in unexpected be-

haviors. With the increase of the level of elements integrated in an IC, optimization and timing constraint checking tasks are becoming more and more important. Modern design tools can perform this tasks automatically, allowing the design engineer to focus in other tasks. At the same time, the information this tools show to the engineer allows him/her to know even before prototyping very important information, for example the maximum speed of the design, area consumption or critical paths.

### 2.1.2 HDL in IC Simulation

Another important feature HDLs have is the capability of be used to simulate HDL designs. To be able to do that, engineers only have to add an extra layer of HDL code called testbench. A testbench is basically an extra module with the test procedures added to the main design. It is written in HDL as the design, but not necessarily in the same language. At least, testbenches have the instantiation of the design inputs/output, some logic to perform the tests and control elements to verify the results. One of the key features of HDLs in this situations is that they have a certain number of instructions or elements that can not be synthesized but can be used in simulation.

Modern simulation environments provide design engineers the necessary tools to perform deep level tests to their designs. Simulations can be stopped and resume any time, break-points can be inserted without having to modify the HDL code and the behavior of the design can be controlled on any layer. Results can be obtained automatically with the use of assertions or other comparative functions, but when debugging a deeper view of the design is more desirable. For this purpose, simulation tools offer graphical environments to display the values and transitions of any signal in the design, facilitating the tasks of error detection.

### 2.1.3 Most Popular HDLs

The most used and supported HDLs today are, *SystemVerilog*, *Verilog* and *VHDL* (VH-SIC HDL or very-high-speed integrated circuits HDL).

#### **SystemVerilog**

*SystemVerilog* [3] is a combination of hardware description language and hardware verification language. It is a major extension of the *Verilog* language so they share most of the structures.

The latest IEEE standard was adopted in 2009 when *SystemVerilog* was merged with the existing *Verilog* standard, creating the IEEE 1800-2009 standard [10].

#### **Verilog**

*Verilog* [17] designs are based in the use of modules as design entities. Modules have input, output and bidirectional ports as well as procedural blocks. If another module needs to be used, its instantiation is also present. Unlike C/C++, *Verilog* uses begin/end instead of curly brackets to delimit procedural blocks.

The latest *Verilog* standard is Verilog 2005, also known as IEEE 1364-2005 [9].

## VHDL

*VHDL* [18] designs are divided in entities and architectures. The entity defines the in/out ports of the design and the architecture defines the inner behavior of the design. There are different ways to describe the behavior of the designs, so there can be several architectures for the same entity. *VHDL* designs use libraries to increase the resources the designer has available.

The latest *VHDL* standard is *VHDL* 4.0 also known as IEEE 1076-2008 [8].

## 2.2 IP Blocks

IP block is the acronym of Intellectual Property block. An IP block is a reusable electronic element, for example a logic unit, cell or chip layout design that has been developed by a company or institution and can be licensed to third parties. Depending on the copyright of the IP core, licenses can be free or not.

## 2.3 NoGap

NoGap is a design automation tool for ASIP and accelerator developed by Per Karlström [6] from the Department of Electrical Engineering of Linköping's University.

In today's design of ASIPs, the designer has to choose one of the two main ways to approach the development of a new ASIP.

With HDL languages, the designer has a strong control over the register transfer level, having the possibility of building a full custom system, but, at the same time, he or she has to deal with all the small details of hardware multiplexing, interconnections and buses, control signals and other elements that makes the whole process really tedious and error prone.

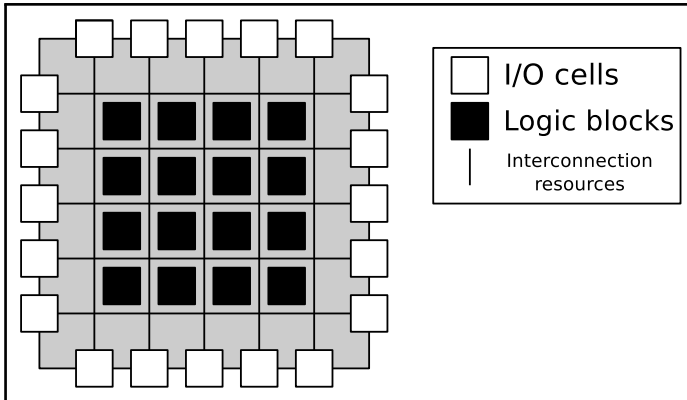
Choosing EDA tools, the designer can focus only in the actual design of the system without having to worry about the small details. But this abstraction from the deepest details only leaves to the designer the ability to work with partially-configurable grey boxes provided by the EDA tool.

NoGap fills the space between the two solutions. The designer can have low level control over the register transfer level while at the same time does not lose the high level perspective of a high level complex system. That means that the designer can focus on what he or she does best: being creative, while the computer takes care of the boring, tedious and repetitive tasks which does better, quicker and with less errors than humans.

## 2.4 FPGA

FPGA. It is a device which its behavior can be programmed by changing the connections between its inner components.

The basic structure of an FPGA consists of I/O cells to connect the FPGA to any external device, programmable logic blocks and interconnection resources to connect the logic block and the I/O cells (see 2.1). In addition to the basic structure, an FPGA can include also RAM blocks, digital clock managers, dedicated multipliers, DSP blocks, etc.. Every manufacturer of FPGAs follows the basic structure but by modifying it and adding extra features tries to make a differentiation of his product against the competitors.



**Figure 2.1:** FPGA basic structure

FPGAs are configured using bit files that can be generated from HDL files. Before receiving the specifications of the IC to be implemented and having the parameters of the target FPGA model, the synthesizer converts the behavior described in HDL (usually Verilog or VHDL) into the internal configuration of the logic blocks and the interconnection resources.

FPGAs are used in early prototyping, aerospace applications, bio-medicine, computer aided vision systems and voice recognition among many others. Nowadays its use is increasing in applications with high needs of parallelism.

## 2.5 Simulation, Synthesis and FPGA Programming Tools

*Modelsim* has been the tool used for simulation. Developed by *Mentor Graphics*, is used for simulation and debugging of *Verilog*, *SystemVerilog*, *VHDL*, and *SystemC*.

*Precision* (also developed by *Mentor Graphics*) has been used. *Precision* is a vendor independent synthesis tool which brings the designer the capability of developing code without having to focus on a concrete FPGA architecture. Instead of using the graphical environment, synthesis commands have been included in a *Makefile* to make all the process accessible through a single command over the terminal.



---

The tool used to program the FPGA is *xc3sprog*. This tool uses the JTAG connector to program Xilinx's FPGA's.



# 3

---

## Related Work

A similar system was developed by Olle Seger and Per Karlström. It is named Dafk and is being used in the course code TSEA44 Computer hardware - A system on a chip [1].

This system has been used to verify the contents of the memories once read by this project memory controllers.

Another CPU that has been used in the development of SoCs is LEON (and its revisions LEON2, LEON3 and LEON4). It is a 32-bit CPU microprocessor core, based on the SPARC-V8 RISC architecture and instruction set. It was originally designed by the European Space Research and Technology Centre (ESTEC), part of the European Space Agency (ESA), and after that by Gaisler Research [2].



# 4

---

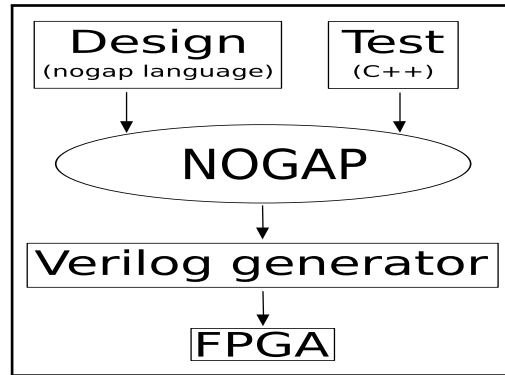
## The System

As the reader will have noticed reading previous chapters, the need of providing a versatile test platform is a very important aspect of today's IC design industry. In this case, the target of the versatile test platform is the NoGap platform. In this chapter, the reader will find the specifications of the test platform as well as the FPGA board that is going to be used for its development.

### 4.1 Choosing the Test Platform

The task of testing accelerators and processors is not a simple one. Due to their complexity, low level testing techniques are too complicated: there are so many situations that need to be tested and doing that bit by bit can be a tedious and error prone work. That is why high level techniques are more suitable in this concrete case.

The NoGap platform has been written in C++ and that brings the possibility of developing high level tests using C++ that can be easily processed by the NoGap core.

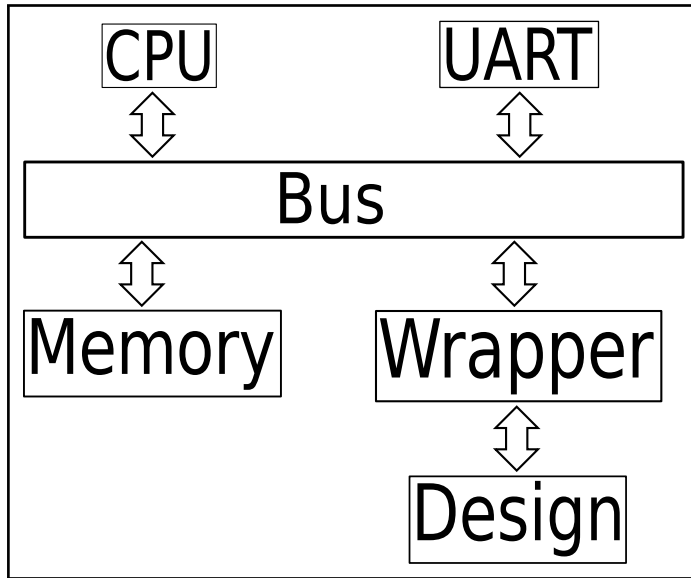


**Figure 4.1:** Basic test flow

As the reader can see in Figure 4.1, the flow of the testing procedures is not very complicated. Having the test and the design itself as inputs, the NoGap platform will process them and generate an output that will be translated into HDL by the Verilog generator and then synthesized into a FPGA to perform the tests described in the input test file.

Creating an individual hardware for testing purposes is possible, but not very practical. A better way to perform the test without having to create a specific hardware for each design is to have a predefined hardware with the possibility of changing the way it works to meet the requirements of the input test file. This means that even though the hardware will be static, the way it works will be dynamic. At the same time it should be able to send and receive data to and from the design to test, process this data and send it in a clear way to the user. With all these features on mind, the choice of an SoC is almost obvious.

The use of a SoC provides a powerful solution. In addition to the basic structure of CPU + memory, the user can add any module that can be useful to the tasks it will perform: standard I/O modules such as UART or Ethernet, quick and/or non volatile memories such as SDRAM and Flash, external device controllers, etc.. That means almost infinite possibilities of configuration and as the objective is: a versatile test platform.



**Figure 4.2:** Test platform

The basic structure of the SoC will include a CPU, an UART, a memory module, an interconnection bus and a wrapper (as shown in Figure 4.2). The NoGap core will translate the contents of the input test file into instructions that can be executed by the CPU and then, store them into the memory module so they can be reached by the CPU. The instruction will be loaded through the interconnection bus, and once decoded and through the wrapper, the CPU will perform the tests directly over the design. Once the results data are obtained, the CPU will process and send them to the user through the I/O module.

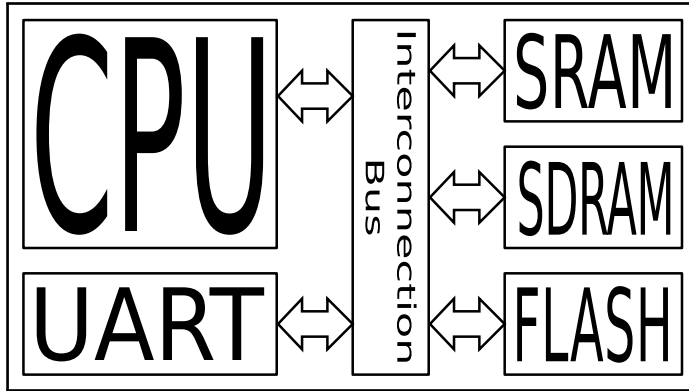
## 4.2 Specifications of the SoC

As the wrapper specifications will be exclusive of each design, the specifications of the SoC will be focused on the CPU, memory, I/O and interconnection bus. In order to expand the possibilities, the system will include 3 kinds of memory, SRAM, SDRAM and Flash. One possible configuration is:

- Tests stored in the SDRAM, the largest memory available in the system.
- Results stored in the SRAM.
- Firmware stored in the Flash.

A UART is going to be in charge of the I/O operations, sending and receiving data over the RS232 protocol.

The resulting system will consist of a CPU, 3 memory controllers (one for each memory), a UART and an interconnection bus (as shown in Figure 4.3).



**Figure 4.3:** SoC structure

### 4.3 Target FPGA Board

The testing platform will be developed for the *Avnet Virtex II Development Board* [14] along with the *Avnet Communications/memory Module* [15]. It has a *Virtex II* (model XC2V4000) FPGA manufactured by Xilinx.

The specifications of the XC2V4000 shown on its data sheet [20] are:

- 4M System Gates
- 80 by 72 array of CLB, 23,040 slices and a maximum of 720 kilobytes of distributed RAM
- 120 multiplier blocks
- 120 18 kbit SelectRAM blocks with a maximum of 2,160 Kbits of RAM
- 12 DCM's
- A maximum of 912 user I/O pads

The *Avnet Virtex II Development Board* and the *Avnet Communications/memory Module* have the following features:

#### Development Board

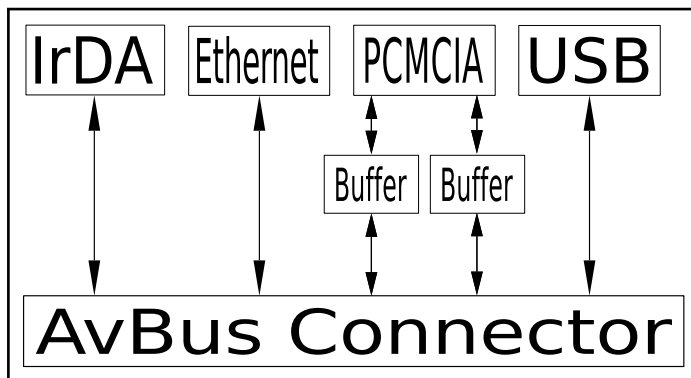
- *Xilinx Virtex II* model XC2V4000 FPGA
- *System ACE Multi-Package Module (MPM)* solution addresses the need for a space-efficient, pre-engineered, high-density configuration solution in multiple FPGA systems
- 128 MB DIMM module of DDR SDRAM over a 64Bit wide data bus running at a maximum speed of 133 Mhz
- 16 MB of Flash memory in a 16 bit configuration over a 16Bit wide data bus



- JTAG
- RS232 connector
- AvBus expansion module bus and connectors

#### Communications/Memory Module

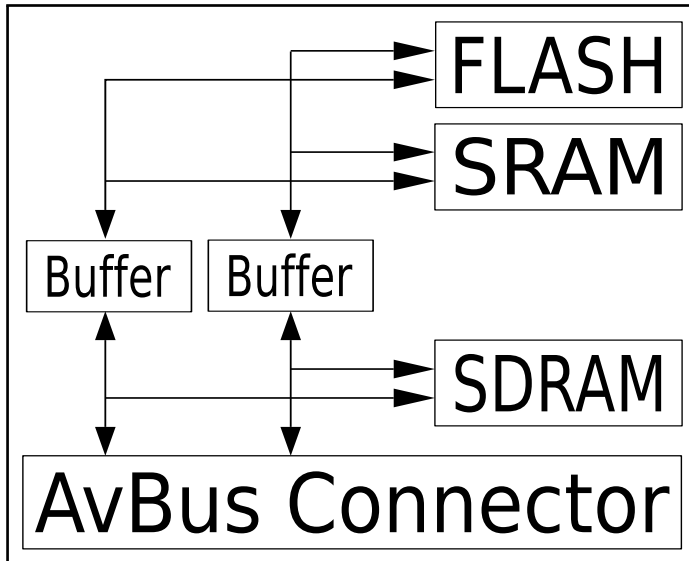
- Two AvBus connectors
- 64Mbytes of SDRAM organized as 16Mbits x 32
- 1Mbyte of SRAM organized as 256K x 32
- 16 Mbytes of Flash memory organized as 4M x 32
- Gigabit Ethernet
- USB 2.0
- IrDA
- PCMCIA/PCCard



**Figure 4.4:** I/O modules interconnection to AvBus I/O connector

The AvBus connector is a standard interconnection bus defined by Avnet for their products.

The different modules of the *Avnet Communications/memory Module* are connected in the following way. The four I/O modules (Ethernet, USB, IrDA and PCMCIA) are connected to one AvBus connector, the 3 memory modules (SRAM, SDRAM and Flash) are connected to the other AvBus connector. All the I/O modules are directly connected to the AvBus connector except the PCMCIA, which has two buffers (one for data and one for addresses) in between. The same happens with the memory modules, the SDRAM is directly connected to the AvBus connector and the Flash and SRAM share two buffers (one for data and one for addresses) in between. The overall structure is shown in Figure 4.4 and Figure 4.5.

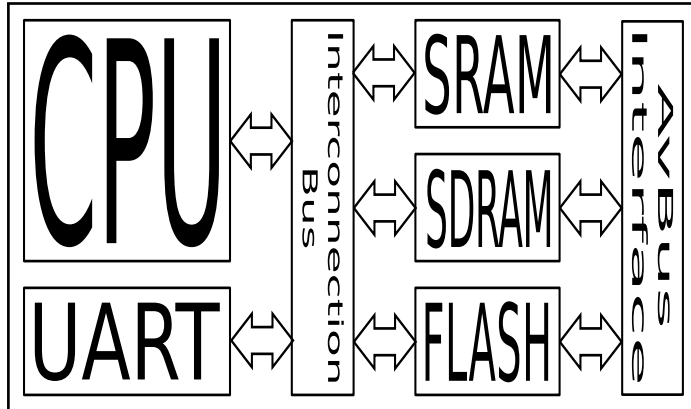


**Figure 4.5:** Memory modules interconnection to AvBus memory connector

## 4.4 Final SoC Specifications

Once the specifications of the target FPGA have been listed, there is a little change that has to be done in the SoC initial specification in order to be compatible with the *Avnet Virtex II Development Board*. The three memory modules share the same AvBus connector, so the SoC will need an interface to control the common signals and the buffers that the SRAM and Flash module share.

The final SoC specification will include then a CPU, an interconnection bus, a UART, SRAM, Flash and SDRAM memories and an interface to the AvBus connector (as shown in Figure 4.6) .



**Figure 4.6:** Final SoC specifications

At this point, the reader might find many similarities with the Dafk system [1]. It is also a SoC and initially it can be used for the same purpose as the system this project is focused. But there are some elements and objectives that only can be achieved by a new system.

This system will have an updated version of the *OR1200* CPU used in the Dafk system that has been proved to be able to run Linux. Instead of having a big memory controller for the three memory modules, separated controllers are going to be implemented and the SDRAM controller will now support burst from the beginning.



# 5

---

## Available Resources

In this chapter, the available resources present in the *Avnet Virtex II Development Board* and the *Avnet Communications/memory Module* are going to be listed and explained.

### 5.1 SDRAM Memory

The SDRAM memory modules present in the *AvNet communications/memory Module* are two Micron MT48LC16M16A2 [16]. The basic size of this chips is 16Mb x 16 (4 Mb x 16 x 4 banks), but using two of them (sharing the address for both chips) allows the user to have a total of 64MB (2MB x 32 configuration). The module has the following control signals and buses:

- 13 bits wide address bus
- 32 bits wide data bus (2 x 16)
- active high Clock (CLK)
- active low Chip Enable ( $\overline{CE}$ )
- active low Write Enable ( $\overline{WE}$ )
- active low Output Enable ( $\overline{OE}$ )
- active low RAS ( $\overline{RAS}$ )
- active low CAS ( $\overline{CAS}$ )
- active high Clock Enable (CKE)
- active high 2 bit wide Bank selection (BA)

- active high 4 bit wide (2 x 2) Byte selection (DQM)

The memory module also have many timing constrains in order to work properly. These are the most important ones:

- Input setup time (min 1.5 ns)
- Input hold time (min 0.8 ns)
- ACTIVE to PRECHARGE command  $T_{RAS}$  (37 to 120 ns)
- ACTIVE to ACTIVE command period  $T_{RC}$  (min 60 ns)
- ACTIVE to READ/WRITE delay  $T_{RCD}$  (min 15 ns)
- Refresh period (8102 rows)  $T_{REF}$  (64 ms)
- AUTO REFRESH period  $T_{RFC}$  (min 66 ns)
- WRITE recovery time  $T_{WR}$  (min 1 CLK + 7 ns)

A parameter that defines most of the values showed in the previous list is the CAS(READ) latency, the delay between the registration of a READ command and the availability of the first piece of output data in clock cycles (from now on referred as CL). This value is related with the frequency of the input clock. For frequencies up to 133 Mhz the value of CL is 2 and for frequencies from 133 Mhz to 143 Mhz the value of CL is 3. The system clock runs at 40 Mhz, so even if its doubled to feed the SDRAM memory modules, CL = 2 is the correct choice.

The memory chip works with commands rather than with single signals. A command is a combination of values of the signals CS, RAS, CAS and WE. The different commands are shown in Table 5.1

**Table 5.1:** SDRAM controller commands

Name (function)	CS	RAS	CAS	WE
Command inhibit	H	X	X	X
No operation	L	H	H	H
ACTIVE	L	L	H	H
READ	L	H	L	H
WRITE	L	H	L	L
BURST TERMINATE	L	H	H	L
PRECHARGE	L	L	H	L
AUTO REFRESH or SELF REFRESH	L	L	L	H
LOAD MODE REGISTER	L	L	L	L

## 5.2 SRAM Memory

The SRAM memory modules present in the *AvNet communications/memory Module* are two Cypress CY7C1041V33 [16]. The basic size of this chips is 256k x 16, but using two of them (sharing the address for both chips) allows the user to have a total of 1Mb (256k x 32 configuration).

The module has the following signals:

- 18 bits wide address bus
- 32 ( 2 x 16) bits wide data bus
- active low Chip Enable ( $\overline{CE}$ )
- active low Write Enable ( $\overline{WE}$ )
- active low Output Enable ( $\overline{OE}$ )
- active low Byte High Enable ( $\overline{BHE}$ )
- active low Byte Low Enable ( $\overline{BLE}$ )
- 20 ns. reading and writing cycle

The chip can perform two tasks: read and write. For reading a certain position of the memory, the address must be placed in the address bus and then CE and OE must be set to low level while WE is set to high. To write something into a certain memory address, the data must be placed in the data bus and the address to write in the address bus, then CE and WE must be set to low while OE is set to high. Setting BLE and BHE to high or low level the user can control which bytes are read or written. As there are two memory chips, BLE and BHE of the first one will control byte 0 and 1 and BLE and BHE of the second chip will control bytes 2 and 3. All these four signals are merged in the controller in a 4 bit wide signal called  $\overline{SEL}$ .

## 5.3 Flash Memory

The Flash memory modules present in the *AvNet communications/memory Module* are two Intel E28F640J3 [11]. The basic size of this chips is 128Mb x 16, but using two of them (sharing the address for both chips) allows the user to have a total of 16MB (128Mb x 32 configuration).

The module has the following signals:

- 23 bits wide address bus
- 32 bits wide data bus
- active high Chip Enable (CE)
- active low Write Enable ( $\overline{WE}$ )
- active low Output Enable ( $\overline{OE}$ )
- 120 ns. read and write cycle

The internal behavior of this chip is a little bit complex and it is explained in detail in its data sheet.

## 5.4 Interconnection Bus

Like the main goal of the system, the interconnection bus must be versatile but without losing a strong, well defined structure. According with the SoC versatile characteristics, any new module should be able to be added to the system without having to make major changes in the interconnection bus structure. At the same time, the structure in the bus must be powerful enough to support any kind of master and slave devices. With this features on mind, the *Wishbone* standard was a very good choice.



### 5.4.1 The Wishbone Bus Standard

The Wishbone Bus is an open source hardware computer bus developed by Silicore Corporation [13]. The standard does not specify any electrical information neither the structure of the bus, instead of that, it specifies the in/out signals, clock cycles, high and low logic levels.

The specifications of the wishbone bus implemented are the following:

- 32 bit data bus
- 32 bit address bus
- No TAG (user defined) signals
- Shared bus topology for point-to-point module connections
- Separated arbiter and decoder from the main bus implementation

The in/out signals a master and a slave have are predefined by the *Wishbone* standard. By having strict control over the interfaces, compatibility with new modules can be ensured as long as they have the predefined interface.

The signals both master and slave wishbone devices have are shown in Table 5.2 and Table 5.3.

**Table 5.2:** Master device signals

Name	Type	Width	Description
CLK_I	input	1 bit	Input clock
RST_I	input	1 bit	Input reset
ADR_O	output	32 bits	Output address
DAT_O	output	32 bits	Output data
DAT_I	input	32 bits	Input data
WE_O	output	1 bit	Write enable
SEL_O	output	4 bits	Byte sel
STB_O	output	1 bit	Valid data transfer cycle
ACK_I	input	1 bit	Acknowledge signal

In addition to these signals, some modules can have the *CTI\_I* or *CTI\_O* signal. This 3 bits wide signal is in charge of the control of the burst cycles.

For further information about the Wishbone B3 standard please check the Wishbone B3 standard documentation [13].

*Table 5.3: Slave device signals*

Name	Type	Width	Description
CLK_I	input	1 bit	Input clock
RST_I	input	1 bit	Input reset
ADR_I	input	32 bits	Output address
DAT_O	output	32 bits	Output data
DAT_I	input	32 bits	Input data
WE_I	input	1 bit	Write enable
SEL_I	input	4 bits	Byte sel
STB_I	input	1 bit	Valid data transfer cycle
ACK_O	output	1 bit	Acknowledge signal

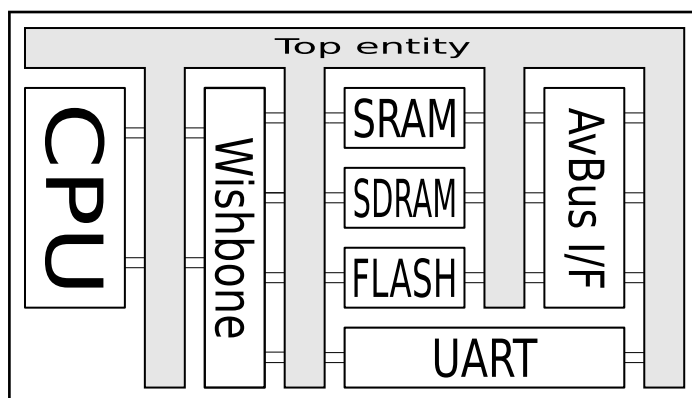
# 6

---

## The SoC and its elements

In this chapter, all the elements that conforms the SoC will be explained in detail, its connections and how they are expected to behave. Some of them are IP cores, so more detailed explanations about them is available at their respective documentation.

As shown in Figure 6.1, all the modules are connected using a top entity. This entity has all the inner connections between the modules as well as the output and input pins connected to the FPGA and the memory/communication peripherals.



**Figure 6.1:** Global structure of the SoC



## 6.1 The CPU

The CPU used is the *OpenRISC OR1200* [5], an implementation of the *OpenRISC 1000* [12] architecture specifications for 32/64 bits RISC/DSP processors. The architecture specifications and the CPU have been developed by Open Cores ([www.opencores.org](http://www.opencores.org)).

### 6.1.1 OpenRISC 1000 Specifications

The *OpenRISC 1000* architecture is a completely open architecture. It defines the architecture of a family of open source, RISC microprocessor cores. The *OpenRISC 1000* architecture allows for a spectrum of chip and system implementations at a variety of price/performance points for a range of applications. It is a 32/64-bit load and store RISC architecture designed with emphasis on performance, simplicity, low power requirements, and scalability. *OpenRISC 1000* targets medium and high performance networking and embedded computer environments. Performance features include a full 32/64-bit architecture; vector, DSP and floating-point instructions; powerful virtual memory support; cache coherency; optional SMP and SMT support, and support for fast context switching. The architecture defines several features for networking and embedded computer environments. Most notable are several instruction extensions, a configurable number of general-purpose registers, configurable cache and TLB sizes, dynamic power management support, and space for user-provided instructions.

The principal features of the *OpenRISC 1000* architecture specifications are:

- A completely free and open architecture.
- A linear, 32-bit or 64-bit logical address space with implementation-specific physical address space.
- Simple and uniform-length instruction formats featuring different instruction set extensions:
  - OpenRISC Basic Instruction Set (ORBIS32/64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 32- and 64-bit data
  - OpenRISC Vector/DSP eXtension (ORVDX64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 8-, 16-, 32- and 64-bit data
  - OpenRISC Floating-Point eXtension (ORFPX32/64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 32- and 64-bit data
- Two simple memory addressing modes, whereby memory address is calculated by:
  - Addition of a register operand and a signed 16-bit immediate value
  - Addition of a register operand and a signed 16-bit immediate value followed by update of the register operand with the calculated effective address
- Two register operands (or one register and a constant) for most instructions who then place the result in a third register

- Shadowed or single 32-entry or narrow 16-entry general purpose register file
- Branch delay slot for keeping the pipeline as full as possible
- Support for separate instruction and data caches/MMUs (Harvard architecture) or for unified instruction and data caches/MMUs (Stanford architecture)
- A flexible architecture definition that allows certain functions to be performed either in hardware or with the assistance of implementation-specific software
- Number of different, separated exceptions simplifying exception model
- Fast context switch support in register set, caches, and MMUs

### 6.1.2 OpenRISC OR1200

The *OR1200* is a 32-bit scalar RISC with Harvard micro-architecture, 5 stage integer pipeline, virtual memory support and basic DSP capabilities. It has been successfully tested and implemented in various FPGA models and ASIC designs.

The main characteristics of the *OR1200* are:

- Central CPU/DSP block
- IEEE 754 compliant single precision FPU
- Direct mapped data cache
- Direct mapped instruction cache
- Data MMU based on hash-based DTLB
- Instruction MMU based on hash-based ITLB
- Power management unit and power management interface
- Tick timer
- Debug unit and development interface
- Interrupt controller and interrupt interface
- Instruction and Data WISHBONE B3 compliant interfaces

All of this modules can be enabled or disabled for synthesis via the defines file (*or1200\_defines.v*). Along with the modules, the list of available instructions can be modified enabling a specific instruction meeting the concrete requirements of the target system.

### 6.1.3 The Implemented OR1200

Due to the versatility this processor has, it can be configured to have only the features needed and no other useless ones which will only consume area and power without providing any performance improvement.

This are the implemented features of the *OR1200*

**Data and Instructions cache and MMU's.** As the main goal of the system is to perform high levels hardware tests, and primarily because the *OR1200* is supported by the LINUX kernel, it is very feasible that at the end it might be running a UNIX operative system. That is why caches and MMUs are needed.

**Interrupt controller and I/F.** Some of the modules present in the system (communication modules for example) needs an interrupt I/F to communicate with the processor when they have new data available, when they wait for a transmission, etc.

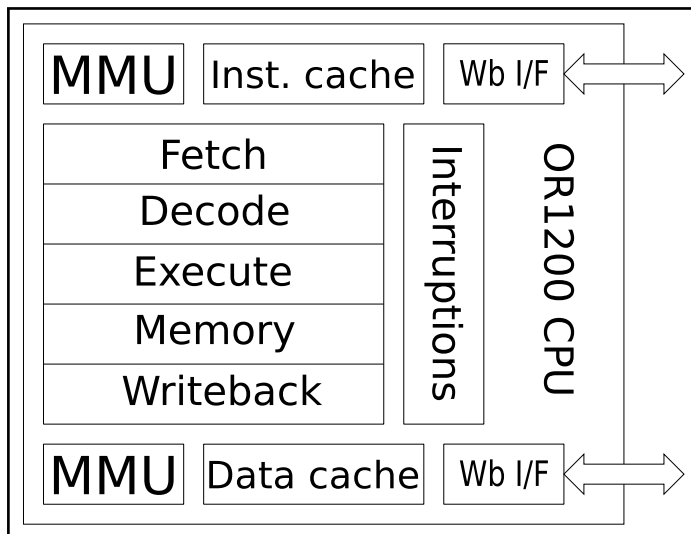
**Instruction and Data WISHBONE I/F.** To include a *Wishbone* I/F is a basic requisite of any module implemented in the system as it is the intercommunication standard chosen for the communication between all the modules.

Finally this are the features that will not be implemented (at least in the first version of the system).

**Power management unit.** Power consumption is not one of the primary goals of the system. It is going to be implemented on a FPGA, so energy consumption are assumed small.

**Tick Timer.** This module has not been implemented for the only reason of saving area. If later there is a real need of precise time measure this module can be implemented without any problem.

**Debug unit and development interface.** The processor is supposed to be 100% functional, so there are, in principle, no reasons to include it.



**Figure 6.2:** Implemented *OR1200* structure

For further Information about the *OpenRISC 1000* architecture specifications or the *OR1200* processor, please check the data sheets.



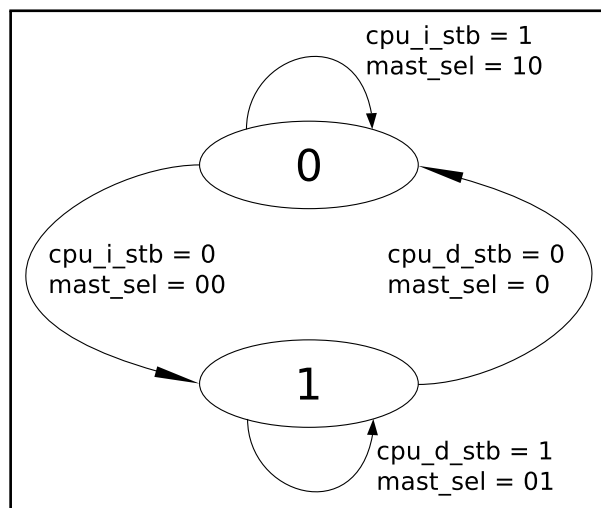


## 6.2 Wishbone Bus

The structure of the *Wishbone bus* implemented is very simple, but is this simplicity what gives its enormous versatility. There are three modules: the arbiter, the decoder and the bus itself.

### 6.2.1 The Arbiter

The arbiter is in charge of giving the control of the bus to the master which wants to perform an operation (reading or writing). It has as input the STB signals from the masters and as output it has a signal called *mast\_sel* with contains the 2 bit wide code of the masters which owns the control of the bus.



**Figure 6.3:** Arbiter algorithm

Each clock cycle, the arbiter checks if any of the input signals is at a high level. If it is, then a certain code is assigned to the output signal, if not, it continues checking another input signal. The codes are "01" for the instructions cache, "10" for the data cache and "00" for no master selected (see fig Figure 6.3).

### 6.2.2 The Decoder

The decoder has as inputs both output addresses from the instruction and data caches and the *mast\_sel* signal from the arbiter. Depending on the value of that last signal, the decoder selects the slave requested by the master which owns the bus operation. The selection is done by decoding the 4 MSB of the owners output address. The address ranges which correspond to each slave are shown in Table 6.1.

Once the decoding is done, the arbiter assigns a value to the an output signal called *slv\_sel* with contains a 3 bits wide code of the slave selected by the master. The codes are "001"

**Table 6.1:** Device address range for decoding

Device	Start address	End address
SRAM	0x2000_0000	0x2001_0000
FLASH	0xF000_0000	0xf100_0000
SDRAM	0x0000_0000	0x00ff_ffff
UART	0x9000_0000	0x9000_ffff
Boot monitor	0x4000_0000	0x4000_ffff

for the SRAM, "010" for the FLASH, "100" for the SDRAM, "011" for the UART, "111" for the boot monitor and "000" for no slave selected.

### 6.2.3 The Bus

The bus can be seen as a big switch. It receives the *mast\_sel* and *slv\_sel* from the arbiter and the decoder, and depending on the values the signals have, the bus connects the correct master with the slave it has requested. Whenever a slave is selected, the *wb\_stb* signals of the rest of them are disabled (set to zero).

## 6.3 SDRAM Controller

The SDRAM memory controller is an IP core developed by Dinesh Annayya and hosted by *OpenCores*. The controller is distributed under the GNU Lesser General Public License.

**Table 6.2:** SDRAM address range

Base address	Length	Offset
0x00000000	0x03ffe000	0x00000000

The main features of the controller are:

- 8/16/32 Configurable SDRAM data width
- Support asynchronous application layer and SDRAM layer
- Wishbone compatible application layer
- Programmable column address
- Support for industry-standard SDRAM devices and modules
- Supports all standard SDRAM functions
- Fully Synchronous; All signals registered on positive edge of system clock
- One chip-select signals
- Support SDRAM with four bank
- Programmable CAS latency
- Data mask signals for partial write operations
- Bank management architecture, which minimizes latency
- Automatic controlled refresh
- Static synchronous design

The SDRAM controller top file (*sdrc\_top.v*) inputs/outputs are shown in Table 6.3

The last 3 signals are connected into a tri-state buffer (as shown in Figure 6.4). Since there is a common interface for the three memory controllers into the *AvBus connector* and all of them share the data bus, the tri-state buffer has been changed into two signals, one for input data and one for output data, as in the other two controllers. The new signals are called *sir\_dq\_in* and *sir\_dq\_out*.

Table 6.3: SDRAM input/output signals

Name	Type	Width	Description
wb_clk_i	input	1 bit	clock (from the wishbone bus)
wb_rst_i	input	1 bit	reset (from the wishbone bus)
wb_dat_i	input	32 bits	data (from the wishbone bus)
wb_dat_o	output	32 bits	data (to the wishbone bus)
wb_ack_o	output	1 bit	acknowledge (to the wishbone bus)
wb_addr_i	input	32 bits	address (from the wishbone bus)
wb_we_i	input	1 bit	write enable (from the wishbone bus)
wb_cyc_i	input	1 bit	cycle (from the wishbone bus)
wb_stb_i	input	1 bit	strobe (from the wishbone bus)
wb_cti_i	input	1 bit	cycle type identifier (from the wishbone bus)
sdr_cke	output	1 bit	clock enable (to memory module)
sdr_cs_n	output	1 bit	chip select (to memory module)
sdr_ras_n	output	1 bit	ras (to memory module)
sdr_cas_n	output	1 bit	cas (to memory module)
sdr_we_n	output	1 bit	write enable (to memory module)
sd_dqm	output	4 bits	data bus mask (to memory module)
sdr_ba	output	2 bits	bank address (to memory module)
sdr_addr	output	32 bits	address (to memory module)
pad_sdr_din	input	32 bits	data (from memory module)
sdr_dout	output	32 bits	data (to memory module)
sdr_den_n	internal	1 bit	(tri-state buffer control signal)

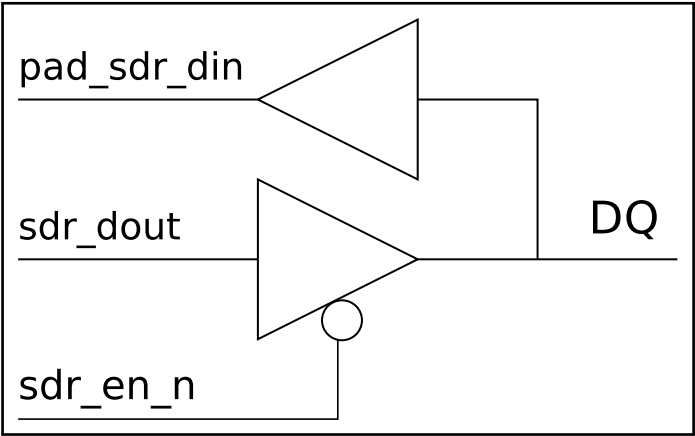


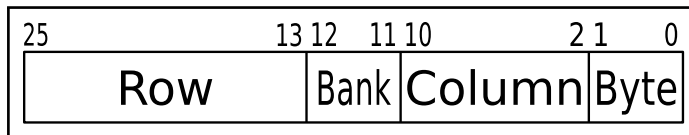
Figure 6.4: SDRAM tri-state

The standard address was 12 bits wide. To be able to use the whole memory, the structure of the controller has been modified to use 13 bits wide addresses (needed to access all the

**Table 6.4:** SDRAM configuration signals

Name	Parameter
cfg_sdr_tras_d	SDRAM active to precharge delay, specified in clocks
cfg_sdr_trp_d	SDRAM precharge command period ( $T_{RP}$ ), in clocks
cfg_colbits	SDRAM column bit
cfg_sdr_trcd_d	SDRAM active to read or write delay ( $T_{Rcd}$ ), in clocks
cfg_sdr_en	SDRAM Controller Enable
cfg_req_depth	Maximum Request accepted by SDRAM controller
cfg_sdr_mode_reg	SDRAM Mode Register
cfg_sdr_cas	SDRAM CAS latency, in clocks
cfg_sdr_trcar_d	SDRAM active to active/auto-refresh command period ( $T_{Rc}$ ), in clocks
cfg_sdr_twr_d	SDRAM write recovery time ( $T_{WR}$ ), in clocks
cfg_sdr_rfsh	Period between auto-refresh commands issued by the controller, in clocks
cfg_sdr_rfmax	Maximum number of rows to be refreshed at a time

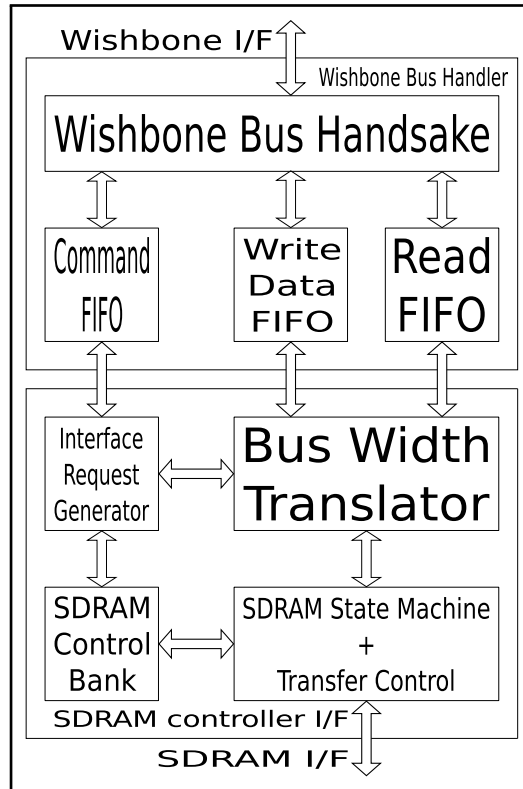
rows available). The way the controller decodes the 25 LSB of the wishbone address into the row, bank and column to be accessed is shown in Figure 6.5.

**Figure 6.5:** Wishbone address decode

In addition to the memory/wishbone inputs and outputs, the SDRAM controller has another set of inputs: the configuration inputs. These inputs are the different timing parameters that need to be set in order to make the controller work as expected. Having them as an external file connected to the controller confers to the user an easy way to modify any of the parameters without having to dig deep into the controller architecture. These inputs are shown in Table 6.4.

In addition to all the inputs/outputs described before, the SDRAM controller has another one called *sir\_init\_done* and as its name indicates, is used to tell when the controller has been initialized.

The structure of the SDRAM memory controller is divided into two main blocks, the wishbone bus handler and the SDRAM controller itself (as shown in Figure 6.6).



**Figure 6.6:** SDRAM memory controller structure

The Wishbone bus handler controls the Protocol handshake between wish bone master and custom SDRAM controller. This block also takes care of necessary clock domain change over. This block includes the Command Async FIFO, Write Data Async FIFO and Read Data Async FIFO.

The SDRAM controller is divided in 4 sub-blocks: SDRAM Bus convertor, SDRAM request generator, SDRAM Bank controller and SDRAM transfer controller. The SDRAM bus convertor converts and re-aligns the the system side 32 bit into equivalent 8/16/32 SDR format. The SDRAM request generator controls the interaction between the request and the application layer. The SDRAM bank controller takes requests from SDRAM request generator, checks for page hit/miss and issues precharge/activate commands and then passes the request to SDRAM Transfer Controller. The SDRAM transfer controller takes requests from SDRAM Bank controller, runs the transfer and controls data flow to/from the app. At the end of the transfer it issues a burst terminate if not at the end of a burst and another command to this bank is not available.

For further and more detailed information about the controller, please check de controller specifications file [4].

# 6.4 SRAM Controller

Table 6.5: SRAM address range

Base address	Length	Offset
0x00000000	0x00100000	0x20000000

The SRAM controller (*sram\_ctrl.v*) inputs/outputs are sown in Table 6.6.

Table 6.6: SRAM input/output signals

Name	Type	Width	Description
addr_o	optut	18 bits	address (to the memory chip)
data_i	input	32 bits	data (from the memory chip)
data_o	output	32 bits	data (to the memory chip)
byte_sel_o	output	4 bits	byte selection (to the memory chip)
ce_o	output	1 bit	chip enable (to the memory chip)
we_o	output	1 bit	write enable (to the memory chip)
oe_o	output	1 bit	output enable (to the memory chip)
wb_clk_i	input	1 bit	clock (from the wishbone bus)
wb_rst_i	input	1 bit	reset (from the wishbone bus)
wb_dat_i	input	32 bits	data (from the wishbone bus)
wb_dat_o	output	32 bits	data (to the wishbone bus)
wb_ack_o	output	1 bit	acknowledge (to the wishbone bus)
wb_addr_i	input	32 bits	address (from the wishbone bus)
wb_we_i	input	1 bit	write enable (from the wishbone bus)
wb_sel_i	input	4 bits	byte selection (from the wishbone bus)
wb_cyc_i	input	1 bit	cycle (from the wishbone bus)
wb_stb_i	input	1 bit	strobe (from the wishbone bus)
busy	input	1 bit	busy signal (from the SDRAM controller)

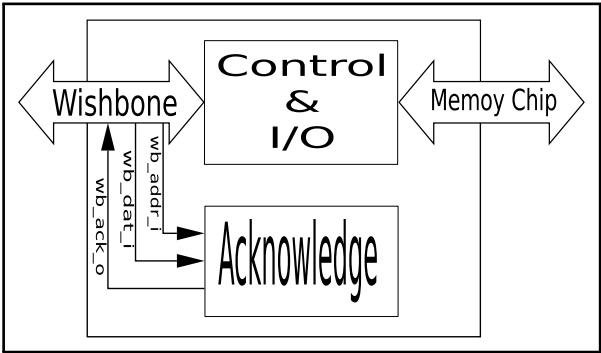


Figure 6.7: SRAM controller structure

The SRAM controller is divided in two main blocks: control & I/O and Acknowledge control (as shown in Figure 6.7).

### 6.4.1 Control & I/O Block

The control & I/O block is in charge of the control signals of the chip itself ( $\overline{WE}$ ,  $\overline{CE}$ ,  $\overline{OE}$ ,  $\overline{SEL}$ ) and the inputs/outputs (*data\_in*, *data\_out*, *address*).

To generate all the signals, this block receives the commands from the *Wishbone bus* interface. Before activating any of the signals, the block checks the address for a writing on the Acknowledge delay register.

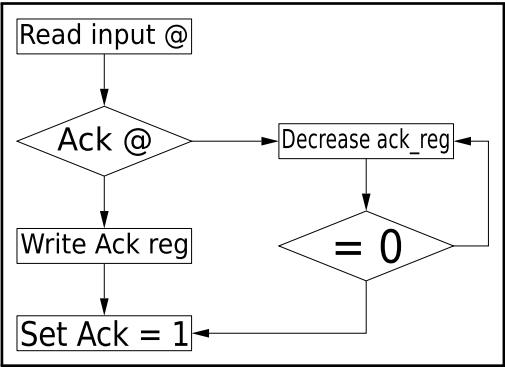
The *Acknowledge delay register* contains the number of cycles the controller should wait until activating the acknowledge signal for any writing or reading signal (*wb\_ack\_o*). If the address in a write operation points to this register, the controller will overwrite the value of the register and none of the control signals to the memory chip will be activated (as it is included in the controller architecture). When the address points to a valid memory position (not the acknowledge delay register), the controller translates the input signals from the wishbone bus into signals that will be sent to the memory controller (inverting control signals, creating  $\overline{OE}$  from *wb\_we*, etc). The signals will be static until the controller is deselected, when they will be set into their inactive values.

As the three memory controllers share the same memory interface, that could be the case when an AUTO REFRESH command from the SDRAM occurs at the same time that a SRAM reading/writing cycle. To prevent non desired behavior, the *busy* signal is asserted every time an AUTO REFRESH command happens. This signal, forces the SRAM controller to stop the current cycle while the refreshing happens and to restart it when it's finished.

### 6.4.2 Acknowledge Control Block

The acknowledge control block (as its name indicates) is in charge of the control of the acknowledge signal and register. When a write/read cycle starts, a counter starts to decrease the value stored in the acknowledge delay register. Once it reaches zero, the controller asserts the *wb\_ack\_o* to high level.





**Figure 6.8:** SRAM acknowledge control flow

When the address sent by the master is the one that points to the acknowledge delay register (address 0x20100000), the block copies the 8 LSB from the data to the *ack\_delay\_def* (default value for the acknowledge delay) and immediately asserts the acknowledge signal to high level. When the controller is unselected or a reset happens, the acknowledge register is written with its default value, all the control signal are set to inactive and the acknowledge signal is set to zero.

**Table 6.7:** SRAM delay registers default values

Register	Default value
ack_delay_def	00001010



## 6.5 Flash controller

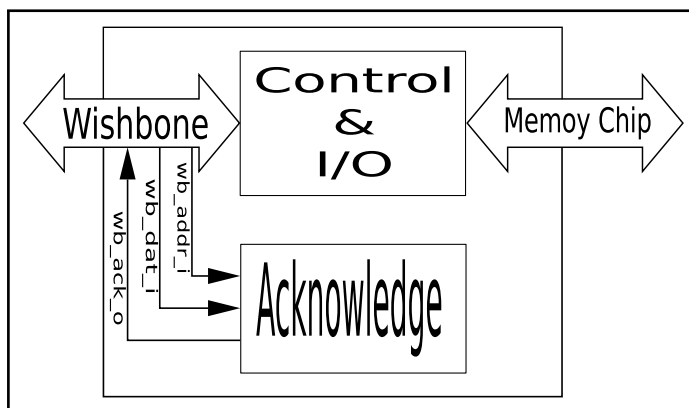
**Table 6.8:** Flash address range

Base address	Length	Offset
0x00000000	0x01000000	0xf0000000

The Flash controller inputs/outputs are shown in Table 6.9.

**Table 6.9:** Flash input/output signals

Name	Type	Width	Description
addr_o	output	23 bits	address (to the memory chip)
data_i	input	32 bits	data (from the memory chip)
data_o	output	32 bits	data (to the memory chip)
ce_o	output	1 bit	chip enable (to the memory chip)
we_o	output	1 bit	write enable (to the memory chip)
oe_o	output	1 bit	output enable (to the memory chip)
wb_clk_i	input	1 bit	clock (from the wishbone bus)
wb_rst_i	input	1 bit	reset (from the wishbone bus)
wb_dat_i	input	32 bits	data (from the wishbone bus)
wb_dat_o	output	32 bits	data (to the wishbone bus)
wb_ack_o	output	1 bit	acknowledge (to the wishbone bus)
wb_addr_i	input	32 bits	address (from the wishbone bus)
wb_we_i	input	1 bit	write enable (from the wishbone bus)
wb_cyc_i	input	1 bit	cycle (from the wishbone bus)
wb_stb_i	input	1 bit	strobe (from the wishbone bus)
busy	input	1 bit	busy signal (from the SDRAM controller)



**Figure 6.9:** Flash controller structure

The Flash controller is divided in two main blocks: control & I/O and Acknowledge control (as shown in Figure 6.9).

### 6.5.1 Control & I/O Block

The control & I/O block is in charge of the control signals of the chip itself ( $\overline{WE}$ , CE,  $\overline{OE}$ ) and the inputs/outputs (*data\_in*, *data\_out*, *address*).

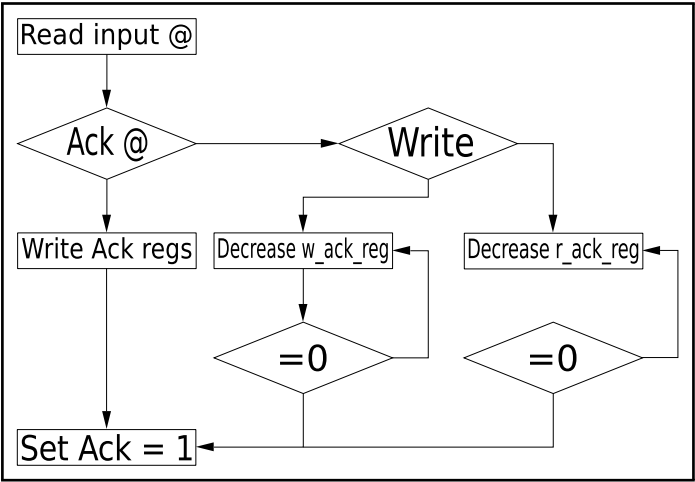
To generate all the signals, this block receives the commands from the *Wishbone bus* interface. Before activating any of the signals, the block checks the address for a write on any of the two Acknowledge delay registers (one for read and one for write operations).

The *Acknowledge delay registers* contains the number of cycles the controller should wait till activating the acknowledge signal for any write or read operation (*wb\_ack\_o*). If a write on any of these registers happened (they share the same address but not the same byte position in the data bus), the controller will overwrite the values of the registers and none of the control signals to the memory chip will be activated (as they are included in the controller architecture). When an address points to a valid memory position (not the acknowledge delay registers), the controller translates the input signals from the wishbone bus into signals that will be sent to the memory controller (inverting control signals, creating  $\overline{OE}$  from *wb\_we*, etc). The signals will be static until the controller is deselected, when they will be set into their inactive values.

As the three memory controllers share the same memory interface, that could be the case when an AUTO REFRESH command from the SDRAM occurs at the same time that a Flash read/write cycle. To prevent non desired behavior, the *busy* signal is asserted every time an AUTO REFRESH command happens. This signal, forces the Flash controller to stop the current cycle while the refreshing happens and to restart it when it's finished.

### 6.5.2 Acknowledge Control Block

The acknowledge control block (as its name indicates) is in charge of the control of the acknowledge signal and registers. When a write/read cycle starts, a counter starts to decrease the value stored in the acknowledge delay register (*ack\_delay\_w* for write operations and *ack\_delay\_r* for read operations). Once it reaches zero, the controller asserts the *wb\_ack\_o* to high level.



**Figure 6.10:** Flash acknowledge control flow

When the address sent by the master is the one that points to the acknowledge delay registers (address 0xf1000000, same address for both of them), the block copies the 8 LSB form the data to the *ack\_delay\_def\_r* (default value for the read acknowledge delay) and the 16-9 LSB form the data to the *ack\_delay\_def\_w* (default value for the writing acknowledge delay) and immediately asserts the acknowledge signal to high level. This means that whenever a register is written, the other one should be too. When the controller is unselected or a reset happens, the acknowledge registers are written with their default value, all the control signals are set to inactive and the acknowledge signal is set to zero.

**Table 6.10:** Flash delay registers default values

Register	Default value
ack_delay_def_w	01000000
ack_delay_def_r	00011101



## 6.6 AvBus

As the three memory modules share many of the signals that compose the AvBus connector an interface is needed to avoid unexpected behaviors. This are the shared signals:

- Address bus
- Data bus
- Write enable
- Output enable
- Byte selection

To efficiently control the way the output signals are assigned we need to establish some kind of hierarchy between the memory controllers. The SDRAM controller has to be at the top, the AUTO REFRESH command can not be postponed or the data stored in the SDRAM might be corrupted or even lost.

Between the other two controllers (SRAM and Flash), things are more or less at the same level, but always below the SDRAM controller. Any time the SDRAM controller needs to access the memory modules it needs to have access to the shared signals even if they are used by any of the other two controllers. That is possible by the use of the busy signal (*sdr\_cke* signal from the SDRAM). Any time the SDRAM controller accesses to the AvBus connector shared signals, the busy signal is set to high level and the other controllers (in case they were reading or writing the memory modules) will stop their current action and restart it once the busy signal is set to low level (once the SDRAM had finished its actions). That gave us the confidence that even when an SDRAM action is taken at the middle of another controller actions, its data will not be lost.

### 6.6.1 AvBus Interface

This is the most critical section of the whole design. As it is placed between the memory controllers and the memory modules, the slightest delay can turn into a non desired memory behavior.

To ensure all the signals are assigned as quick as possible, all the selections have been distributed in small case statements (for the shared signals) or straight assignments. The control signals for the case elements are both SRAM and Flash chip select signals and the SDRAM cke signal. With this three signals we can ensure that all the possible situations regarding the three controllers are covered:

- SRAM memory access
- Flash memory access
- SDRAM memory access
- SDRAM and SRAM memory access
- SDRAM and Flash memory access

The last two situation can occur when an AUTO REFRESH command happened in the middle of a SRAM or Flash memory access. In this two cases, the *sdr\_cke* signal (used as busy signal) ensures the integrity of the SDRAM stored data.

The *avbus\_addr\_o* signal, when connected to the memory modules, skips the 2 LSB. In the SRAM an Flash cases, it does not affect the behavior of the memory modules, the Flash memory controller always reads and writes 2 byte words and with the *avbus\_bs\_o* signal the SRAM and SDRAM memory controllers can control which bytes are written or read.

In addition to the interface itself, a small control must be set for controlling the two buffers between the AvBus connector and the SRAM and Flash memory modules (shown in Figure 4.5). The address buffer must be only enabled by setting its control value to zero (*MABUF\_OE\_o*), because its direction is always the same (from the connector to the memory modules) and its direction control value is fixed.

The data buffer must be enabled by setting its control value to zero (*MDBUF\_OE\_o*) and depending on the action that will be performed, the direction of the buffer must be changed from the module to the connector (reading operation, *MABUF\_DIR\_o* set to zero) to from the connector to the module (writing operation, *MABUF\_DIR\_o* set to one).

Apart from the input/signals from the three memory controllers (which are connected to the AvBus I/F), the board input/output signals of the AvBus interface are shown in Table 6.11.

**Table 6.11:** AvBus I/F board input/output signals

Name	Type	Width	Description
<i>avbus_addr_o</i>	output	32 bits	Address output signal
<i>avbus_data_io</i>	inout	32 bits	inout signal
<i>avbus_sdram_cs_o</i>	output	1 bit	SDRAM chip select output signal
<i>avbus_sram_cs_o</i>	output	1 bit	SRAM chip select output signal
<i>avbus_flash_cs_o</i>	output	1 bit	Flash chip select output signal
<i>avbus_oe_o</i>	output	1 bit	Output enable output signal
<i>avbus_we_o</i>	output	1 bit	Write enable output signal
<i>avbus_cas_o</i>	output	1 bit	CAS output signal
<i>avbus_ras_o</i>	output	1 bit	RAS output signal
<i>avbus_clken_o</i>	output	1 bit	Clock enable output signal
<i>avbus_sdram_clk_i</i>	input	1 bit	SDRAM clk input signal
<i>avbus_sdram_clk_o</i>	output	1 bit	SDRAM clk output signal
<i>avbus_bs_o</i>	output	4 bit	Byte select output signal
<i>MDBUF_OE_o</i>	output	1 bit	Data buffer output enable output signal
<i>MDBUF_DIR_o</i>	output	1 bit	Data buffer direction output signal
<i>MABUF_OE_o</i>	output	1 bit	Address buffer output enable output signal

The assignment of the output data to the memory controllers is done by direct assignment from the *avbus\_data\_io* tri-state.



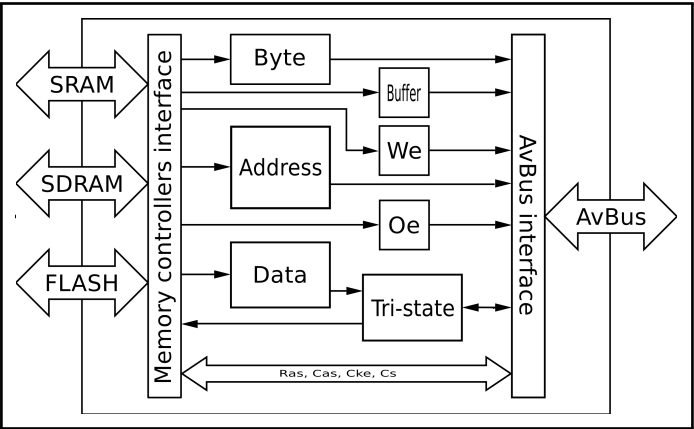


Figure 6.11: AvBus interface structure



## 6.7 UART

The board has a RS232 DB9 connector with allows us to use it for a UART. The UART IP core which has been implemented, has been developed by Jacob Gorban and it is available in Open Cores ([www.opencores.org](http://www.opencores.org)). The IP core is free hardware and it is distributed under the terms of the GNU Lesser General Public License.

The main features of the UART core are:

- WISHBONE interface in 32-bit or 8-bit data bus modes (selectable)
- FIFO only operation
- Register level and functionality compatibility with NS16550A (but not 16450).
- Debug Interface in 32-bit data bus mode.

To configure the UART to work with the parameters we need, we should perform the following tasks in order:

- Set the Line Control Register to the desired line control parameters. Set bit 7 to 1 to allow access to the Divisor Latches.
- Set the Divisor Latches, MSB first, LSB next.
- Set bit 7 of LCR to 0 to disable access to Divisor Latches. At this time the transmission engine starts working and data can be sent and received.
- Set the FIFO trigger level. Generally, higher trigger level values produce less interrupt to the system, so setting it to 14 bytes is recommended if the system responds fast enough.
- Enable desired interrupts by setting appropriate bits in the Interrupt Enable register.

It is important to point that the value of the Divisor Latch responds to Equation 6.1.

$$\text{Baud rate} = \frac{\text{Input Clock Frequency}}{16 \cdot \text{Divisor Latch Value}} \quad (6.1)$$

Every time there is a reset applied to the core the following actions are performed:

- The receiver and transmitter FIFOs are cleared.
- The receiver and transmitter shift registers are cleared
- The Divisor Latch register is set to 0.
- The Line Control Register is set to communication of 8 bits of data, no parity, 1 stop bit.
- All interrupts are disabled in the Interrupt Enable Register.

That means that every time a reset is applied to the core, the initialization procedure must be done again in order to behave as expected.

For further information about the UART module please check the data sheet [7].

## 6.8 Boot Monitor

The boot monitor is a special firmware stored in the FPGAs block RAMs which provides the initialization of devices (the UART) and a basic GUI to allow the user to do basic memory manipulation and serving as boot platform for other operative systems stored in the FLASH memory or other storage devices.

The boot monitor was developed for the Dafk system [1] by Olle Seger, and as it has the same CPU and UART, it can be reused in this system.

The address of the boot monitor is stored in the defines file of the CPU (*or1200\_defines.v*)

## 6.9 Clock Manager

A small clock manager, created using a DCM, is used to generate the system clock and the SDRAM clock, which is a 180 degrees shifted version of the system clock.



# 7

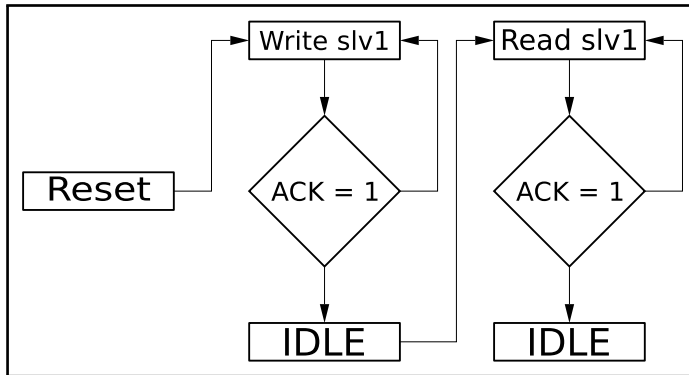
---

## Tests

To verify the SoC, several tests have been made. The initial tests were made on the simulator in order to check if the system behaves as expected. First module by module and then several of them conforming a more complex system that in the last test was the SoC itself. The hardware tests have been done the same way. First separated modules, and then, step by step, more complex elements that ended with the SoC.

### 7.1 Simulation Tests

The structure of the test (without considering the number of modules tested) has always been the same. All the modules receive the data coming from the testbench have a wishbone interface. The testbench has been made as a fake wishbone bus master (excepting the case when the whole SoC was tested, where no testbench was needed, only clock and reset signals) which sends write and read commands. The testbench structure, shown in 7.1, was implemented using a FSM. Triggered by the reset, the whole test process was automated. Loading the *.do* files, the waveforms of the most important signals can be seen before the test was finished. In case there were more than one slave to be tested, the basic structure of the testbench (from Write slv1 to the section on IDLE) can be duplicated changing the target addresses to point to another slave.



**Figure 7.1:** Testbench basic structure

## 7.1.1 Single Tests

### SRAM

The SRAM memory controller was tested to check the acknowledge signal delay and the translation from the wishbone input signals into the sram output signals.

### SDRAM

The SDRAM memory controller was tested to check the translation of the wishbone signals into the SDRAM output signals. Also all the control signals and the AUTO REFRESH period were tested.

### Flash

The Flash memory controller was tested to check the two acknowledge signals delay and the translation from the wishbone input signals into the flash output signals.

### Wishbone Bus

The wishbone bus was tested to check the arbiter and decoder capability to decode the selected master and slave depending on the input address. In addition, the scenario of both masters requesting a bus cycle was also tested.

### UART

Initialization sequence and character sending were tested.

## 7.1.2 Group Tests

In order to test write and read operations without having to synthesize the memory controllers, some models of the memory modules present in the *Avnet Communications/Memory module* were used.



### Memory models

The SRAM memory model used is contained in the files *Sram\_1mb.v* and *128Kx8.v*. The SDRAM memory model was contained in the files *sdram.v* and *mt48lc16m16a2.v*.

#### SRAM + SRAM Model

Read and write operations were tested.

#### SDRAM + SDRAM Model

Read and write operations were tested, also the AUTO-REFRESH period and other commands.

#### SRAM + AvBus I/F + SRAM Model

Faking the input signals from the SDRAM and Flash controller, the AvBUS I/F output signals assignments were tested.

#### Wishbone bus + SRAM +AvBus I/F + SRAM Model

Using one of the Master's wishbone inputs, read and write operations were tested as well as the correct signal assignments in the wishbone bus.

#### Wishbone bus + SRAM + SDRAM +AvBus I/F + SRAM Model + SDRAM Model

In this test, several situations were tested.

- SRAM read and write
- SDRAM read and write
- SDRAM AUTO-REFRESH command in the middle of a SRAM write operation
- SRAM and SDRAM operation request from different masters at the same time

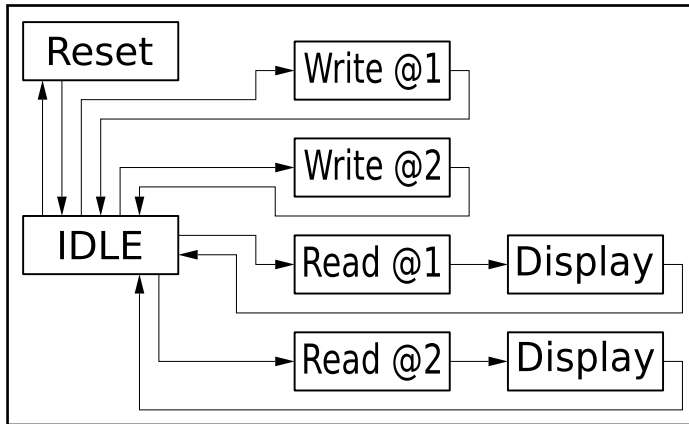
### SoC

Including the memory content files of the boot monitor to the rest of the modules of the system, the start sequence was tested.

## 7.2 Hardware Tests

The structure of the hardware tests resembles the testbenches of the simulation tests. Both of them have a fake wishbone master that feeds the slave modules, but instead of having the whole test process automated, its done in steps triggered by the keys available on the FPGA board (add-on board made by the department of Electrical Engineering).

The reason of not having the whole process automated is because to show the read data (32 bits) there are only 8 LEDs available on the board. To show the entire data length, once is read from the memory, the test switches into display state, where, controlled by 4 of the 6 keys, the user can select which byte of the 4 will be displayed on the LEDs. Once the results have been checked, with other key, the user can go back to the IDLE state.



**Figure 7.2:** Hardware test structure

In order to control when the reads and writes ends, the acknowledge signal is also shown in the read and write states. Once the acknowledge signal is asserted, the user can continue to the IDLE state (write operations) or to the display state (read operations).

### SRAM tests

First, the SRAM memory controller was synthesized and tested alone. Next the AvBus I/F was added and finally the Wishbone Bus. In all the test, read and write operations were tested in two different addresses to ensure that neither the buffers or the internal registers of the AvBus I/F stored the data sent, faking the reads with previously sent and stored data.

### SDRAM tests

The SDRAM memory controller was synthesized with all the elements present in the SRAM hardware test. In this test, the only element tested, was the capability of the AvBus interface to deal with two controllers at the same time. While the operations of the SRAM were being executed, the SDRAM controller was making AUTO REFRESH commands, which forced the SRAM controller and the AvBus interface to deal with the memory controllers hierarchy.

Due to the complexity of the SDRAM memory controller, the rest of the test were performed once the whole SoC was implemented. The use of a terminal to display the data loaded from the SDRAM memory modules was the best way to find errors and unexpected behaviors.

### SoC tests

In this test, the functionality of the SDRAM memory controller and whole system were tested. As there was a CPU and the essential peripherals to work as a real system, test were no longer made of HDL elements or fake wishbone bus masters. In this case, the test were performed by the use of terminal commands or programs written in C.

The use of the UART allows the use of a text based user interface provided by the Boot Monitor, which makes the basic operations (read, write and copy memory contents) much easier for the user and brings out the possibility of more complex test.

The SDRAM memory controller was tested at first with basic memory manipulation commands. The same was done with the SRAM and Flash controller. Using terminal commands, several positions of the different memory modules were written, read and copied.

The possibility of loading C programs, allows testing the whole SoC as a real system. Arithmetical operations can be used along with pointers, printf, loops, counters and almost any possibility supported by C language.

As the CPU is not a regular X86 architecture, some special libraries need to be added to the C files along with some memory positions specifications to point the addresses where the code and its related elements are going to be placed.

The file is loaded into the system as an hexadecimal file. To compile and transform the standard C file into the .hex file that can be interpreted by the system there is a makefile placed in the folder *simpleprog*. This makefile takes as input the file *simpleprog.c* that contains the test and generates the *simpreprog.hex* file that can be loaded into the system.

The basic structure of the SoC tests written in C is the following:

1. Take 3 numbers, store them in three memory positions (in the SRAM or SDRAM)
2. Load the three numbers, make some operations between them and store the results in three different memory positions (not necessarily in the same memory module)
3. Load the results, combine them into one final number and display it using the return command
4. At the same time, display a message using the printf function

This basic structure can be modified to support more complex operations and therefore more complex tests.



# 8

---

## Results

All the simulation tests were successful, the modules behave as expected and no big errors were found.

In the hardware tests, all the modules worked as expected excepting the Flash memory controller and the AvBus I/F.

The AvBus interface had showed as the critical element of the design. When only the SDRAM and the SRAM modules were working together, the results were as expected. But when the Flash memory controller was added, both the SRAM and the Flash memory controllers did not work as expected. The problem is somehow related with the use or not of the Flash chip select signal. When is not used, the SRAM works well (the Flash does not as the memory chip is not enabled), but when is used, both controllers behave in an unexpected way. This issue has to be studied in detail, because there is a problem that is not shown in the simulation tests and can be related with the speed of the AvBus I/F module. This problem is probably related with the hierarchy of the memories. The SDRAM actions can not be stopped, but at the same time, some Flash operations can not be stopped too, so that might be the problem behind the non desired behavior. Nevertheless, it should be studied in detail in future revisions of the platform.

When only the SDRAM and the SRAM memory controllers where used, the SoC test were successful. Data was written in the correct positions and the results and the text displayed at the terminal were correct. This happened regardless of the memory module where the programs were loaded.

Synthesizing the SoC and the different modules separately, hints of the maximum clock frequencies were obtained.

- SRAM memory controller  $f_{max} = 226$  Mhz.

- Flash memory controller  $f_{max} = 204$  Mhz.
- SDRAM memory controller  $f_{max} = 47$  Mhz.
- CPU  $f_{max} = 71$  Mhz.
- UART  $f_{max} = 135$  Mhz.
- Wishbone  $f_{max} = 518$  Mhz.
- AvBus I/F  $f_{max} = \text{none}$  (there is no clock signal).
- System  $f_{max} = 35/71$  Mhz. (depending on the synthesizer used)

Those values were obtained without setting any timing constraint. As there were no desired working frequencies set, the synthesizer did not push to obtain the maximum possible frequency. Nevertheless, these frequencies are a ballpark figure of the real frequencies.

Regarding to the FPGA utilization, the result obtained after the place and route operations are shown in Table 8.1 and Table 8.2.

**Table 8.1: FPGA logic utilization**

Number of Slice Flip Flops	2,416 out of 46,080 (5%)
Number of 4 input LUTs	7,126 out of 46,080 (15%)

**Table 8.2: FPGA logic distribution**

Number of occupied Slices	4,072 out of 23,040 (17%)
Number of Slices containing only related logic	4,072 out of 4,072 (100%)
Number of Slices containing unrelated logic	0 out of 4,072 (0%)
Total Number of 4 input LUTs	7,209 out of 46,080 (15%)
Number used as logic	6,890
Number used as a route-thru	83
Number used for Dual Port RAMs	236 (Two LUTs used per Dual Port RAM)
Number of bonded IOBs	154 out of 824 (18%)
IOB Flip Flops	134
Number of RAMB16s	36 out of 120 (30%)
Number of MULT18X18s	4 out of 120 (3%)
Number of BUFGMUXs	2 out of 16 (12%)
Number of DCMs	1 out of 12 (8%)

As the reader can notice, even having a full system synthesized, the use of the FPGA resources is not very high.

# 9

---

## Conclusions

Looking again at the primary goal this project has been trying to accomplish, I think it is fair to say that it has been accomplished.

I have developed the basic system to be used in the future as a versatile test platform and, despite it can be improved in many ways, the base platform is done.

It has the most updated version of the *OR1200* CPU and will be able to run Linux in the future. The memory controllers are now separated and the SDRAM memory controller now supports burst.

The system, once integrated in the NoGap platform, will be able to perform any kind of test that can be described using C or C++ programming languages. At the same time, it will not depend on external computational elements to process the results of the performed tests. And with the use of the Wishbone interconnection standard it ensures that if in the future, new features are required, they can be added in an easy way.

As the objective was, this features are the ones that define a versatile platform, and that at the end was the reason of this project.





# 10

---

## Future Work

The possible ways of continuing with this project can be divided in two. On one side the improvements to the main system, that can increase the versatility of the system, and in the other side, the future goals related with the integration of the system in the NoGap platform.

### 10.1 System Improvements

One good way of improving the performance of the system without having to add any extra module is to modify the SRAM and Flash controllers to support burst mode. It can be easily done by adding a FIFO in the controllers and some logic to check the *wb\_cti* signal for burst start and burst end.

A good way to improve the SDRAM memory controller can be to store the configuration parameters in the SRAM or Flash memory, being able to modify them while running the system and loading them any time a reset is applied.

Another aspect that needs to be studied in detail is the behavior of the Flash memory controller. Probably a readjustment of the AUTO REFRESH command periods of the SDRAM memory controller and a better AvBus I/F are going to be needed.

There is also some modules that can be added in order to increase the available features and give the system more versatility.

- An Ethernet module can offer quicker communications based on a very common protocol (IP) and the possibility of using resources stored in an FTP server, remote control, internet access, etc..
- A DDR SDRAM memory controller can boost the performance of the primary mem-

ory of the system

- A SD card module (using the add-on board) can provide the system of a flexible and portable storage solution for tests, results, OS hosting, etc..
- A PCMCIA controller can be used to add multiple peripherals

## 10.2 Future Goals

The primary goal of the system is, of course, to be integrated in the NoGap platform to provide a flexible hardware platform for testing ASIPs and accelerators. To accomplish that goal, it is necessary to add a wrapper generator to the NoGap platform, to be able to establish a communication between the test platform and the design to be tested.

Another goal that can improve the system capabilities of testing is the support of a UNIX operating system. The *OR1200* CPU is supported in the latest versions of the Linux kernel, so only the slaves modules might need to be tested for incompatibilities. By doing this, the range of possibilities will be increased widely.

---

## Bibliography

- [1] TSEA44 web page, . URL <http://www.da.isy.liu.se/courses/tsea44/>.
- [2] Gaisler LEON web page. . URL <http://www.gaisler.com/cms/index.php?option>
- [3] Accelera. SystemVerilog. URL <http://www.systemverilog.org/>.
- [4] Dinesh Annayya. *SDRAM controller specification*, February 2012.
- [5] Julius Baxter and Damjan Lampret. OpenRISC 1200 IP Core Specification. September 2011.
- [6] Per A. Carlström. *NOGAP: Novel Generator of Accelerators and Processors*. PhD thesis, Linköping University, 2010.
- [7] Jacob Gorban. UART IP Core Specification. August 2002.
- [8] Iee. 1076-2008 IEEE Standard VHDL Language Reference Manual. Technical report, 2009.
- [9] Ieee. 1364-2005 IEEE Standard for Verilog Hardware Description Language. Technical report, 2006.
- [10] Ieee. 1800-2009 IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. Technical report, 2009.
- [11] Intel. Intel StrataFlash Memory data sheet. March 2005.
- [12] Damjan Lampret, Rohit Mathur, Jeanne Wiegelmann, and Marko Mlinar. OpenRISC 1000 Architecture Manual. April 2006.
- [13] Opencores. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, September 2002.
- [14] Avnet D. Services. *Xilinx Virtex-II Development Kit*, November 2002.
- [15] Avnet D. Services. *Communications/Memory Module*, November 2002.
- [16] Micron Technology. *MT48LC16M16A2 SDRAM data sheet*, March 2002.
- [17] Wikipedia. Verilog. . URL <http://en.wikipedia.org/wiki/Verilog>.

- 
- [18] Wikipedia. Vhdl. . URL <http://en.wikipedia.org/wiki/Vhdl>.
  - [19] Wikipedia. Hadware Description Languages. . URL [http://en.wikipedia.org/wiki/Hardware\\_description\\_language](http://en.wikipedia.org/wiki/Hardware_description_language).
  - [20] Xilinx. *Virtex II family documentation*, 2007.

## Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

## Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>