ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ

SPARTAN-3 FPGA WITH PCI INTERFACE FOR MASS MEMORY PCB-TESTER

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Λάμπρος Μπερσεντές

Επιβλέπων : Κ.Πεκμεστζή
Καθηγητής ΕΜΠ

Athens, October 2010

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ

SPARTAN-3 FPGA WITH PCI INTERFACE FOR MASS MEMORY PCB-TESTER

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Λάμπρος Μπερσεντές

Επιβλέπων : Κ.Πεκμεστζή
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την .....
.

……..	……..	……..
Κ. Πεκμεστζή	Δ. Σουντρής	Γ. Οικονομάκος
Καθηγητής Ε.Μ.Π.	Επ.Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Athens, October 2010

……..
Μπερσεντές Λάμπρος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η διπλωματική εργασία επικεντρώνεται στην περιοχή του FPGA. Ένα FPGA design κατασκευάστηκε το οποίο προσομοιώνει το χρονισμό ενός μικροεπεξεργαστή. Χρησιμοποιώντας αυτό το design ειναί δυνατή η διέγερση διάφορων PCBs που είναι συνδεδεμένα με την κάρτα. Τα PCBs χρησιμοποιούνται για πραγματικού χρόνου και μακράς διάρκειας έλεγχο.

Ένα module κατασκευάστηκε μέσα στο FPGA design που μπορεί να λειτουργήσει σε δύο καταστάσεις, είτε ως master είτε ως slave. Ως master, τα σήματα που εκπέμπονται προσομοιώνουν τη λειτουργία ενός ERC32 μικροεπεξεργαστή, ενώ οι τιμές των σημάτων εξαρτώνται από τις εντολές τύπου assembly που δίνει ο χρήστης. Αυτές οι εντολές μεταφέρονται μέσω του PCI Interface στο FPGA. Ως slave, το FPGA design δεν προσομοιώνει πλέον τη λειτουργία ενός ERC32 μικροεπεξεργαστή, αλλά λαμβάνει σήματα που έχουν δημιουργηθεί από έναν τέτοιο μικροεπεξεργαστή. Μπορεί να «μεταφράσει» τα σήματα και να εκτελέσει την λειτουργία που αυτά τα σήματα υποδηλώνουν.

Ένα software interface αναπτύχθηκε, το οποίο δίνει απευθείας και εύκολη πρόσβαση στο νέο module, που έχει υλοποιηθεί μέσα στο FPGA.

Λέξεις Κλειδιά

FPGA design,Χρονισμός μικροεπεξεραστή, Προσομοίωση μικροεπεξεργαστή, Έλεγχος PCB, Εφαρμογές πραγματικού χρόνου,Channel-link,Space-Wire.

Abstract

The thesis deals with FPGA design. More concretely, a FPGA design is created, which simulates a Microprocessor's timing. Using this FPGA design, the stimulation of slave PCBs, which are connected on the board is possible. The PCBs are used for real time and long term tests.

Inside the FPGA design the module which is implemented can work in two modes, master and slave. In master mode, the signals sent out, will simulate the work of an ERC 32 microprocessor and their values will depend on the input "Assembly type" commands given from the user. These commands are transferred through the PCI Interface to the FPGA. In slave mode, the FPGA design doesn't simulate any more the work of an ERC 32 microprocessor but it receives signals which are produced by such a microprocessor. It can translate these signals and it can carry out the operation these signals imply.

Finally, a user interface is developed, which gives a direct and easy access to the new module, which is implemented inside the FPGA.

Keywords

FPGA design, microprocessor timing, microprocessor simulation, PCB testing, real-time applications, Channel-link,Space-Wire.

Ευχαριστίες

Ευχαριστώ τους γονείς μου και την αδερφή μου που στέκονται πάντα πλάι μου, την εταιρεία EADS Astrium και ιδιαίτερα το κ. Oliver Birk για την πολύτιμη βοήθεια τους καθόλη τη διάρκεια περάτωσης της διπλωματικής καθώς και τον καθηγητή μου κ.Πεκμετζή για τη συνεργασία μας.

Table of contents

List of figures

List of Tables

List of Codes

1. Introduction

1.1 Main points and benefits of the research

The MSS timing is simulated depending on the commands which are executed. The Input commands are sent to the MSS module where they are processed. These commands are constructed by the user and they will be simply read out from a *.txt file by C#-SW. These commands will be sent to the board through PCI and they will have the following form:

Command	Address(Hex)	Data(Hex)
WR	0005	FFAD

The ERC32 is one of the few microprocessors, which is available in radiation tolerant technology, and therefore applicable for space electronics.

The FPGA on the Dig-IO Board is part of a PCB Test system, which will be used to stimulate as many interfaces of the DUT as possible in an early phase. One of these interfaces is the microprocessor (ERC32) interface. Later, the pretested PCB is integrated in the PDHU (Payload Data Handling Unit) box and connected to the real ERC32 Microprocessor. The implementation shall just simulate the access of the ERC32 Microprocessor to the slave PCB (which has to be tested). It shall provide an easy way to generate read and write access to the DUT without having the processor system running. It is not intended to have a full replacement of the ERC32 by this test system.

The **benefit** of this work shall be, that the microprocessor interface can behave exactly like the real microprocessor. This allows a seamless integration of the PCB into the box with no timing problems. There is no need to have application software on the tester. What is more, the ***real time applications*** in which this card will be used makes this card useful as it cannot be replaced by typical I/O-Cards which have slower data-rates and which therefore cannot support such applications. The design shall be done in a way, that the particular microprocessor timing can be exchanged easily if another processor is used in the project.

The following figure gives an overview of all stages which have to be accomplished.

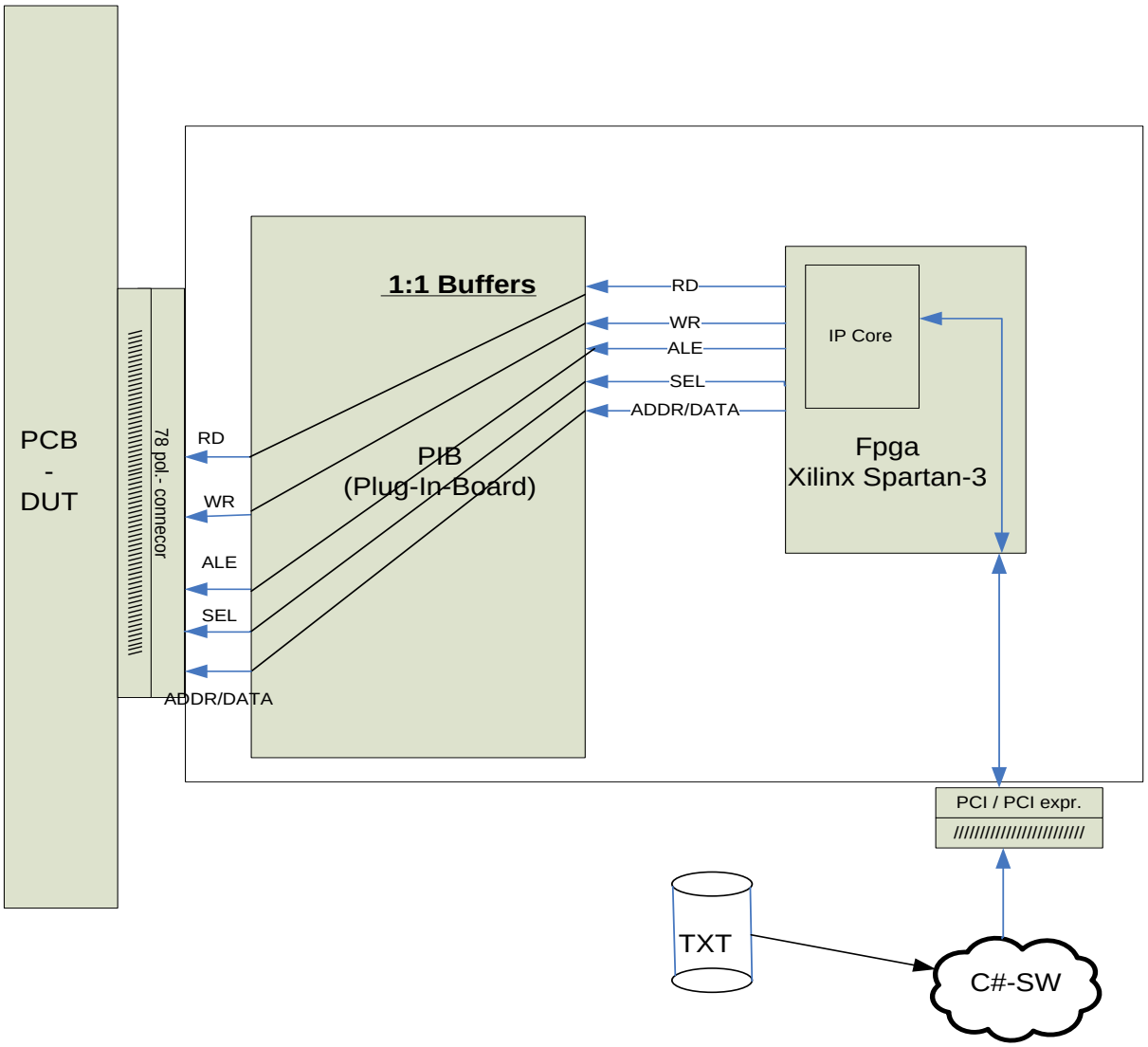


Figure 1 Plan of the PCI Card

1.2 General Information about FPGAs

F*ield*-P*rogrammable* G***ate*** A*rray is* a type of logic chip that can be programmed. An FPGA is similar to a PLD, but whereas PLDs are generally limited to hundreds of gates, FPGAs support thousands of gates. They are especially popular for prototyping integrated circuit designs. Once the design is set, hardwired chips are produced for faster performance. FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, partial re-configuration of the portion of the design and the low non-recurring engineering costs relative to an ASIC design (not withstanding the generally higher unit cost), offer advantages for many applications.FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"—somewhat like a one-chip programmable breadboard. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

Applications of FPGAs include digital signal processing, software-defined radio, aerospace

and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of DSPs began to incorporate FPGAs instead.

FPGAs especially find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture. One such area is code breaking, in particular brute-force attack, of cryptographic algorithms.

FPGAs are increasingly used in conventional high performance computing applications where computational kernels such as FFT or Convolution are performed on the FPGA instead of a microprocessor.

The inherent parallelism of the logic resources on an FPGA allows for considerable computational throughput even at a low MHz clock rates. The flexibility of the FPGA allows for even higher performance by trading off precision and range in the number format for an increased number of parallel arithmetic units. This has driven a new type of processing called reconfigurable computing, where time intensive tasks are offloaded from software to FPGAs.

The adoption of FPGAs in high performance computing is currently limited by the complexity of FPGA design compared to conventional software and the turn-around times of current design tools.

Traditionally, FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications.

1.3 General Information about the PCI card

Nowadays, the high demands for rapid development speed and flexibility in hardware design are responsible for the fact that the PCIS3BASE board is so popular. On PCIS3BASE board there is a FPGA installed, which belongs to SPARTAN-3 family. The 93 I/O Balls of this FPGA are routed to the expansion connector of the Plug-In-Board(PIB) slot ,which on the other side has connections to a 78-pin HD-SUB I/O Connector. This HD-SUB I/O Connector makes possible the attachment of external Hardware to the FPGA.

PIBs carry a functionality, which may be specific depending on the board will be installed on or can be even defined from an engineer personally according to the job he wants to carry out. Plug-In-Boards can carry various interfaces such as ADC, DAC, TTL LEVEL I/O, RS232, RS485, LVDS, Camera Link or user-defined interface standards.

Apart from the FPGA and the PIB on PCIS3BASE board there are 32 MByte SDRAM, Serial Flash Memory, a bus-master PCI bridge on board and JTAG interface. Each of these interfaces will be described thoroughly.

A 50 MHz clock oscillator supplies the basic clock that can be used by the FPGA. Besides, additional clock sources can be present on PIBs.

It should be underlined, that the PCI Interface is not implemented inside the FPGA, in other words no PCI IP-Core is needed. However, a PCI-Bridge Chip is responsible for this implementation. Spartan-3 FPGA connects to its local bus and this is how the communication between the PC and the FPGA is possible.[1]

1.4 General Information about PCI Technology

During the early 1990s, Intel introduced a new bus standard for consideration, the Peripheral Component Interconnect (PCI) bus. It provides direct access to system memory for connected devices, but uses a bridge to connect to the front side bus and therefore to the CPU.

Front side bus is actually a physical connection between the processor and the other hardware components such as RAM, hard drives and PCI Slots. The PCI bridge chip is responsible for the speed regulation of the PCI bus independently of the CPU's speed. This provides a higher degree of reliability and ensures PCI-hardware manufactures know exactly what to design for.

Generally, PCI operates at 33 MHz using a 32-bit local bus. PCI cards use 47 pins to connect to the bus. The PCI bus is able to work with so few pins due to hardware multiplexing. That means, the device sends more than one signal over a single pin. What is more, PCI supports devices that use either 5 Volts or 3.3 Volts.

Plug and Play (PnP) is a main feature that made PCI cards really popular since Windows 95 OS released as this was the first OS which supported this technology. Plug and Play means that once you connect a device or insert a card into your computer, it is automatically recognized and configured to work in your system.

The new breakthrough idea in computer bus technology which is likely to replace PCI as it is nowadays, is the **point-to-point switching connection.** AMD took into advantage this idea and leaving behind the shared bus technology, developed the Hypertransport**,** a new standard in which, for each session between nodes two point-to-point links are provided. Intel introduced the new version of PCI, which implements a point-to-point switching connection. Its name is PCI-Express and seems to be the new cutting edge in computer technology as its benefits can bring a revolution not only in the performance of computers, but also the very shape and form of home computer systems.[2]

[1] http://www.cesys.com/resources/CE031.pdf
[2] http://computer.howstuffworks.com/pci.htm

2. Hardware on the card

PCIS3BASE board consists of the following Hardware features:

• XILINX Spartan-3 FPGA 1.5 MIO system gates (XC3S1500-4FGG456C)

• PCI host bridge supports 3,3 Volt and 5 Volt PCI (PLX PCI9056BA66)

• High performance, up to 120 MByte/s data rate on PCI bus possible

• 32 MByte SDRAM (MICRON 48LC16M16A2)

• SPI Serial Flash Memory 4 MBit (512 KBytes x 8)

• PCI 2.1 compliant device (Plug-and-Play)

• 78-pin external I/O connector

• PIB64IO board included (64 I/O signals on ext. I/O connector, 5 Volt TTL)

• Allocated space for plug-in-board with two 100 pin connectors

• Internal expansion port RM 2,54 mm (28 I/O pins)

• 8 Leds connected to the FPGA

• JTAG connector for debugging and configuration

• Driver for Windows XP and test-program included

2.1 Xilinx Spartan-3 FPGA

Spartan 3 family of FPGAs is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. This specific FPGA has a density of 1.5 Million system gates. Spartan 3 family in comparison with its ancestor Spartan-IIE family has an increased amount of logic resources, increased capacity of internal RAM and improved clock management functions. The previous enhancements combined with advanced process technology make Spartan-3 FPGAs really popular and what is more, they set new standards in the programmable logic industry.[3]

The most important feature of this FPGA family is their exceptionally low cost. That makes them ideally suit to a wide range of consumer electronics applications such as home networking, digital television equipment etc.

[3] http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf

SPARTAN-3 FPGA which is used on the board has the following features:

Device	XC3S1500-4FGG456C
System Gates	1500k
Configurable Logic Blocks	64 x 52
Logic cells	29,952
Block Ram Bits	576k
Distributed Ram Bits	208k
DCMs	4
Multipliers	32

Table 1 Spartan-3 FPGA features

All FPGA VCCO-Pins on the board are connected to 3.3 Volt. It must be taken into account that higher Voltage signals (5 Volt for instance) would cause the destruction of the FPGA. Even 3.3 Volt signals with long traces or cables in conjunction with improper termination resulting overshoot and undershoot can be destructive as well.

2.2 Cesys PIB Slot

This PIB Slot consists of two 100-pin connectors. The one connector is wired to FPGA I/O Balls while the second one is wired to an external 78-pin HD-Sub connector. The Plug-In-Board (PIB64IO) which comes as standard with PCIS3BASE consists of 64 IOs with 5V tolerant buffers.

2.3 LEDs

The PCIS3BASE is equipped with eight LEDs, four green and four yellow. Upon successful configuration the LEDs light up and stay on providing that the device is configured.

Below the connections between the **LEDs** and the corresponding **FPGA I/O balls** are shown:

LEDs	***Comment***
LED1 Green	FPGA I/O Ball U2
LED2 Green	FPGA I/O Ball U3
LED3 Green	FPGA I/O Ball U4
LED4 Green	FPGA I/O Ball U5
LED1 Yellow	FPGA I/O Ball V1
LED2 Yellow	FPGA I/O Ball V2
LED3 Yellow	FPGA I/O Ball V3
LED4 Yellow	FPGA I/O Ball V4
CFG LED	Configuration LED

Table 2 LED-FPGA Connections

2.4 Plug-In-Board Connectors

On the PIB there are two 100-pin external expansion connectors of type female.

An overview of the connections between the two connectors and other interfaces on the board follows below.

PIN	Signal name	FPGA I/O	PIN	Signal Name	FPGA I/O
1	PIB_IO 0	A14	51	PIB_IO 43	T6
2	PIB_IO 1	B14	52	PIB_IO 44	T5
3	PIB_IO 2	D14	53	PIB_IO 45	T4
4	GND	---	54	PIB_IO 46	T2
5	PIB_IO 3	E14	55	PIB_IO 47	T1
6	PIB_IO 4	A13	56	PIB_IO 48	N4
7	PIB_IO 5	B13	57	PIB_IO 49	N3
8	PIB_IO 6	C13	58	PIB_IO 50	N2
9	PIB_IO 7	D13	59	PIB_IO 51	N1
10	PIB_IO 8	E13	60	PIB_IO 52	M6
11	PIB_IO 9	F13	61	PIB_IO 53	M5
12	PIB_IO 10	A12	62	PIB_IO 54	M4
13	PIB_IO 11	B12	63	PIB_IO 55	M3
14	PIB_IO 12	C12	64	PIB_IO 56	M2
15	PIB_IO 13	D12	65	PIB_IO 57	M1
16	PIB_IO 14	E12	66	PIB_IO 58	L1
17	PIB_IO 15	F12	67	PIB_IO 59	L2
18	PIB_IO 16	A11	68	PIB_IO 60	L3
19	PIB_IO 17	B11	69	PIB_IO 61	L4
20	PIB_IO 18	C11	70	PIB_IO 62	L5
21	PIB_IO 19	D11	71	PIB_IO 63	L6
22	PIB_IO 20	E11	72	PIB_IO 64	K1
23	GND	---	73	PIB_IO 65	K2
24	PIB_IO 21	F11	74	PIB_IO 66	K3
25	PIB_IO 22	A10	75	PIB_IO 67	K4
26	PIB_IO 23	B10	76	PIB_IO 68	H5
27	PIB_IO 24	C10	77	PIB_IO 69	G1
28	PIB_IO 25	D10	78	PIB_IO 70	G2
29	PIB_IO 26	E10	79	PIB_IO 71	G5
30	PIB_IO 27	F10	80	PIB_IO 72	G6
31	PIB_IO 28	A9	81	PIB_IO 73	F2
32	PIB_IO 29	B9	82	PIB_IO 74	F3
33	PIB_IO 30	D9	83	PIB_IO 75	F4
34	PIB_IO 31	E9	84	PIB_IO 76	F5
35	PIB_IO 32	F9	85	PIB_IO 77	F6
36	PIB_IO 33	A8	86	PIB_IO 78	E1
37	PIB_IO 34	B8	87	PIB_IO 79	E2
38	PIB_IO 35	C7	88	PIB_IO 80	E3
39	PIB_IO 36	D7	89	PIB_IO 81	E4
40	PIB_IO 37	E7	90	PIB_IO 82	E6

41	PIB_IO 38	F7	91	PIB_IO 83	D1
42	PIB_IO 39	A5	92	PIB_IO 84	D2
43	PIB_IO 40	A3	93	PIB_IO 85	D3
44	PIBCLK(50MHz)	---	94	PIB_IO 86	D4
45	GND	---	95	PIB_IO 87	D5
46	PIB_IO 41	B5	96	PIB_IO 88	D6
47	PIB_IO 42	B6	97	PIB_IO 89	C1
48	+3.3 V	---	98	PIB_IO 90	C2
49	+3.3 V	---	99	PIB_IO 91	C5
50	+3.3 V	---	100	PIB_IO 92	C6

Table 3 CON7 (Connection between FPGA and PIB)

PIN	HD-Sub	Comment	PIN	HD-Sub	Comment
1	GND		51	+12 Volt	---
2	GND		52	+12 Volt	---
3	GND		53	+12 Volt	---
4	HD-SUB PIN 39	Pair 0	54	+5 Volt	---
5	HD-SUB PIN 20	Pair 0	55	+5 Volt	---
6	HD-SUB PIN 38	Pair 1	56	+5 Volt	---
7	HD-SUB PIN 19	Pair 1	57	GND	---
8	HD-SUB PIN 37	Pair 2	58	GND	---
9	HD-SUB PIN 18	Pair 2	59	GND	---
10	HD-SUB PIN 36	Pair 3	60	HD-SUB PIN 41	---
11	HD-SUB PIN 17	Pair 3	61	HD-SUB PIN 42	---
12	HD-SUB PIN 35	Pair 4	62	HD-SUB PIN 43	---
13	HD-SUB PIN 16	Pair 4	63	HD-SUB PIN 44	---
14	HD-SUB PIN 34	Pair 5	64	HD-SUB PIN 45	---
15	HD-SUB PIN 15	Pair 5	65	HD-SUB PIN 46	---
16	HD-SUB PIN 33	Pair 6	66	HD-SUB PIN 47	---
17	HD-SUB PIN 14	Pair 6	67	HD-SUB PIN 48	---
18	HD-SUB PIN 34	Pair 7	68	HD-SUB PIN 49	---
19	HD-SUB PIN 13	Pair 7	69	HD-SUB PIN 39	---
20	HD-SUB PIN 12	---	70	HD-SUB PIN 60	---
21	HD-SUB PIN 31	---	71	HD-SUB PIN 61	---
22	HD-SUB PIN 11	---	72	HD-SUB PIN 62	---
23	HD-SUB PIN 30	---	73	HD-SUB PIN 63	---
24	HD-SUB PIN 10	---	74	HD-SUB PIN 64	---
25	HD-SUB PIN 9	---	75	HD-SUB PIN 65	---
26	HD-SUB PIN 8	---	76	HD-SUB PIN 66	---
27	HD-SUB PIN 7	---	77	HD-SUB PIN 67	---
28	HD-SUB PIN 6	---	78	HD-SUB PIN 68	---
29	HD-SUB PIN 5	---	79	HD-SUB PIN 69	---
30	HD-SUB PIN 4	---	80	HD-SUB PIN 50	---
31	HD-SUB PIN 3	---	81	HD-SUB PIN 70	---
32	HD-SUB PIN 2	---	82	HD-SUB PIN 51	Pair 15

33	HD-SUB PIN 29	---	83	HD-SUB PIN 71	Pair 15
34	HD-SUB PIN 28	---	84	HD-SUB PIN 52	Pair 14
35	HD-SUB PIN 27	---	85	HD-SUB PIN 72	Pair 14
36	HD-SUB PIN 26	---	86	HD-SUB PIN 53	Pair 13
37	HD-SUB PIN 25	---	87	HD-SUB PIN 73	Pair 13
38	HD-SUB PIN 24	---	88	HD-SUB PIN 54	Pair 12
39	HD-SUB PIN 23	---	89	HD-SUB PIN 74	Pair 12
40	HD-SUB PIN 22	---	90	HD-SUB PIN 55	Pair 11
41	HD-SUB PIN 21	---	91	HD-SUB PIN 56	Pair 11
42	GND	---	92	HD-SUB PIN 76	Pair 10
43	GND	---	93	HD-SUB PIN 57	Pair 10
44	GND	---	94	HD-SUB PIN 77	Pair 9
45	+5 Volt	---	95	HD-SUB PIN 58	Pair 9
46	+5 Volt	---	96	HD-SUB PIN 78	Pair 8
47	+5 Volt	---	97	HD-SUB PIN 59	Pair 8
48	+12 Volt	---	98	GND	---
49	+12 Volt	---	99	GND	---
50	+12 Volt	---	100	GND	---

Table 4 CON8 (Connection between HD-SUB Connector CON9 and PIB)

2.5 Plug-In-Board Hardware

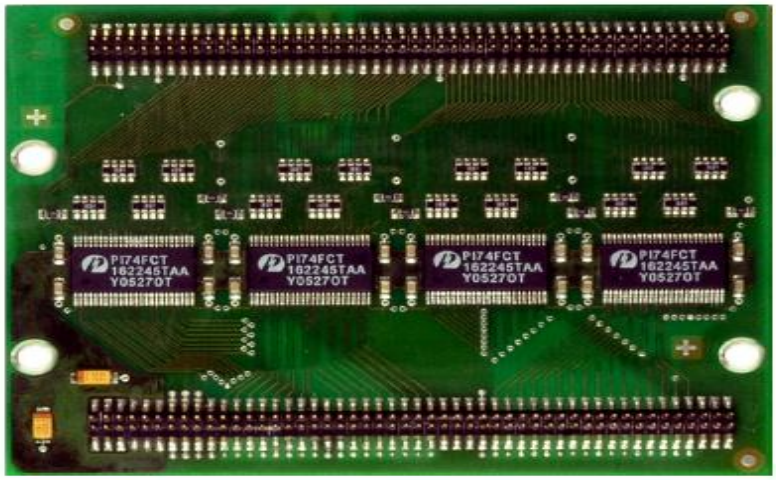


Figure 2 Plug In Board

The PIB64IO functions as daughterboard to the CESYS base series cards. It provides 64 TTL compatible IOs, organized in 8 banks. Each bank can be switched between input and output mode. Each bank has its own enable signal. Each of the 4 ICs on the PIB implements 2 of these banks. In both sides of the PIB there are installed two 100-pin connectors (CON7 + CON8) which connect the PIB with the FPGA balls and external 78-pin HD-Sub respectively. It is already described in 2.4 analytically every single connection of these connectors.

All in all the PIB consists of the following hardware features:

- 8 banks x 8 IOs (5 Volt TTL compatible)
- Each bank configurable as Input or Output
- Individual output-enable for each bank
- Typical output scew per bank < 250 ps
- ESD > 2000V per MIL-STD-883, Method 3015
- ESD > 200V using machine model (C=200pF, R=0)
- Balanced output drivers ± 24mA
- Uses quadruple of 74FCT162245 Fast CMOS bidirectional transceiver

Regarding the information flow from the FPGA balls to the output pins of the external 78-pin HD-Sub, it is written down on the table below which FPGA ball speaks to which PIB port and to which pin of 78-pin HD-Sub.

Signal name	FPGA I/O ball#	HD SubD Pin#	Signal name	FPGA I/O ball#	HD SubD Pin#
P0.0	F9	10	P4.0	A11	39
P0.1	L1	9	P4.1	F4	38
P0.2	A8	8	P4.2	B11	37
P0.3	M1	7	P4.3	F3	36
P0.4	B8	6	P4.4	C11	35
P0.5	M2	5	P4.5	F2	34
P0.6	C7	4	P4.6	D11	33
P0.7	M3	3	P4.7	G6	32
DIR P0	E9	--	DIR P4	G5	--
#OE P0	L2	--	#OE P4	E11	--
P1.0	D7	29	P5.0	C12	20
P1.1	M4	28	P5.1	E2	19
P1.2	E7	27	P5.2	D12	18
P1.3	M5	26	P5.3	E1	17
P1.4	F7	25	P5.4	E12	16
P1.5	M6	24	P5.5	F6	15
P1.6	A5	23	P5.6	F12	14
P1.7	N1	22	P5.7	F5	13
DIR P1	N2	--	DIR P5	B12	--
#OE P1	A3	--	#OE P5	E3	--
P2.0	E10	49	P6.0	C13	59
P2.1	K1	48	P6.1	D3	58
P2.2	F10	47	P6.2	D13	57
P2.3	L6	46	P6.3	D2	56
P2.4	A9	45	P6.4	E13	55
P2.5	L5	44	P6.5	D1	54
P2.6	B9	43	P6.6	F13	53
P2.7	L4	42	P6.7	E6	52
DIR P2	L3	--	DIR P6	E4	--
#OE P2	D9	--	#OE P6	A12	--

P3.0	A10	68	P7.0	D14	78
P3.1	H5	67	P7.1	C2	77
P3.2	B10	66	P7.2	E14	76
P3.3	K4	65	P7.3	D6	75
P3.4	C10	64	P7.4	A13	74
P3.5	K3	63	P7.5	D5	73
P3.6	D10	62	P7.6	B13	72
P3.7	K2	61	P7.7	D4	71
DIR P3	F11	--	DIR P7	B14	--
#OE P3	G1		#OE P7	C5	--

Table 5 Pinout PIB64IO on PCIS3BASE

Having already an overview of the connections between the FPGA and the PIB ports, it is time to present a figure which describes the dataflow for the PIN F9 of the FPGA which corresponds to the bit 0 of port 0 (signal name =P0.0).

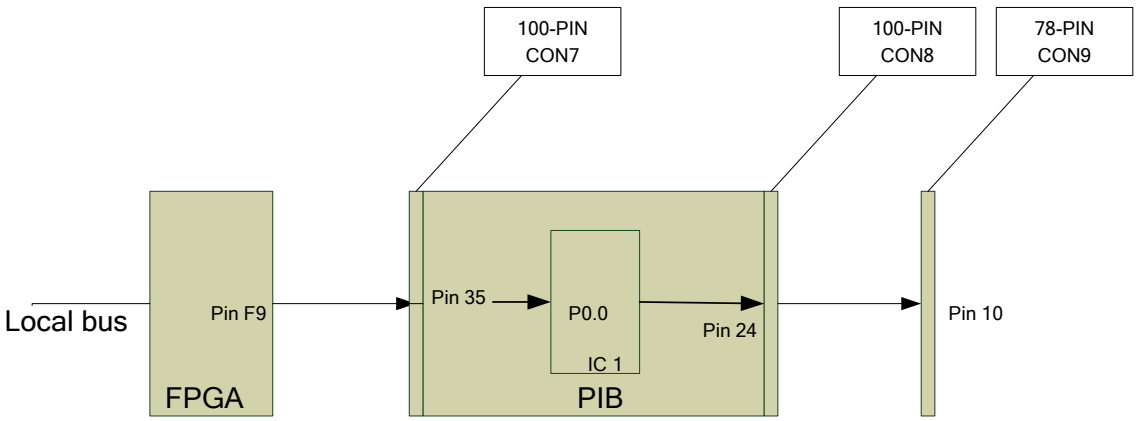


Figure 3 Dataflow from FPGA to 78-pin HD-Sub for P0.0 signal

Signal P0.0 (bit0 of port0) speaks to PIN F9 of the FPGA through PIN 35 of CON7 and simultaneously it also speaks to PIN10 of HD-SUB through PIN 24 of CON8. The connections of signal P0.0 can be verified from Table 5 written just above. What is more, looking on tables of CON7 and CON8 (Table 3, 4) the other 2 connections can be verified as well. It should be underlined that Figure 2 only gives a plain description of the hardware inside the PIB. For more information you can refer to the documentation of PIB under the name **CE035.pdf.**[4]

[4] http://www.cesys.com/resources/C1050-3506_PIB64IO_UserManual.pdf

2.6 Internal Expansion port J21

The Internal expansion port J21 is of type male and the pins are directly connected to the FPGA I/O Balls. These pins are not 5V tolerant. Through J21 28 FPGA I/O are accessible, 24 of which are routed as 12 pairs to support differential signalling. 3.3 Volt power supply is also available, so it is even possible to power active devices on boards connected to J21. Current supplied over J21 should not exceed 100mA.

PIN	FPGA I/O ball	Comment	PIN	FPGA I/O ball	Comment
1	E21	Bank2_IO21N	18	D18	Bank1_IO09P
2	E21	Bank2_IO21P	19	GND	---
3	D21	Bank2_IO17N	20	GND	---
4	D22	Bank2_IO17P	21	A18	Bank1_IO10N
5	C22	---	22	B18	Bank1_IO10P
6	F17	---	23	D17	Bank1_IO15N
7	E19	Bank2_IO20N	24	E17	Bank1_IO15P
8	E20	Bank2_IO20P	25	E16	---
9	GND	---	26	F16	---
10	GND	---	27	B17	Bank1_IO16P
11	D19	Bank2_IO16P	28	C17	Bank1_IO16N
12	D20	Bank2_IO16N	29	3.3 Volt	---
13	E18	Bank2_IO19N	30	3.3 Volt	--
14	F18	Bank2_IO19P	31	D15	Bank1_IO24N
15	A19	Bank1_IO06N	32	E15	Bank1_IO24P
16	B19	Bank1_IO09P	33	A15	Bank1_IO25N
17	C18	Bank1_IO09N	34	B15	Bank1_IO25P

Table 6 J21 Internal Expansion Connector-FPGA Connections

2.7 Local Bus Signals

In this section, there is a short description of the interface between Spartan-3 FPGA and PLX PCI9056. In general, PCI9056 supports three types of local bus processor interfaces. However for PCIS3BASE only J mode with multiplexed address/data bus is available. From the three existing data transfer modes of PCI9056, direct slave mode and DMA mode are implemented. For data transmission, 32-bit single read/wrote and DMA single and continuous burst cycles are supported.

The following spreadsheet gives an overview of the local bus signals and to which FPGA I/O balls they are connected.

FPGA I/O	I/O Standard	Signal Name	External pull up/down	Comment
W4	LVCMOS33	ADS#	pull –up	Address strobe
Y18	LVCMOS33	ALE	pull –down	Address latch enable

W5	LVCMOS33	BIGEND#	pull –up	Big endian select
W1	LVCMOS33	BLAST#	pull –up	Burst last
W2	LVCMOS33	BREQi	pull –down	Bus request in
AA6	LVCMOS33	BREQo	pull –up	Bus request out
U10	LVCMOS33	BTERM#	pull –up	Burst terminate
U6	LVCMOS33	CCS#	pull –up	Configuration register select
W6	LVCMOS33	DACK0#	---	DMA channel 0 demand mode acknowledge
U7	LVCMOS33	DACK1#	---	DMA channel 1demand mode acknowledge
W18	LVCMOS33	DEN#	pull –up	Data enable
W9	LVCMOS33	DP0	pull –down	Data parity 0
Y1	LVCMOS33	DP1	pull –down	Data parity 1
AA8	LVCMOS33	DP2	pull –down	Data parity 2
V9	LVCMOS33	DP3	pull –down	Data parity 3
Y5	LVCMOS33	DREQ0#	pull –up	DMA channel 0 demand mode request
V7	LVCMOS33	DREQ1#	pull –up	DMA channel 1demand mode request
V18	LVCMOS33	DT/R#	pull –up	Data transmit/receive
AA4	LVCMOS33	EOT#	pull –up	End of transfer for current DMA channel
AA14	LVCMOS33	LAD 0	pull –up	Multiplexed data address bus
AB14	LVCMOS33	LAD 1	pull –up	Multiplexed data address bus
U12	LVCMOS33	LAD 2	pull –up	Multiplexed data address bus
V12	LVCMOS33	LAD 3	pull –up	Multiplexed data address bus
W11	LVCMOS33	LAD 4	pull –up	Multiplexed data address bus
V11	LVCMOS33	LAD 5	pull –up	Multiplexed data address bus
AB9	LVCMOS33	LAD 6	pull –up	Multiplexed data address bus
AA9	LVCMOS33	LAD 7	pull –up	Multiplexed data address bus
Y10	LVCMOS33	LAD 8	pull –up	Multiplexed data address bus
V10	LVCMOS33	LAD 9	pull –up	Multiplexed data address bus
W10	LVCMOS33	LAD 10	pull –up	Multiplexed data address bus

AA10	LVCMOS33	LAD 11	pull –up	Multiplexed data address bus
V13	LVCMOS33	LAD 12	pull –up	Multiplexed data address bus
Y13	LVCMOS33	LAD 13	pull –up	Multiplexed data address bus
W13	LVCMOS33	LAD 14	pull –up	Multiplexed data address bus
AA13	LVCMOS33	LAD 15	pull –up	Multiplexed data address bus
U11	LVCMOS33	LAD 16	pull –up	Multiplexed data address bus
AB10	LVCMOS33	LAD 17	pull –up	Multiplexed data address bus
AB11	LVCMOS33	LAD 18	pull –up	Multiplexed data address bus
U13	LVCMOS33	LAD 19	pull –up	Multiplexed data address bus
AB15	LVCMOS33	LAD 20	pull –up	Multiplexed data address bus
AA15	LVCMOS33	LAD 21	pull –up	Multiplexed data address bus
W16	LVCMOS33	LAD 22	pull –up	Multiplexed data address bus
Y16	LVCMOS33	LAD 23	pull –up	Multiplexed data address bus
AB13	LVCMOS33	LAD 24	pull –up	Multiplexed data address bus
V14	LVCMOS33	LAD 25	pull –up	Multiplexed data address bus
W14	LVCMOS33	LAD 26	pull –up	Multiplexed data address bus
U14	LVCMOS33	LAD 27	pull –up	Multiplexed data address bus
V16	LVCMOS33	LAD 28	pull –up	Multiplexed data address bus
U16	LVCMOS33	LAD 29	pull –up	Multiplexed data address bus
U17	LVCMOS33	LAD 30	pull –up	Multiplexed data address bus
AA17	LVCMOS33	LAD 31	pull –up	Multiplexed data address bus
V17	LVCMOS33	LBE0#	pull –up	Local byte enable 0
AA18	LVCMOS33	LBE1#	pull –up	Local byte enable 1
Y17	LVCMOS33	LBE2#	pull –up	Local byte enable 2
AB18	LVCMOS33	LBE3#	pull –up	Local byte enable 3
--	LVCMOS33	LCLK	pull –up	Local processor clock(66MHz)
AB4	LVCMOS33	LHOLD	pull –down	Local hold request

W3	LVCMOS33	LHOLDA	pull –down	Local hold acknowledge
V6	LVCMOS33	LINTi#	pull –up	Local interrupt input
Y6	LVCMOS33	LINTo#	pull –up	Local interrupt output
V5	LVCMOS33	LRESET#	pull –up	Local bus reset
W8	LVCMOS33	LSERR#	pull –up	Local system error interrupt output
W17	LVCMOS33	LW/R#	pull –up	Local write/read
AB8	LVCMOS33	READY#	pull –up	Ready I/O
V8	LVCMOS33	WAIT#	pull –up	Wait I/O

Table 7 Local Bus-FPGA Connections

Detailed description of some signals:

ADS#

Indicates a valid address and start of a new Bus access. ADS# asserts for the first clock of the Bus access.

LCLK

Local clock input. Sourced by onboard 50MHz oscillator.

LHOLD

Asserted to request use of the Local Bus.

LHOLDA

The external Local Bus Arbiter asserts LHOLDA when bus ownership is granted in response to LHOLD. The Local Bus should not be granted to PCI 9056, unless requested by LHOLD.

LINTo#

Synchronous output that remains asserted as long as the interrupt is enabled and the interrupt condition exists.

LW/R#

Asserted low for reads and high for writes.

READY#

A Local slave asserts READY# to indicate that Read data on the bus is valid or that a Write Data transfer is complete. READY# input is not sampled until the internal Wait State Counter(s) expires (WAIT# output de-asserted).

2.8 JTAG Interface

In addition to configuration via PCI, it is possible to download configuration data using a JTAG interface. The *PCIS3BASE* is equipped as standard with a 2- row 14- pin connector to plug in the *Parallel Cable IV* from XilinxTM. The JTAG interface is not only suitable to download designs for testing purposes but enables the user to check a running design by the help of software tools provided by XilinxTM.

Pin	Comment
1,3,5,7,9,11,13	GND
2	+ 2.5V
4	TMS
6	TCK
8	TDO
10	TDI
12,14	Not Connected

Table 8 CON1-JTAG Connector

2.9 Memory Interface

The PCIS3BASE card is equipped with 32MByte of dynamic high speed RAM.
The following spreadsheet gives an overview of the signals' names and which FPGA I/O balls they are connected to.

Signal Name	FPGA I/O BALL	Comment
A0	K20	Multiplexed row/column address input
A1	G22	Multiplexed row/column address input
A2	G19	Multiplexed row/column address input
A3	G17	Multiplexed row/column address input
A4	G18	Multiplexed row/column address input
A5	G21	Multiplexed row/column address input
A6	K19	Multiplexed row/column address input
A7	K21	Multiplexed row/column address input
A8	L17	Multiplexed row/column address input
A9	L19	Multiplexed row/column address input
A10	K22	Multiplexed row/column address input
A11	L21	Multiplexed row/column address input
A12	L22	Multiplexed row/column address input
BA0	L20	Bank Address Input
BA1	L18	Bank Address Input

DQ0	Y21	Data input/output
DQ1	W21	Data input/output
DQ2	W19	Data input/output
DQ3	V21	Data input/output
DQ4	V19	Data input/output
DQ5	U20	Data input/output
DQ6	U18	Data input/output
DQ7	T21	Data input/output
DQ8	T22	Data input/output
DQ9	U19	Data input/output
DQ10	U21	Data input/output
DQ11	V20	Data input/output
DQ12	V22	Data input/output
DQ13	W20	Data input/output
DQ14	W22	Data input/output
DQ15	Y22	Data input/output
CS#	N19	Chip Select input
WE#	R18	Write Enable
CAS#	N22	Column Address Strobe
RAS#	N20	Row Address Strobe
CKE#	N21	Clock Enable Input
Clock	Y11	SDRAM CLK Input
DQMH	T17	Input/Output Data Mask
DQML	T18	Input/Output Data Mask

Table 9 SDRAM-FPGA Connections

2.10 SPI Flash

In addition to 32MByte dynamic SDRAM, 4MBit nonvolatile memory in form of a SPI Flash from STMicroelectronis is available. This flash memory is not intended for storing FPGA configuration bitstreams (no connection to FPGA configuration logic is available) but to give the user the opportunity to store board specific data directly onboard.

The following table gives information about IO usage:

Signal Name	FPGA I/O Ball	Comment
FLASH_#CS	F20	Chip Select
FLASH_SO	F19	Serial Data Output
FLASH_SI	M22	Serial Data Input
FLASH_SCK	F21	Serial Clock
FLASH_#HOLD	---	Active-Low Hold Signal
FLASH_#WP	---	Active-Low Write Protect Signal

Table 10 SPI FLASH-FPGA Connections

3. Wishbone bus architecture

3.1 General Information about Wishbone bus

The WISHBONE bus is an open source hardware computer bus allowing the connection of different cores to each other inside of a chip.

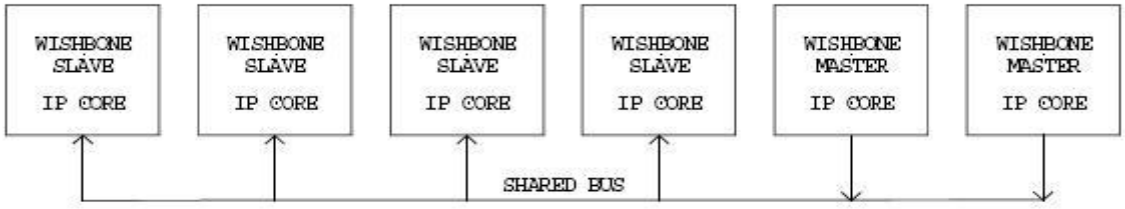
WISHBONE is intended to be a “logic bus”, which instead of specifying electrical information or bus topology, the specification is written in terms of signals, clock cycles and high/low levels.

WISHBONE is defined to have 8, 16, 32 and 64-bit buses. All signals are synchronous to a single clock but some slave responses must be generated asynchronously for maximum performance.

3.1.1 Wishbone Topologies

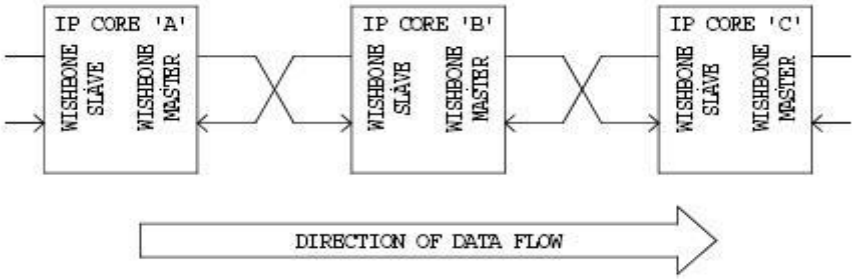
WISHBONE is really flexible and adapts well to common topologies such as point-to-point, many-to-many, hierarchical, or even switched fabrics such as crossbar switches. Normally WISHBONE requires a bus controller or arbiter, but devices still maintain the same interface. We can see below some topologies in which WISHBONE can be adopted.[5]

Shared bus

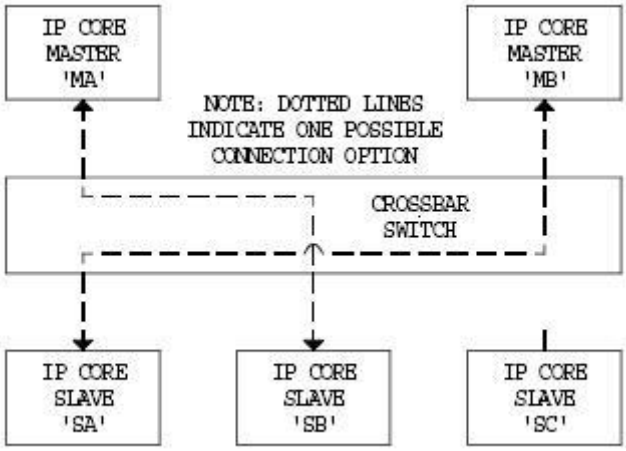


[5] http://en.wikipedia.org/wiki/Wishbone_(computer_bus)

Pipeline



Cross bar switch



3.2 Wishbone implementation on pcis3base

A 32-bit WISHBONE based shared bus architecture is implemented on the pcis3base. All devices of the WISHBONE system support only single READ/WRITE cycles. Files and modules having something to do with the WISHBONE system are labelled with the prefix “wb_”. WISHBONE master is labelled additionally with the prefix “ma_” and the slaves with the prefix “sl_”.

3.2.1 Wishbone topology used on pcis3base

First of all, an image of this architecture will make clear how the WISHBONE architecture is built.

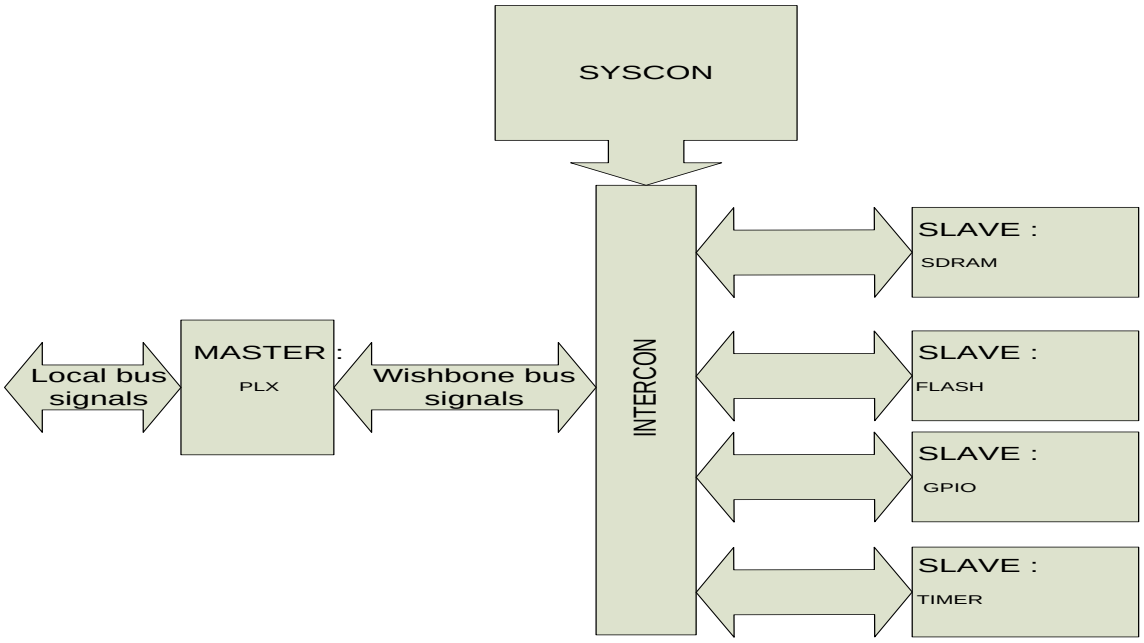


Figure 4 WISHBONE Architecture

A thorough description of each .vhd file which is a member of this WISHBONE based shared bus architecture will follow and will clarify how WISHBONE architecture works.

3.2.2 Basic wishbone files and modules

src/wishbone.vhd

A package containing data types, constants, components, signals and information needed for the WISHBONE system.

src/pcis3base_top.vhd

The top module of our design. WISHBONE components are instantiated here and internal VHDL signals are mapped to the 100 pin connector of the PIB.

src/wb_syscon.vhd

It provides two basic WISHBONE system signals: rst and clk.

src/wb_intercon.vhd

Through this module it is possible to exist a communication between the modules through intercon signal. All WISHBONE devices are connected to this shared bus interconnection logic. Some MSBs of the address ($\log_2$(number_of_slave_Devices)) define which slave device is selected each time.

src/wb_ma_plx.vhd

There is only one master device in the WISHBONE architecture which is responsible for generating the WISHBONE signals having as input the signals from local bus.

src/wb_sl_sdr.vhd

This entity represents the low level SDRAM controller sdr_ctrl.vhd for the 32MB/16-bit SDRAM.

src/wb_sl_flash.vhd

This entity represents the low level FLASH controller flash_ctrl.vhd for the 4Mbit SPI FLASH memory.

src/wb_sl_gpio.vhd

This is the basic slave device which is most alike to the new slave device (wb_sl_mss.vhd). It is responsible for communicating with the 8 PIB ports. That means that through this module the direction of the PIB ports (Input or Output) is defined and what is more, due to this module information packages can be sent or received through the PIB ports. In other words, in this module the dual 8-bit bus transceiver circuits are controlled. The 8 LEDs and the 28 bidirectional I/Os at the internal 34-pin connector are controlled by this module as well.

src/wb_sl_timer.vhd

This entity represents a 32-bit timer with programmable period (20 ns), which generates an interrupt at overflow time.

3.2.3 Local bus signals

The unique master device (wb_ma_plx) uses the local bus signals in order to arbitrate the generation of the WISHBONE related signals. As a result, it is obvious that the local bus signals are the signals which judge whether information should be received or should be sent from and to PIB transceiver circuits. A deep understanding of the local bus signals' values in both reading and writing bus transactions will make it easier afterwards to explain the values of the WISHBONE related signals.

These are the local bus signals:

LW/R#
It defines if a write (LW/R# =1) or read (LW/R# =0) cycle is in progress.

ADS#
It defines if the address is valid (if asserted low) or not.

READY#
Indicates a successful data transfer for writing and valid data on bus for reading by asserting this signal low.

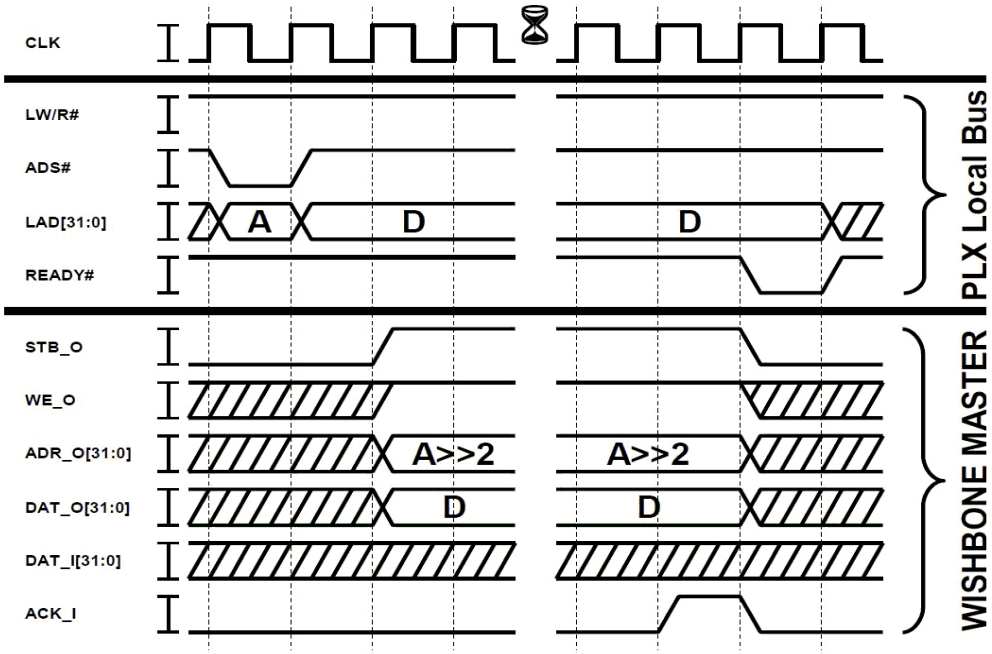
LAD[31:0]
It is the multiplexed (Address/Data) piece of information which is sent or received.

LHOLD
It is driven by plx and is used for local bus arbitration.

LHOLDA
It is also used for local bus arbitration but it is driven by the FPGA.

From the following two images, it can be easier understood not only which values the local bus signals have in a write and a read cycle, but also the relationship between local bus and WISHBONE signals' values. WISHBONE related signals will be described in the following subchapter.

Write Cycle:



Read Cycle:

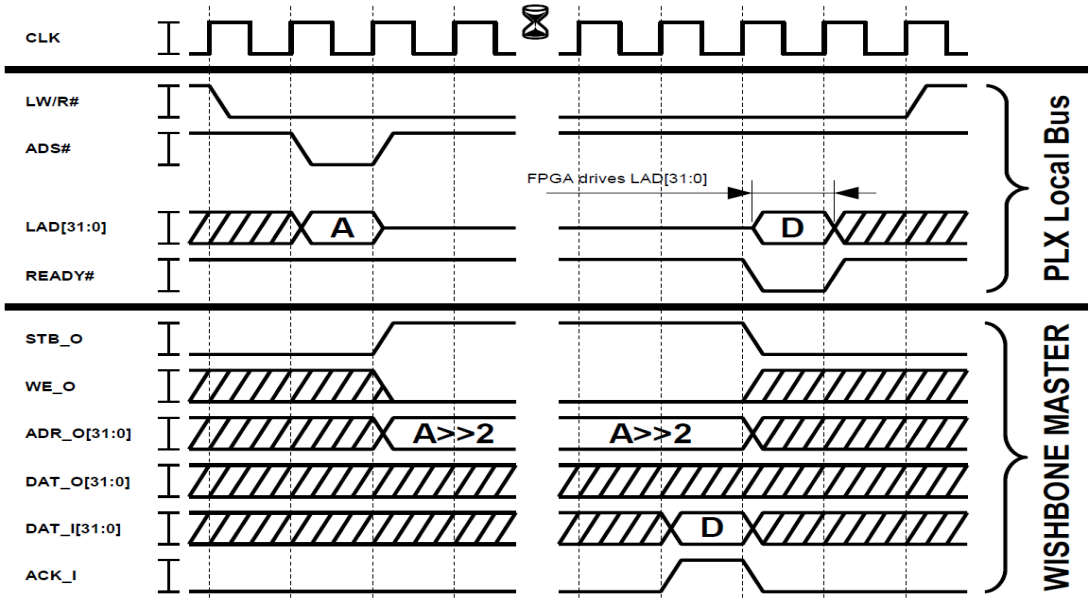


Figure 5 Local Bus transactions for write and read cycle

3.2.4 Wishbone signals

WISHBONE signals can be divided in two main categories:

- Signals from master devices to slaves :
  - ✓ STB_O : Indicates valid data and control signals.
  - ✓ WE_O : Indicates a write or read cycle.
  - ✓ CYC_O : Defines which slave device will be active (CYC_O='1').
  - ✓ ADR_O : Address where the Data package will be written.
  - ✓ DAT_O : 32-bit data out bus for data transportation from master to slaves.

- Signals from slave devices to master (plx device) :
  - ✓ DAT_I: 32-bit data in bus for data transportation from slaves to master.
  - ✓ ACK_I: Handshake signal, which indicates a successful data transfer for writing and valid data on bus for reading.

3.2.5 Wishbone signal structure

As already showed above in both illustrations, the WISHBONE signals are written as simple bit types or bit vector types, but in the VHDL code these signals could be encapsulated in extended data types like arrays or records. As WISHBONE related signals will be regularly referred throughout this text, it is worth writing down the WISHBONE data type structure.

```
type rec_syscon_port is record
    rst : std_logic;
    clk : std_logic;
  end record;

type rec_master_port is record
    cyc : std_logic;
    stb : std_logic;
    we : std_logic;
    adr : std_logic_vector(rng_adr);
    dat : std_logic_vector(rng_dat);
  end record;

type rec_slave_port is record
    ack : std_logic;
    dat : std_logic_vector(rng_dat);
  end record;

type rec_wishbone_signal is record
    syscon : rec_syscon_port;
    master : rec_master_port;
    slave : rec_slave_port;
  end record;

rec_wishbone_signal wishbone;
```

For better understanding of the data type definition refer to the figure below:

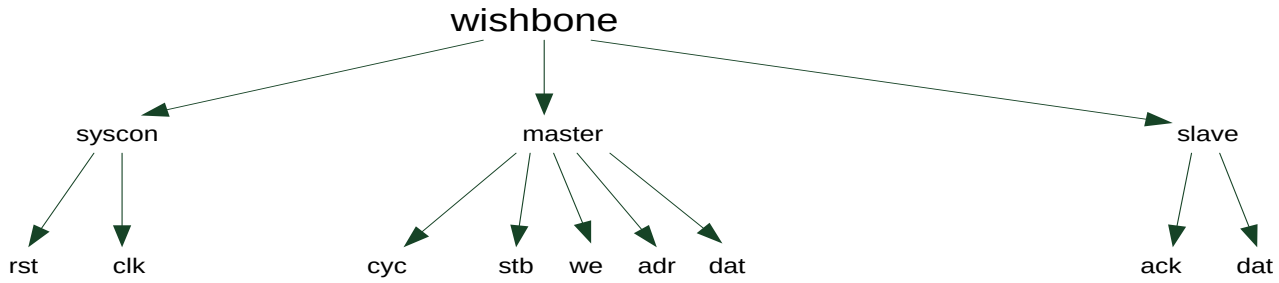


Figure 6 WISHBONE Datatype

3.2.6 Relationship between wishbone and local bus signals

In order to make it easier for the reader to understand how the WISHBONE signals are generated, the whole process will be displayed below in which the local bus signals are translated to WISHBONE bus signals and the opposite.

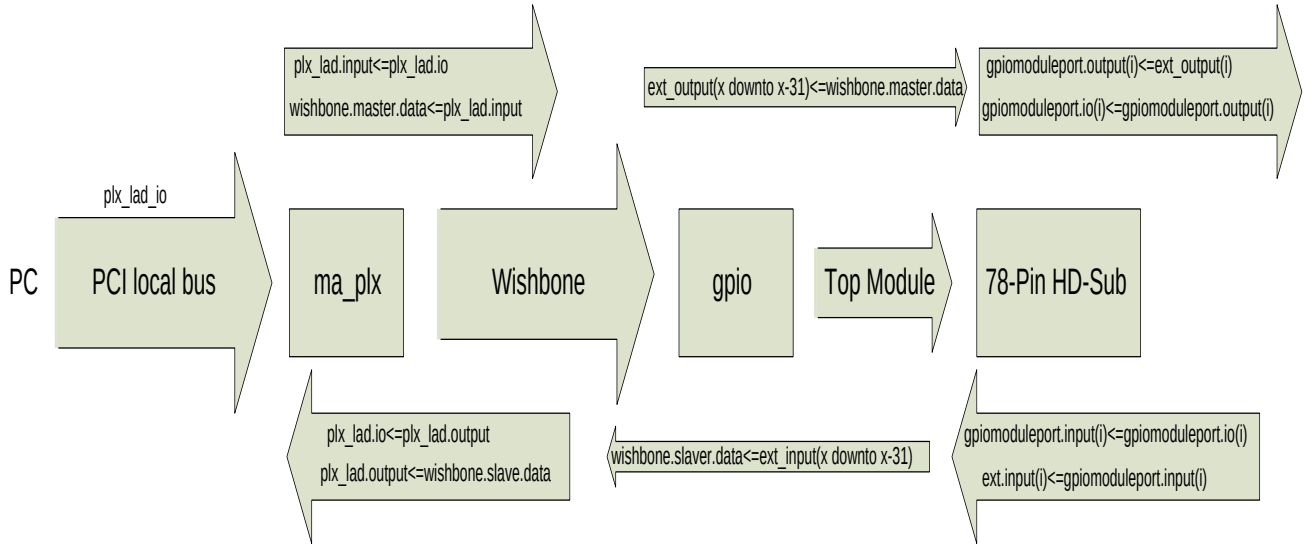


Figure 7 Dataflow process between PC and PCB

3.2.6.1 Dataflow from PC to PCB

In case a data package is sent to the PCB which will be connected to PC through the 78-Pin-HD-Sub, the data package will be embedded in the "plx_lad_io" signal, which is an input/output type (i/o type) of signal. Obviously here the "plx_lad_io" is used as an input signal. Thus, firstly its info is passed to the "plx_lad.input" help signal and then the data package is transmitted to WISHBONE bus through "wishbone.master.data" signal. Depending on which pins of the 78-Pin-HD-Sub the data package is sent through, the package will be further transmitted either to "ext_output(31 downto 0)" signal or "ext_output(63 downto 32)" signal. Unfortunately, it is not possible to send data simultaneously to all ports as the implemented WISHBONE bus has 32 bit width, in other words the signal "wishbone.master.data" is a std_logic_vector(31 downto 0). Therefore, as every port has 8-bits width the user is able to send in one clock cycle a data package either in the first four ports or in the last four ports of the PIB. For the first four ones is responsible the "ext_output(31 downto 0)" signal and for the last four the "ext_output(63 downto 32)". As the data package information has been passed to "ext_output", the final step of the dataflow process is ready to begin. The "gpiomoduleport.io" signal as its name implies, is a bidirectional 92-bit signal (i/o type) which sends or receives data from the PIB ports through CON7 external expansion connector. In this case this signal is used as an output signal in order to send the data package to the PIB ports.

3.2.6.2 Dataflow from PCB to PC

Describing the dataflow in the opposite direction and beginning from the 78-Pin-HD-Sub from which the PCIS3BASE card receives a data package, the information received flows through some of the PIB ports and then its context is embedded in the "gpiomoduleport.io" signal, which in this case is an input signal. As in the previous description of the dataflow towards PCB there was the "ext_output()" signal, here another signal is responsible for carrying the information. This signal is the "ext_input()" which as soon as receives the data package info from "gpiomoduleport.io", it is ready to pass on the information in the WISHBONE bus through "wishbone.slave.data" signal. In the last step of the dataflow

process towards PC, in ma_plx module the data package is embedded in the local bus as the information is carried by “plx_lad_io” which is an output signal in this occasion.

3.2.7 Basic local bus control signals used from wishbone and their operation

In the bus transactions of Read and Write Cycle, there are several local bus signals used, which made possible the generation of WISHBONE bus signals, which were shown in the previous subchapter. Here, the basic points are referred inside the code where this job is done.

First of all, starting from the master device connected to the WISHBONE bus, **plx_ads_n** signal is a control signal which is responsible for passing the 32-bit data package, carried by plx_lad.io signal, either on **wishbone.master.adr** signal or on **wishbone.master.dat** signal. More specifically, if the **plx_ads_i** is '0 'then the content of the local bus address is translated as the WISHBONE address for the WISHBONE bus (wishbone.master.adr <= plx_lad_io) and accordingly if the signal **plx_ads_i** is '1' then the content corresponds to the data package of the WISHBONE bus (wishbone.master.dat <= plx_lad_io). The above function is evident in the following code segment of file “wb_ma_plx.vhd”:

```
if plx_ads_n(0) = '0' then
      wishbone.master.adr <= b"00" &  plx_lad.input(31 downto 2);
else
      wishbone.master.dat <= plx_lad.input;
end if;
```

Moreover, as already showed in section 3.2.5 the master device has various control signals (cyc, we, stb) whose values are in turn based on values of the respective local bus signals. Starting with the signal **wishbone.master.we,** it receives the value of signal **plx_lw_r_i** of the local bus.

```
wishbone.master.we <= plx_lw_r_n;
```

Additionally, the value of signal **wishbone.master.stb** depends also on the value of signal **plx_ads_n** of the local bus.

```
if wishbone.master.we = '1' then
      if plx_ads_n(1) = '0' then
        wishbone.master.stb <= '1';
      elsif wishbone.slave.ack = '1' then
        wishbone.master.stb <= '0';
      end if;
else
      if plx_ads_n(0) = '0' then
        wishbone.master.stb <= '1';
      elsif wishbone.slave.ack = '1' then
        wishbone.master.stb <= '0';
      end if;
end if;
```

It is worth stressing out that the value of **wishbone.master.cyc** signal is always '1' as this signal indicates the particular device selected for use in the WISHBONE bus and since there is only one master device, it must be permanently in use.

```
wishbone.master.cyc<= '1';
```

Regarding the values of the slave devices' control signals **(ack)**, the signal **wishbone.slave.ack** has a specific value based on the values of signals **wishbone.master.cyc** and **wishbone.master.stb** as the following segment of code shows (wb_sl_gpio.vhd):

```
if wishbone.slave.ack = '1' then
      wishbone.slave.ack <= '0';
elsif (wishbone.master.cyc and wishbone.master.stb) = '1' then
      wishbone.slave.ack <= '1';
end if;
```

Finally, it should be noted that there are signals on the local bus whose value depends on the corresponding value of signals which belong to the WISHBONE bus. Such signals are for instance **plx_lad.oe, plx_ready_n** and **plx_lad.output.** The following code snippet comes from the source file of the unique master device connected to the WISHBONE bus.

```
plx_lad.oe <= wishbone.slave.ack and (not wishbone.master.we);
plx_ready_n <= not wishbone.slave.ack;
plx_lad.output <= wishbone.slave.dat;
```

3.3 Vhdl implementation of Wishbone Bus Architecture

In this subchapter the basic points of the VHDL implementation of WISHBONE bus architecture will be analyzed. However, the whole structure of this architecture will not be presented as it is represented in vhdl, but only the modules on which the further extension of the architecture will be based with final goal the implementation of the data bus of the microprocessor.

The main points of each module will be highlighted and reading the comments inside the code will make it easier to get a closer view of how each component of WISHBONE architecture works. With the prefix MSS it is made clear in which parts of the code there have to be some changes in the values of constants/signals or even it has to be written additional code so that the MSS module can fit properly inside the WISHBONE bus. [6]

3.3.1 Wishbone Module

First of all, the basic data types used throughout the code will be mentioned and their importance will be explained. The file "**wishbone.vhd**" is the one which includes not only all the data types but also the definition of the components of the WISHBONE structure. It is a package containing as well constants, signals and information for software developers needed for the WISHBONE system. What is more, there are C/C++ -style "#define"s with important

[6] Cesys PCIS3BASE PCI Card sourcecode

addresses and values to copy and paste into the software source code after VHDL comments("--").The code of file "**wishbone.vhd**" follows below.

```
library IEEE

use IEEE.STD_LOGIC_1164.all

use   IEEE.NUMERIC_STD.all;

package wishbone is
  -- basic system configuration
  ----------------------------
  -- nr of master devices
  -- MSS:There is only one master device connected to the wishbone bus which is implemented in the "wb_ma_plx" module
  constant nr_of_masters : positive := 1;
  subtype rng_masters is natural range (nr_of_masters-1) downto 0;
  -- nr of slave devices
  -- MSS:There are already 4 slave devices implemented. For the mss module we will another one
  constant nr_of_slaves : positive := 4;
  subtype rng_slaves is natural range (nr_of_slaves-1) downto 0;
  -- nr of address- and data-lines
  -- Wishbone architecture uses 4 Bytes for addressing and 4 Bytes for the data packets
  constant adr_width : positive := 32;
  subtype rng_adr is natural range (adr_width-1) downto 0;
  constant dat_width : positive := 32;
  subtype rng_dat is natural range (dat_width-1) downto 0;

  -- interrupt settings
  ---------------------
  -- nr of modules connected to the interrupt bus
  -- MSS: The mss module will be the 6th device connected to the interrupt bus
  constant nr_of_irqdevs : positive := 5;
  subtype rng_irqdevs is natural range (nr_of_irqdevs-1) downto 0;
  -- nr of global IRQs
  -- MSS: The number of global IRQs will be remain equal to on.
  constant nr_of_irqs : positive := 1;
  subtype rng_irqs is natural range (nr_of_irqs-1) downto 0;
  -- signals building the interrupt bus
  type arr_irqos is array(natural range <>) of std_logic_vector(rng_irqs);
  -- each module connected to the interrupt bus gets an unique interrupt device id
  signal irqos : arr_irqos(rng_irqdevs) := (others => (others => '0'));
  -- each IRQ on the interrupt bus gets an unique interrupt id
  signal irqs : std_logic_vector(rng_irqs) := (others => '0');
  -- WISHBONE specific data types and default values
  --------------------------------------------------
  -- ports driven by WISHBONE syscon modules
  type rec_syscon_port is record
    rst : std_logic;
    clk : std_logic;
  end record;
  constant syscon_default : rec_syscon_port :=
  (
    rst => '0',
    clk => '0'
  );
  type arr_syscon_port is array(natural range <>) of rec_syscon_port;
  -- ports driven by WISHBONE master modules
  type rec_master_port is record
    cyc : std_logic;
    stb : std_logic;
    we : std_logic;
    adr : std_logic_vector(rng_adr);
    dat : std_logic_vector(rng_dat);
  end record;
  constant master_default : rec_master_port :=
  (
```

```
  cyc => '0',
  stb => '0',
  we => '0',
  adr => (others => '0'),
  dat => (others => '0')
);
type arr_master_port is array(natural range <>) of rec_master_port;
-- ports driven by WISHBONE slave modules
type rec_slave_port is record
  ack : std_logic;
  dat : std_logic_vector(rng_dat);
end record;
constant slave_default : rec_slave_port :=
(
  ack => '0',
  dat => (others => '0')
);
type arr_slave_port is array(natural range <>) of rec_slave_port;
-- the whole WISHBONE signal
type rec_wishbone_signal is record
  syscon : rec_syscon_port;
  master : rec_master_port;
  slave : rec_slave_port;
end record;
constant wishbone_default : rec_wishbone_signal :=
(
  syscon => syscon_default,
  master => master_default,
  slave => slave_default
);
-- signals connecting masters to intercon module
type rec_masters_signal is record
  syscon : arr_syscon_port(rng_masters);
  master : arr_master_port(rng_masters);
  slave : arr_slave_port(rng_masters);
end record;
constant masters_default : rec_masters_signal :=
(
  syscon => (others => syscon_default),
  master => (others => master_default),
  slave => (others => slave_default)
);
-- signals connecting slaves to intercon module
type rec_slaves_signal is record
  syscon : arr_syscon_port(rng_slaves);
  master : arr_master_port(rng_slaves);
  slave : arr_slave_port(rng_slaves);
end record;
constant slaves_default : rec_slaves_signal :=
(
  syscon => (others => syscon_default),
  master => (others => master_default),
  slave => (others => slave_default)
);
-- all signals connected to intercon module
type rec_intercon_signal is record
  syscon : rec_syscon_port;
  masters : rec_masters_signal;
  slaves : rec_slaves_signal;
end record;
constant intercon_default : rec_intercon_signal :=
(
  syscon => syscon_default,
  masters => masters_default,
  slaves => slaves_default
);
-- WISHBONE modules
-------------------
```

```
-- WB_INTERCON
 ------------------------------------------------------------------------------------------------
-- module responsible for carrying out the interconnection logic on which is based the communication
-- of the master and slave modules with each other through wishbone bus.
 ------------------------------------------------------------------------------------------------
 component wb_intercon is
  generic
  (
    nr_of_dbgports : positive := 1
  );
  port
  (
    SYSCON_I : in rec_syscon_port;
    SYSCON_MA_O : out arr_syscon_port(rng_masters);
    MASTER_I : in arr_master_port(rng_masters);
    SLAVE_O : out arr_slave_port(rng_masters);
    SYSCON_SL_O : out arr_syscon_port(rng_slaves);
    MASTER_O : out arr_master_port(rng_slaves);
    SLAVE_I : in arr_slave_port(rng_slaves);

    debug : inout std_logic_vector((nr_of_dbgports-1) downto 0)
  );
end component;
-- a single signal connects all WISHBONE devices to intercon
signal intercon : rec_intercon_signal := intercon_default;

-- partial address decoding is done in the WISHBONE system
-- some msbs of the address are used to select the appropriate slave device
-- in the interconnection logic (base address decoding)
-- a unique slave id is derived from the unsigned integer interpretation of these bits
-- the slave id is used as an index and identifies the slave module in array data types
-- the master id is assigned manually
-- please note, that the plx 9056 pci controller's local bus and the
-- software use byte addressing, while the WISHBONE system uses 32-bit
-- word addressing!
-- => WISHBONE address := local-bus/software address rightshift by 2
-- some lsbs of the address are used to select which memory unit or parameter to read
-- or write inside slave devices (address offset decoding)
-- FPGA resources can be saved by decoding only the necessary bits
-- example: if you have five parameters in your slave device,
-- then you decode only the three lsbs
-- absolute address := base address + offset
subtype rng_slave_select is natural range 29 downto 26;

-- WB_SYSCON
------------------------------------------------------------
-- module providing global system signals clock and reset
------------------------------------------------------------
component wb_syscon is
  generic
  (
    nr_of_dbgports : positive := 1
  );
  port
  (
    SYSCON_O : out rec_syscon_port;
    lbus_rst_i : in std_logic;
    sysclk_i : in std_logic;
    sdr_clk_o : out std_logic;
    sdr_clk_fb_i : in std_logic;
    debug : inout std_logic_vector((nr_of_dbgports-1) downto 0)
  );
end component;
signal lbus_rst : std_logic := '0';

--WB_MA_PLX
------------------------------------------------------------
```

```
-- plx local bus <=> WISHBONE bridge
-- The only master device connected to the wishbone bus
------------------------------------------------------------
constant ma_plx_id : natural := 0;
component wb_ma_plx is
  generic
  (
    nr_of_irqiports : positive := 1;
    nr_of_irqoports : positive := 1;
    nr_of_dbgports : positive := 1
  );
  port
  (
    SYSCON_I : in rec_syscon_port;
    MASTER_O : out rec_master_port;
    SLAVE_I : in rec_slave_port;

    IRQ_I : in std_logic_vector((nr_of_irqiports-1) downto 0);
    IRQ_O : out std_logic_vector((nr_of_irqoports-1) downto 0);
    plx_lreset_n_i : in std_logic;
    plx_lhold_i : in std_logic;
    plx_lholda_o : out std_logic;
    plx_ads_n_i : in std_logic;
    plx_lw_r_n_i : in std_logic;
    plx_lad_io : inout std_logic_vector(31 downto 0);
    plx_ready_n_o : out std_logic;
    plx_linti_n_o : out std_logic;
    lbus_rst_o : out std_logic;
    debug : inout std_logic_vector((nr_of_dbgports-1) downto 0)
  );
end component;
constant irqdev_plx_id : natural := 0;

-- WB_SL_SDR
------------------------------------------------------------------------------
-- sdram controller is the first slave device connected to the wishbone bus.
-- address space : x"00000000"-x"007FFFFF" --> 1MByte
-- 4 Commands available: nop,precharge,loadregister,operation.
------------------------------------------------------------------------------

-- define sdram memory space
constant sdr_baseadr : std_logic_vector(rng_adr) := x"0000_0000";
constant sdr_highadr : std_logic_vector(rng_adr) := x"007F_FFFF";
-- valid address bits
subtype rng_sdr_adr is natural range 22 downto 0;
-- define address of command register
constant sdr_cmd_offset : std_logic_vector(rng_adr) := x"0080_0000";
-- define bitmask for command register
subtype rng_sdr_cmd is natural range 1 downto 0;
-- define valid sdram commands
constant sdr_cmd_val_nop : std_logic_vector(rng_dat) := x"0000_0000";
constant sdr_cmd_val_precharge : std_logic_vector(rng_dat) := x"0000_0001";
constant sdr_cmd_val_loadmoderegister : std_logic_vector(rng_dat) := x"0000_0002";
constant sdr_cmd_val_operation : std_logic_vector(rng_dat) := x"0000_0003";
-- pragma SW_ON
--#define SDR_BASEADR 0x00000000
--#define SDR_HIGHADR 0x01FFFFFC
--#define SDR_CMD_OFFSET 0x02000000
--#define SDR_CMD_MASK 0x00000003
--#define SDR_CMD_VAL_NOP 0x00000000
--#define SDR_CMD_VAL_PRECHARGE 0x00000001
--#define SDR_CMD_VAL_LOADMODEREGISTER 0x00000002
--#define SDR_CMD_VAL_OPERATION 0x00000003
-- pragma SW_OFF
constant sl_sdr_id : natural := TO_INTEGER(unsigned(sdr_baseadr(rng_slave_select)));
component wb_sl_sdr is
  generic
  (
```

```
    nr_of_irqoports : positive := 1;
    nr_of_irqiports : positive := 1;
    nr_of_dbgports : positive := 1
  );
  port
  (
    SYSCON_I : in rec_syscon_port;
    MASTER_I : in rec_master_port;
    SLAVE_O : out rec_slave_port;
    IRQ_I : in std_logic_vector((nr_of_irqiports-1) downto 0);
    IRQ_O : out std_logic_vector((nr_of_irqoports-1) downto 0);
    sdr_clk_o : out std_logic;
    sdr_cke_o : out std_logic;
    sdr_cs_n_o : out std_logic;
    sdr_we_n_o : out std_logic;
    sdr_cas_n_o : out std_logic;
    sdr_ras_n_o : out std_logic;
    sdr_dqm_o : out std_logic_vector(1 downto 0);
    sdr_ba_o : out std_logic_vector(1 downto 0);
    sdr_adr_o : out std_logic_vector(12 downto 0);
    sdr_dq_io : inout std_logic_vector(15 downto 0);
    debug : inout std_logic_vector((nr_of_dbgports-1) downto 0)
  );
end component;
constant irqdev_sdr_id : natural := 1;

-- WB_SL_FLASH
--------------------------------------------------------------------------------
-- SPI flash controller is the second slave device connected to the wishbone bus.
-- address Space : x"04000000"-x"0401FFFF" --> 16KByte
-- 1 Command available: erase.
--------------------------------------------------------------------------------
-- define flash memory space
constant flash_baseadr : std_logic_vector(rng_adr) := x"0400_0000";
constant flash_highadr : std_logic_vector(rng_adr) := x"0401_FFFF";
-- valid address bits
subtype rng_flash_adr is natural range 16 downto 0;
-- define address of command register (write only!)
constant flash_cmd_offset : std_logic_vector(rng_adr) := x"0002_0000";
-- define bitmask for command register
subtype rng_flash_cmd is natural range 0 downto 0;
-- define erase command
constant flash_cmd_val_erase : std_logic_vector(rng_dat) := x"0000_0000";
-- pragma SW_ON
--#define FLASH_BASEADR 0x10000000
--#define FLASH_HIGHADR 0x1007FFFC
--#define FLASH_CMD_OFFSET 0x00080000 /* write only ! */
--#define FLASH_CMD_MASK 0x00000001
--#define FLASH_CMD_VAL_ERASE 0x00000000
-- pragma SW_OFF
constant sl_flash_id : natural := TO_INTEGER(unsigned(flash_baseadr(rng_slave_select)));
component wb_sl_flash is
  generic
  (
    nr_of_irqoports : positive := 1;
    nr_of_irqiports : positive := 1;
    nr_of_dbgports : positive := 1
  );
  port
  (
    SYSCON_I : in rec_syscon_port;
    MASTER_I : in rec_master_port;
    SLAVE_O : out rec_slave_port;
    IRQ_I : in std_logic_vector((nr_of_irqiports-1) downto 0);
    IRQ_O : out std_logic_vector((nr_of_irqoports-1) downto 0);
    flash_s_n_o : out std_logic;
    flash_c_o : out std_logic;
    flash_hold_n_o : out std_logic;
```

```
    flash_w_n_o : out std_logic;
    flash_d_o : out std_logic;
    flash_q_i : in std_logic;
    debug : inout std_logic_vector((nr_of_dbgports-1) downto 0)
  );
end component;
constant irqdev_flash_id : natural := 2;

-- WB_SL_GPIO
-------------------------------------------------------------------------------------------------------------
-- external GPIO, internal GPIO and LEDs is the third slave device connected to the wishbone bus.
-- address Space : x"08000000"-x"08000005"
-- MSS: this slave device is the most important slave device connected to the wishbone bus as it will be the one
-- on which the MSS module will be based
-- It is responsible for the communication of the wishbone bus with the outside world and especially
-- with the 8 input/output ports installed on the PIB between the fpga and the SUB-78-pin.
-------------------------------------------------------------------------------------------------------------
-- define GPIO base address
constant gpio_baseadr : std_logic_vector(rng_adr) := x"0800_0000";
-- valid address bits
subtype rng_gpio_adr is natural range 2 downto 0;
-- external I/Os p0, p1, p2 and p3
constant gpio_ext0_offset : std_logic_vector(rng_adr) := x"0000_0000";
-- external I/Os p4, p5, p6 and p7
constant gpio_ext1_offset : std_logic_vector(rng_adr) := x"0000_0001";
-- external I/Os output-enable
constant gpio_extoe_offset : std_logic_vector(rng_adr) := x"0000_0002";
-- define bitmask for external I/Os output-enable
subtype rng_gpio_extoe is natural range 7 downto 0;
-- internal I/Os
constant gpio_int_offset : std_logic_vector(rng_adr) := x"0000_0003";
-- define bitmask for internal I/Os
subtype rng_gpio_int is natural range 27 downto 0;
-- internal I/Os output-enable
constant gpio_intoe_offset : std_logic_vector(rng_adr) := x"0000_0004";
-- define bitmask for internal I/Os output-enable
subtype rng_gpio_intoe is natural range 27 downto 0;
-- LEDs
constant gpio_led_offset : std_logic_vector(rng_adr) := x"0000_0005";
-- define bitmask for LEDs
subtype rng_gpio_led is natural range 7 downto 0;
-- pragma SW_ON
--#define GPIO_BASEADR 0x20000000
--#define GPIO_EXT0_OFFSET 0x00000000
--#define GPIO_EXT1_OFFSET 0x00000004
--#define GPIO_EXTOE_OFFSET 0x00000008
--#define GPIO_EXTOE_MASK 0x000000FF
--#define GPIO_INT_OFFSET 0x0000000C
--#define GPIO_INT_MASK 0x0FFFFFFF
--#define GPIO_INTOE_OFFSET 0x00000010
--#define GPIO_INTOE_MASK 0x0FFFFFFF
--#define GPIO_LED_OFFSET 0x00000014
--#define GPIO_LED_MASK 0x000000FF
-- pragma SW_OFF
constant sl_gpio_id : natural := TO_INTEGER(unsigned(gpio_baseadr(rng_slave_select)));
component wb_sl_gpio is
  generic
  (
    nr_of_irqiports : positive := 1;
    nr_of_irqoports : positive := 1;
    nr_of_dbgports : positive := 1
  );
  port
  (
    SYSCON_I : in rec_syscon_port;
    MASTER_I : in rec_master_port;
    SLAVE_O : out rec_slave_port;
    IRQ_I : in std_logic_vector((nr_of_irqiports-1) downto 0);
```

```
    IRQ_O : out std_logic_vector((nr_of_irqoports-1) downto 0);
    led_o : out std_logic_vector(7 downto 0);
    ext_oe_o : out std_logic_vector(7 downto 0);
    ext_input_i : in std_logic_vector(63 downto 0);
    ext_output_o : out std_logic_vector(63 downto 0);
    gpio_io : inout std_logic_vector(27 downto 0);
    debug : inout std_logic_vector((nr_of_dbgports-1) downto 0)
  );
end component;
constant irqdev_gpio_id : natural := 3;
signal ext_oe : std_logic_vector(7 downto 0) := (others => '0');
signal ext_input : std_logic_vector(63 downto 0) := (others => '0');
signal ext_output : std_logic_vector(63 downto 0) := (others => '0');

  -- WB_SL_FLASH
-------------------------------------------------------------------------------
-- timer is the fourth slave device connected to the wishbone bus.
-------------------------------------------------------------------------------
-- define timer base address
constant timer_baseadr : std_logic_vector(rng_adr) := x"0C00_0000";
-- valid address bits
subtype rng_timer_adr is natural range 0 downto 0;
-- timer period in clocks / timer enable
constant timer_limit_offset : std_logic_vector(rng_adr) := x"0000_0000";
-- define value for timer diable
constant timer_limit_val_disable : std_logic_vector(rng_dat) := x"0000_0000";
-- interrupt acknowledge register (write only!)
constant timer_irqack_offset : std_logic_vector(rng_adr) := x"0000_0001";
-- bitmask for interrupt acknowledge register
subtype rng_timer_irqack is natural range 0 downto 0;
-- value to write for interrupt acknowledge
constant timer_irqack_val_acknowledge : std_logic_vector(rng_dat) := x"0000_0000";
-- pragma SW_ON
--#define TIMER_BASEADR 0x30000000
--#define TIMER_LIMIT_OFFSET 0x00000000
--#define TIMER_LIMIT_VAL_DISABLE 0x00000000
--#define TIMER_IRQACK_OFFSET 0x00000004 /* write only ! */
--#define TIMER_IRQACK_MASK 0x00000001
--#define TIMER_IRQACK_VAL_ACKNOWLEDGE 0x00000000
-- pragma SW_OFF
constant sl_timer_id : natural := TO_INTEGER(unsigned(timer_baseadr(rng_slave_select)));
component wb_sl_timer is
  generic
  (
    nr_of_irqiports : positive := 1;
    nr_of_irqoports : positive := 1;
    nr_of_dbgports : positive := 1
  );
  port
  (
    SYSCON_I : in rec_syscon_port;
    MASTER_I : in rec_master_port;
    SLAVE_O : out rec_slave_port;
    IRQ_I : in std_logic_vector((nr_of_irqiports-1) downto 0);
    IRQ_O : out std_logic_vector((nr_of_irqoports-1) downto 0);
    debug : inout std_logic_vector((nr_of_dbgports-1) downto 0)
  );
end component;
constant irqdev_timer_id : natural := 4;
constant irq_progtime_id : natural := 0; -- the one and only IRQ source

end wishbone;
```

Code 1 WISHBONE.vhd

In the code written above, there is only one master device and four slave devices connected to the WISHBONE bus. There are specific data types defined, which are responsible for the

implementation of the interconnection logic between the devices. An overview of intercon data type follows in the figure below.The structure of WISHBONE data type is already described in detail in subsection 3.2.5. As it is mentioned above, the most important slave device connected to the WISHBONE bus, meaning that it is the only slave device which communicates straight with the PIB ports, is the GPIO slave device. In this device there is implemented not only the communication of the local bus with the 8 ports of the PIB but also with the Internal Expansion Port J21 and with the LEDs. This device will be further analyzed, when the module "**wb_sl_gpio.vhd**" will be presented, in which it is developed. Its importance lies on the fact that it looks similar with the module "**wb_sl_mss.vhd**", which will implement the Microprocessor timing (MSS timing).

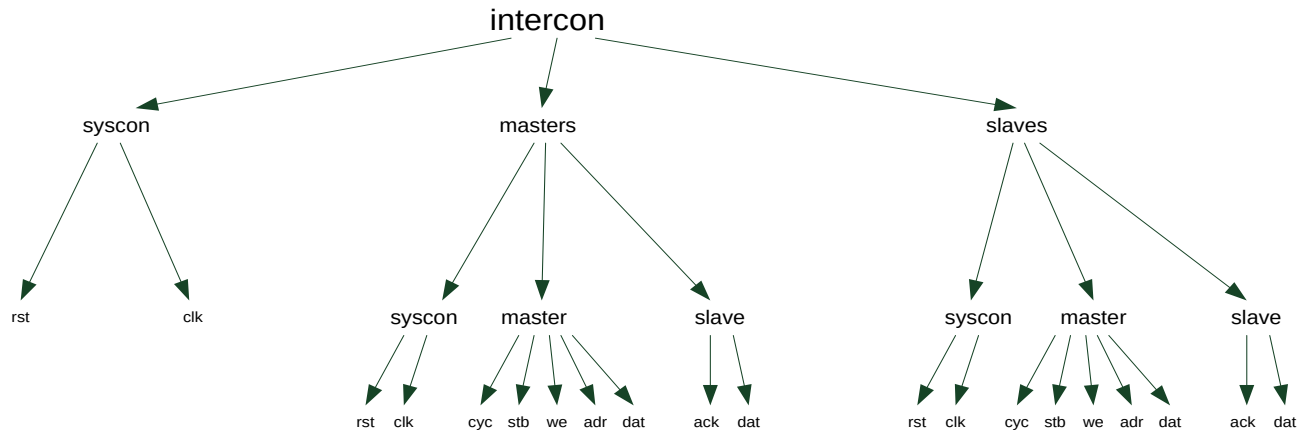


Figure 8 INTERCON Datatype

Furthermore, it is defined inside Code 1, that the WISHBONE address length is equal to 32 bits. The WISHBONE address has a specific structure which is depicted in Table 11.

31-30	29-26	25-0
unused	slave device select	slave device address field

Table 11 WISHBONE address structure

As a result, theoretically the WISHBONE bus can support the definition of up to 2^4 slave devices and each slave device can have a maximum number of 2^{26} addresses. Each slave device connected to the WISHBONE bus, occupies a specific address field which is displayed in Table 12.

Slave device	Start address	End address
SDRAM	0000_0000	007F_FFFF
FLASH	0400_0000	0401_FFFF
GPIO	0800_0000	0800_0005
TIMER	0C00_0000	0C00_0001

Table 12 WISHBONE slave devices address field

3.3.2 Top Module

Having already defined the modules which constitute the structure of WISHBONE, the signals of each module will be instatiated in "**pcis3base_top.vhd**". This is the top level entity of the design.What is more, the internal VHDL signals are mapped to the 100 pin connector CON7 of the general purpose I/O plug in boards, so the pinout of the user constraints file does not need to be changed for other plug in boards. The code of file "**pcis3base_top.vhd**" with the necessary comments follows below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.wishbone.all;

entity pcis3base_top is
  generic
    (
      nr_of_dbgports : positive := 1
      );
  port
    (
      pin_sysclk_i         : in    std_logic;
      pin_plx_lreset_n_i   : in    std_logic;
      pin_plx_lhold_i      : in    std_logic;
      pin_plx_lholda_o     : out   std_logic;
      pin_plx_ads_n_i      : in    std_logic;
      pin_plx_lw_r_n_i     : in    std_logic;
      pin_plx_lad_io       : inout std_logic_vector(31 downto 0);
      pin_plx_ready_n_o    : out   std_logic;
      pin_plx_linti_n_o    : out   std_logic;
      pin_flash_s_n_o      : out   std_logic;
      pin_flash_c_o        : out   std_logic;
      pin_flash_d_o        : out   std_logic;
      pin_flash_q_i        : in    std_logic;
      pin_led_o            : out   std_logic_vector(7 downto 0);
      pin_gpiomoduleport_io : inout std_logic_vector(92 downto 0);
      pin_gpio_io          : inout std_logic_vector(27 downto 0);
      pin_sdr_clk_o        : out   std_logic;
      pin_sdr_cke_o        : out   std_logic;
      pin_sdr_cs_n_o       : out   std_logic;
      pin_sdr_we_n_o       : out   std_logic;
      pin_sdr_cas_n_o      : out   std_logic;
      pin_sdr_ras_n_o      : out   std_logic;
      pin_sdr_dqm_o        : out   std_logic_vector(1 downto 0);
      pin_sdr_ba_o         : out   std_logic_vector(1 downto 0);
      pin_sdr_adr_o        : out   std_logic_vector(12 downto 0);
      pin_sdr_dq_io        : inout std_logic_vector(15 downto 0);
      pin_sdr_clk_fb_i     : in    std_logic;
      pin_debug_io         : inout std_logic_vector((nr_of_dbgports-1) downto 0)
      );
end pcis3base_top;

architecture RTL of pcis3base_top is
  --clk signal
  signal sysclk       : std_logic                 := '0';
  --local bus signals
  signal plx_lreset_n : std_logic                 := '0';
  signal plx_lhold    : std_logic                 := '0';
  signal plx_lholda   : std_logic                 := '0';
  signal plx_ads_n    : std_logic                 := '0';
  signal plx_lw_r_n   : std_logic                 := '0';
  signal plx_ready_n  : std_logic                 := '0';
  signal plx_linti_n  : std_logic                 := '0';
  --spi flash memory signals
  signal flash_s_n    : std_logic                 := '1';
```

```
  signal flash_c      : std_logic                  := '0';
  signal flash_d      : std_logic                  := '0';
  signal flash_q      : std_logic                  := '0';
  --led signal
  signal led          : std_logic_vector(7 downto 0) := (others => '0');
  constant INPUT      : std_logic                  := '0';
  constant OUTPUT     : std_logic                  := not INPUT;

  --record describing the state (Input/Output) and the Input/Output data of the 93 bit I/O port
  type rec_gpiomoduleport is record
    oe     : std_logic_vector(92 downto 0);
    input  : std_logic_vector(92 downto 0);
    output : std_logic_vector(92 downto 0);
  end record;
  constant gpiomoduleport_default : rec_gpiomoduleport :=
    (
      oe     => (others => INPUT),
      input  => (others => '0'),
      output => (others => '0')
      );
  signal gpiomoduleport : rec_gpiomoduleport           := gpiomoduleport_default;
  signal sdr_clk        : std_logic                    := '0';
  signal sdr_cke        : std_logic                    := '0';
  signal sdr_cs_n       : std_logic                    := '0';
  signal sdr_we_n       : std_logic                    := '0';
  signal sdr_cas_n      : std_logic                    := '0';
  signal sdr_ras_n      : std_logic                    := '0';
  signal sdr_dqm        : std_logic_vector(1 downto 0)  := (others => '0');
  signal sdr_ba         : std_logic_vector(1 downto 0)  := (others => '0');
  signal sdr_adr        : std_logic_vector(12 downto 0) := (others => '0');
  signal sdr_clk_fb     : std_logic                    := '0';

begin

  -- connect ports to internal signals
  -------------------------------------
  -- system clock
  sysclk           <= pin_sysclk_i;
  -- ports of local bus
  plx_lreset_n     <= pin_plx_lreset_n_i;
  plx_lhold        <= pin_plx_lhold_i;
  pin_plx_lholda_o <= plx_lholda;
  plx_ads_n        <= pin_plx_ads_n_i;
  plx_lw_r_n       <= pin_plx_lw_r_n_i;
  pin_plx_ready_n_o <= plx_ready_n;
  pin_plx_linti_n_o <= plx_linti_n;
  -- ports of SPI FLASH
  pin_flash_s_n_o  <= flash_s_n;
  pin_flash_c_o    <= flash_c;
  pin_flash_d_o    <= flash_d;
  flash_q          <= pin_flash_q_i;
  -- LEDs
  pin_led_o        <= led;
  -- 93-bit I/O port to 100 pol. extension for plugin-boards
  process(gpiomoduleport)
  begin
    for i in gpiomoduleport.output'range loop
      if gpiomoduleport.oe(i) = OUTPUT then
        pin_gpiomoduleport_io(i) <= gpiomoduleport.output(i);
      else
        pin_gpiomoduleport_io(i) <= 'Z';
      end if;
    end loop;
  end process;
  gpiomoduleport.input <= pin_gpiomoduleport_io;
  -- ports of SDRAM
  pin_sdr_clk_o       <= sdr_clk;
  pin_sdr_cke_o       <= sdr_cke;
```

```
pin_sdr_cs_n_o       <= sdr_cs_n;
pin_sdr_we_n_o       <= sdr_we_n;
pin_sdr_cas_n_o      <= sdr_cas_n;
pin_sdr_ras_n_o      <= sdr_ras_n;
pin_sdr_dqm_o        <= sdr_dqm;
pin_sdr_ba_o         <= sdr_ba;
pin_sdr_adr_o        <= sdr_adr;
sdr_clk_fb           <= pin_sdr_clk_fb_i;

-- instantiate wishbone modules
-------------------------------
inst_wb_intercon : wb_intercon
  generic map
  (
    nr_of_dbgports => 1
    )
  port map
  (
    SYSCON_I    => intercon.syscon,
    SYSCON_MA_O => intercon.masters.syscon,
    MASTER_I    => intercon.masters.master,
    SLAVE_O     => intercon.masters.slave,
    SYSCON_SL_O => intercon.slaves.syscon,
    MASTER_O    => intercon.slaves.master,
    SLAVE_I     => intercon.slaves.slave,
    debug       => open
    );

inst_wb_syscon : wb_syscon
  generic map
  (
    nr_of_dbgports => 1
    )
  port map
  (
    SYSCON_O     => intercon.syscon,
    lbus_rst_i   => lbus_rst,
    sysclk_i     => sysclk,
    sdr_clk_o    => sdr_clk,
    sdr_clk_fb_i => sdr_clk_fb,
    debug        => open
    );

inst_wb_ma_plx : wb_ma_plx
  generic map
  (
    nr_of_irqiports => nr_of_irqs,
    nr_of_irqoports => nr_of_irqs,
    nr_of_dbgports  => 1
    )
  port map
  (
    SYSCON_I => intercon.masters.syscon(ma_plx_id),
    MASTER_O => intercon.masters.master(ma_plx_id),
    SLAVE_I  => intercon.masters.slave(ma_plx_id),

    IRQ_I => irqs,
    IRQ_O => irqos(irqdev_plx_id),

    plx_lreset_n_i => plx_lreset_n,
    plx_lhold_i    => plx_lhold,
    plx_lholda_o   => plx_lholda,
    plx_ads_n_i    => plx_ads_n,
    plx_lw_r_n_i   => plx_lw_r_n,
    plx_lad_io     => pin_plx_lad_io,
    plx_ready_n_o  => plx_ready_n,
    plx_linti_n_o  => plx_linti_n,
```

```
    lbus_rst_o => lbus_rst,

    debug => open
    );

inst_wb_sl_sdr : wb_sl_sdr
  generic map
  (
    nr_of_irqiports => nr_of_irqs,
    nr_of_irqoports => nr_of_irqs,
    nr_of_dbgports  => 1
    )
  port map
  (
    SYSCON_I => intercon.slaves.syscon(sl_sdr_id),
    MASTER_I => intercon.slaves.master(sl_sdr_id),
    SLAVE_O  => intercon.slaves.slave(sl_sdr_id),

    IRQ_I => irqs,
    IRQ_O => irqos(irqdev_sdr_id),

    sdr_clk_o   => open,
    sdr_cke_o   => sdr_cke,
    sdr_cs_n_o  => sdr_cs_n,
    sdr_we_n_o  => sdr_we_n,
    sdr_cas_n_o => sdr_cas_n,
    sdr_ras_n_o => sdr_ras_n,
    sdr_dqm_o   => sdr_dqm,
    sdr_ba_o    => sdr_ba,
    sdr_adr_o   => sdr_adr,
    sdr_dq_io   => pin_sdr_dq_io,

    debug => open
    );

inst_wb_sl_flash : wb_sl_flash
  generic map
  (
    nr_of_irqiports => nr_of_irqs,
    nr_of_irqoports => nr_of_irqs,
    nr_of_dbgports  => 1
    )
  port map
  (
    SYSCON_I => intercon.slaves.syscon(sl_flash_id),
    MASTER_I => intercon.slaves.master(sl_flash_id),
    SLAVE_O  => intercon.slaves.slave(sl_flash_id),

    IRQ_I => irqs,
    IRQ_O => irqos(irqdev_flash_id),

    flash_s_n_o    => flash_s_n,
    flash_c_o      => flash_c,
    flash_hold_n_o => open,
    flash_w_n_o    => open,
    flash_d_o      => flash_d,
    flash_q_i      => flash_q,

    debug => open
    );

inst_wb_sl_gpio : wb_sl_gpio
  generic map
  (
    nr_of_irqiports => nr_of_irqs,
    nr_of_irqoports => nr_of_irqs,
    nr_of_dbgports  => 1
    )
```

```
  port map
  (
    SYSCON_I => intercon.slaves.syscon(sl_gpio_id),
    MASTER_I => intercon.slaves.master(sl_gpio_id),
    SLAVE_O  => intercon.slaves.slave(sl_gpio_id),

    IRQ_I => irqs,
    IRQ_O => irqos(irqdev_gpio_id),

    led_o => led,

    ext_oe_o     => ext_oe,
    ext_input_i  => ext_input,
    ext_output_o => ext_output,

    gpio_io => pin_gpio_io,

    debug => open
    );

inst_wb_sl_timer : wb_sl_timer
  generic map
  (
    nr_of_irqiports => nr_of_irqs,
    nr_of_irqoports => nr_of_irqs,
    nr_of_dbgports  => 1
    )
  port map
  (
    SYSCON_I => intercon.slaves.syscon(sl_timer_id),
    MASTER_I => intercon.slaves.master(sl_timer_id),
    SLAVE_O  => intercon.slaves.slave(sl_timer_id),

    IRQ_I => irqs,
    IRQ_O => irqos(irqdev_timer_id),

    debug => open
    );

-- build interrupt bus with device IRQs
---------------------------------------

irqs(irq_progtime_id) <= irqos(irqdev_timer_id)(irq_progtime_id);

-- connect internal signals to the 100 pol. extension port
----------------------------------------------------------

-- you will find further information about the signal assignment
-- in the the PIB64IO documentation. there is a table with the columns
-- "HDL Pin", "HDL Direction" and "Name", which should give you
-- an idea, how the connection of the entity port "pin_gpiomoduleport_io" to
-- the plugin-board is done
-- port mapping:
-- ext_out/input(7 downto 0) = port 0
-- ext_out/input(15 downto 8) = port 1
-- ...
-- ext_out/input(63 downto 56) = port 7

-- define each of the 93 I/Os as inputs, outputs or bidirs (default: all inputs)
gpiomoduleport.oe <=
  (
    1       => OUTPUT,
    2 to 5   => ext_oe(7),
    6 to 9   => ext_oe(6),
    10 to 11 => OUTPUT,
    12 to 15 => ext_oe(5),
    16 to 19 => ext_oe(4),
    20 to 21 => OUTPUT,
```

```
    22 to 25 => ext_oe(3),
    26 to 29 => ext_oe(2),
    30 to 31 => OUTPUT,
    32 to 35 => ext_oe(0),
    36 to 39 => ext_oe(1),
    40       => OUTPUT,
    50       => OUTPUT,
    51 to 54 => ext_oe(1),
    55 to 58 => ext_oe(0),
    59 to 60 => OUTPUT,
    61 to 64 => ext_oe(2),
    65 to 68 => ext_oe(3),
    69       => OUTPUT,
    71       => OUTPUT,
    72 to 75 => ext_oe(4),
    76 to 79 => ext_oe(5),
    80 to 81 => OUTPUT,
    82 to 85 => ext_oe(6),
    86 to 88 => ext_oe(7),
    90       => ext_oe(7),
    91       => OUTPUT,
    others   => INPUT
    );
-- assign outputs
gpiomoduleport.output(1)  <= ext_oe(7);  -- port 7 direction
gpiomoduleport.output(10) <= '0';
gpiomoduleport.output(11) <= ext_oe(5);  -- port 5 direction
gpiomoduleport.output(20) <= '0';
gpiomoduleport.output(21) <= ext_oe(3);  -- port 3 direction
gpiomoduleport.output(30) <= '0';
gpiomoduleport.output(31) <= ext_oe(0);  -- port 0 direction
gpiomoduleport.output(40) <= '0';
gpiomoduleport.output(50) <= ext_oe(1);  -- port 1 direction
gpiomoduleport.output(59) <= '0';
gpiomoduleport.output(60) <= ext_oe(2);  -- port 2 direction
gpiomoduleport.output(69) <= '0';
gpiomoduleport.output(71) <= ext_oe(4);  -- port 4 direction
gpiomoduleport.output(80) <= '0';
gpiomoduleport.output(81) <= ext_oe(6);  -- port 6 direction
gpiomoduleport.output(91) <= '0';

-- assign bidirs
gpiomoduleport.output(86) <= ext_output(7*8+7);        -- P7.7
gpiomoduleport.output(87) <= ext_output(7*8+5);        -- P7.5
gpiomoduleport.output(88) <= ext_output(7*8+3);        -- P7.3
gpiomoduleport.output(90) <= ext_output(7*8+1);        -- P7.1
ext_input(7*8+7)          <= gpiomoduleport.input(86);  -- P7.7
ext_input(7*8+5)          <= gpiomoduleport.input(87);  -- P7.5
ext_input(7*8+3)          <= gpiomoduleport.input(88);  -- P7.3
ext_input(7*8+1)          <= gpiomoduleport.input(90);  -- P7.1

process(ext_output, gpiomoduleport)
begin
  for i in 0 to 3 loop
    gpiomoduleport.output(2+i)  <= ext_output(7*8+2*i);      -- P7.even
    ext_input(7*8+2*i)          <= gpiomoduleport.input(2+i);
    gpiomoduleport.output(6+i)  <= ext_output(6*8+2*i);      -- P6.even
    ext_input(6*8+2*i)          <= gpiomoduleport.input(6+i);
    gpiomoduleport.output(12+i) <= ext_output(5*8+2*i);      -- P5.even
    ext_input(5*8+2*i)          <= gpiomoduleport.input(12+i);
    gpiomoduleport.output(16+i) <= ext_output(4*8+2*i);      -- P4.even
    ext_input(4*8+2*i)          <= gpiomoduleport.input(16+i);
    gpiomoduleport.output(22+i) <= ext_output(3*8+2*i);      -- P3.even
    ext_input(3*8+2*i)          <= gpiomoduleport.input(22+i);
    gpiomoduleport.output(26+i) <= ext_output(2*8+2*i);      -- P2.even
    ext_input(2*8+2*i)          <= gpiomoduleport.input(26+i);
    gpiomoduleport.output(32+i) <= ext_output(0*8+2*i);      -- P0.even
    ext_input(0*8+2*i)          <= gpiomoduleport.input(32+i);
```

```
    gpiomoduleport.output(36+i) <= ext_output(1*8+2*i);      -- P1.even
    ext_input(1*8+2*i)          <= gpiomoduleport.input(36+i);
    gpiomoduleport.output(54-i) <= ext_output(1*8+(2*i+1));  -- P1.odd
    ext_input(1*8+(2*i+1))      <= gpiomoduleport.input(54-i);
    gpiomoduleport.output(58-i) <= ext_output(0*8+(2*i+1));  -- P0.odd
    ext_input(0*8+(2*i+1))      <= gpiomoduleport.input(58-i);
    gpiomoduleport.output(64-i) <= ext_output(2*8+(2*i+1));  -- P2.odd
    ext_input(2*8+(2*i+1))      <= gpiomoduleport.input(64-i);
    gpiomoduleport.output(68-i) <= ext_output(3*8+(2*i+1));  -- P3.odd
    ext_input(3*8+(2*i+1))      <= gpiomoduleport.input(68-i);
    gpiomoduleport.output(75-i) <= ext_output(4*8+(2*i+1));  -- P4.odd
    ext_input(4*8+(2*i+1))      <= gpiomoduleport.input(75-i);
    gpiomoduleport.output(79-i) <= ext_output(5*8+(2*i+1));  -- P5.odd
    ext_input(5*8+(2*i+1))      <= gpiomoduleport.input(79-i);
    gpiomoduleport.output(85-i) <= ext_output(6*8+(2*i+1));  -- P6.odd
    ext_input(6*8+(2*i+1))      <= gpiomoduleport.input(85-i);
  end loop;
 end process;
end RTL;
```

Code 2 Pcis3base_top.vhd

Apart from the instantiation of the WISHBONE components, in the "**Pcis3base_top.vhd**" module there is also defined with which pads of the FPGA the PIB ports are speaking. Based on this information, the value of each bit of the 8 ports can afterwards be checked straight in the 78-pin CON9.

3.3.3 Intercon Module

Although the Intercon module will not change when the MSS module will be added to the WISHBONE bus, it is worth writing down its code, as it is the heart of the communication between the components of WISHBONE architecture. All WISHBONE devices are connected to this shared bus interconnection logic. Some MSBs of the address are used to select the appropriate slave. The code of file "**wb_intercon.vhd**" with the necessary comments follows below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use work.wishbone.all;


entity wb_intercon is
 generic
  (
   nr_of_dbgports : positive := 1
   );
 port
  (
   SYSCON_I    : in    rec_syscon_port;
   SYSCON_MA_O : out   arr_syscon_port(rng_masters);
   MASTER_I    : in    arr_master_port(rng_masters);
   SLAVE_O     : out   arr_slave_port(rng_masters);
   SYSCON_SL_O : out   arr_syscon_port(rng_slaves);
   MASTER_O    : out   arr_master_port(rng_slaves);
   SLAVE_I     : in    arr_slave_port(rng_slaves);
   debug       : inout std_logic_vector((nr_of_dbgports-1) downto 0)
   );
end wb_intercon;
```

```
architecture RTL of wb_intercon is

  signal intercon : rec_intercon_signal := intercon_default;
  signal wishbone : rec_wishbone_signal := wishbone_default;

begin

  -- connect ports to internal signals
  -----------------------------------
  intercon.syscon         <= SYSCON_I;
  SYSCON_MA_O             <= intercon.masters.syscon;
  intercon.masters.master <= MASTER_I;
  SLAVE_O                 <= intercon.masters.slave;
  SYSCON_SL_O             <= intercon.slaves.syscon;
  MASTER_O                <= intercon.slaves.master;
  intercon.slaves.slave   <= SLAVE_I;
  -- architecture implementation
  ----------------------------
  -- distribute the same syscon signal (rst, clk) to all wishbone devices
  wishbone.syscon         <= intercon.syscon;
  intercon.masters.syscon <= (others => wishbone.syscon);
  intercon.slaves.syscon  <= (others => wishbone.syscon);

  -- the plx pci controller is the one and only master at
  -- the on-chip-bus, no arbitration is needed
  -- connect this master to all slave devices
  wishbone.master         <= intercon.masters.master(ma_plx_id);
  intercon.masters.slave <= (others => wishbone.slave);

  -- building shared bus topology
  process(wishbone, intercon)
  begin
    wishbone.slave         <= slave_default;
    intercon.slaves.master <= (others => wishbone.master);
    -- partial address decoding, using some msbs of address to
    -- select the appropriate slave device
    -- the for-loop represents the slave mux
    for i in rng_slaves loop
      -- cyc is the one and only bit in the master signal, which has to be muxed,
      -- you could mux the whole master signal too, but this would waste FPGA resources
      intercon.slaves.master(i).cyc <= '0';  -- default: slave is deselected
      if i = TO_INTEGER(unsigned(wishbone.master.adr(rng_slave_select))) then
        intercon.slaves.master(i).cyc <= '1';
        wishbone.slave              <= intercon.slaves.slave(i);
      end if;
    end loop;
  end process;
end RTL;
```

Code 3 Wb_intercon.vhd

Reading the code above, apparently the signal cyc is the most important one as by default is set to null (‘0’) and only in case a slave device is selected, it is set to one(‘1’). By this signal, the master device knows on which slave device it has to send the data package.

3.3.4 Master_plx Module

As in the case of Intercon module, so in the case of Master_plx there will not be any change inside the code when the new slave device will be added on. However, it should be analysed how this module behaves as it is the only one master device connected to the WISHBONE

bus. What also makes this device extremely significant, is the fact that all the data transferred on the local bus are passed on it and it is, in turn, responsible for sending the data to the selected slave device (the intercon module is responsible for this selection as shown above). The code of file "**wb_ma_plx.vhd**" with the necessary comments follows below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.wishbone.all;


entity wb_ma_plx is
  generic
    (
      nr_of_irqiports : positive := 1;
      nr_of_irqoports : positive := 1;
      nr_of_dbgports  : positive := 1
      );
  port
    (
      SYSCON_I      : in    rec_syscon_port;
      MASTER_O      : out   rec_master_port;
      SLAVE_I       : in    rec_slave_port;
      IRQ_I         : in    std_logic_vector((nr_of_irqiports-1) downto 0);
      IRQ_O         : out   std_logic_vector((nr_of_irqoports-1) downto 0);
      plx_lreset_n_i : in    std_logic;
      plx_lhold_i    : in    std_logic;
      plx_lholda_o   : out   std_logic;
      plx_ads_n_i    : in    std_logic;
      plx_lw_r_n_i   : in    std_logic;
      plx_lad_io     : inout std_logic_vector(31 downto 0);
      plx_ready_n_o  : out   std_logic;
      plx_linti_n_o  : out   std_logic;
      lbus_rst_o     : out   std_logic;
      debug         : inout std_logic_vector((nr_of_dbgports-1) downto 0)
      );
end wb_ma_plx;


architecture RTL of wb_ma_plx is

  signal wishbone     : rec_wishbone_signal                        := wishbone_default;
  signal irqi         : std_logic_vector((nr_of_irqiports-1) downto 0) := (others => '0');
  signal irqo         : std_logic_vector((nr_of_irqoports-1) downto 0) := (others => '0');
  signal plx_lreset_n : std_logic                                  := '0';
  signal plx_lhold    : std_logic                                  := '0';
  signal plx_lholda   : std_logic                                  := '0';
  signal plx_ads_n    : std_logic_vector(1 downto 0)               := (others => '0');
  signal plx_lw_r_n   : std_logic                                  := '0';
  constant INPUT      : std_logic                                  := '0';
  constant OUTPUT     : std_logic                                  := not INPUT;
  type rec_plx_lad is record
    oe     : std_logic;
    input  : std_logic_vector(31 downto 0);
    output : std_logic_vector(31 downto 0);
  end record;
  constant plx_lad_default : rec_plx_lad :=
    (
      oe     => INPUT,
      input  => (others => '0'),
      output => (others => '0')
      );
  signal plx_lad     : rec_plx_lad := plx_lad_default;
  signal plx_ready_n : std_logic   := '0';
  signal plx_linti_n : std_logic   := '0';
  signal lbus_rst    : std_logic   := '0';
```

```
begin

  -- connect ports to internal signals
  -------------------------------------
  -- ports of WISHBONE bus
  wishbone.syscon <= SYSCON_I;
  MASTER_O        <= wishbone.master;
  wishbone.slave  <= SLAVE_I;
  -- ports of interrupt bus
  irqi            <= IRQ_I;
  IRQ_O           <= irqo;
  -- ports of local bus
  plx_lreset_n    <= plx_lreset_n_i;
  plx_lhold       <= plx_lhold_i;
  plx_lholda_o    <= plx_lholda;
  plx_ads_n(0)    <= plx_ads_n_i;
  plx_lw_r_n      <= plx_lw_r_n_i;
  process(plx_lad)
  begin
    if plx_lad.oe = OUTPUT then
      plx_lad_io <= plx_lad.output;
    else
      plx_lad_io <= (others => 'Z');
    end if;
  end process;
  plx_lad.input       <= plx_lad_io;
  plx_ready_n_o       <= plx_ready_n;
  plx_linti_n_o       <= plx_linti_n;
  -- port for using local bus reset (LRESET#) for resetting internal FPGA logic
  lbus_rst_o          <= lbus_rst;
  -- architecture implementation
  -------------------------------
  lbus_rst            <= not plx_lreset_n;
  -- as long as the plx is the one and only master at the on-chip-bus and cyc is not
  -- needed for arbitration purposes, this signal could be statically set to '1'
  wishbone.master.cyc <= '1';
  -- convert local bus handshake signals to wishbone handshake signals
  -- and demux the local bus lad signal to wishbone data out, data in and
  -- latch the address.
  -- note that the local bus uses byte addressing, while the wishbone system uses
  -- 32-bit-word-addressing! so the address is shifted by two. the I/O signal are
  -- routed through flip-flops to relax timing
  -- local bus and wishbone system are fully synchronous!
  process(wishbone)
  begin
    if wishbone.syscon.rst = '1' then
      plx_lholda          <= '0';
      plx_ready_n         <= '1';
      wishbone.master.stb <= '0';
      plx_lad.oe          <= '0';
      plx_ads_n(1)        <= '1';
    elsif wishbone.syscon.clk'event and wishbone.syscon.clk = '1' then
      -- let the internal timer interrupt pass through to external
      -- local bus interrupt
      plx_linti_n  <= not irqi(irq_progtime_id);
      plx_ads_n(1) <= plx_ads_n(0);
     --adress latching
      if plx_ads_n(0) = '0' then
     --the adress of wishbone bus is equal to the local bus address right shifted two times
        wishbone.master.adr <= b"00" & plx_lad.input(31 downto 2);
      --data latching
       else
        wishbone.master.dat <= plx_lad.input;
      end if;
      wishbone.master.we <= plx_lw_r_n;
      plx_lad.oe         <= wishbone.slave.ack and (not wishbone.master.we);
      plx_ready_n        <= not wishbone.slave.ack;
      plx_lad.output     <= wishbone.slave.dat;
```

```
    plx_lholda        <= plx_lhold;
    --Write Cycle
    if wishbone.master.we = '1' then
      if plx_ads_n(1) = '0' then
        wishbone.master.stb <= '1';
      elsif wishbone.slave.ack = '1' then
        wishbone.master.stb <= '0';
      end if;
    --Read Cycle
    else
      if plx_ads_n(0) = '0' then
        wishbone.master.stb <= '1';
      elsif wishbone.slave.ack = '1' then
        wishbone.master.stb <= '0';
      end if;
    end if;
  end if;
 end process;
end RTL;
```

Code 4 Wb_ma_plx.vhd

Furthermore, reading the code above it is obvious that apart from passing on the data(wishbone.master.adr, wishbone.master.dat) to the slave device, this module has to accomplish another work, which is the correct translation of the local bus control signals(plx_lw_r_n,plx_lhold,plx_ads_n) to the corresponding WISHBONE control signals(wishbone.master.stb,wishbone.master.we).As stated in the top module previously, the address which is transferred on the local bus is not equal to the address which is passed on the WISHBONE bus. Specifically, it is four times greater (local bus address=WISHBONE bus address << 2). This relationship between the local bus address and WISHBONE bus address is defined in the master module.

3.3.5 GPIO Module

At the begin of subchapter 3.3 it was mentioned, that the GPIO Module is the most important slave device connected to the WISHBONE bus in matters of the new slave device which will be developed. It will be also the only slave device whose function will be explained in this subsection as the others are irrelevant to the new module which has to be implemented.

The GPIO slave device implements the communication of the WISHBONE with the 8 ports of the PIB, with the LEDs and with the Internal Expansion Port J21. In other words, the GPIO module is the last stage in the process before the data package, sent to the FPGA from PC through local bus, goes to the PIB ports and then to the CON9 pins. The code of file "**wb_sl_gpio.vhd**" with the necessary comments follows below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.wishbone.all;


entity wb_sl_gpio is
  generic
    (
      nr_of_irqiports : positive := 1;
      nr_of_irqoports : positive := 1;
      nr_of_dbgports  : positive := 1
      );
  port
```

```
  (
    SYSCON_I    : in    rec_syscon_port;
    MASTER_I    : in    rec_master_port;
    SLAVE_O     : out   rec_slave_port;
    IRQ_I       : in    std_logic_vector((nr_of_irqiports-1) downto 0);
    IRQ_O       : out   std_logic_vector((nr_of_irqoports-1) downto 0);
    led_o       : out   std_logic_vector(7 downto 0);
    ext_oe_o    : out   std_logic_vector(7 downto 0);
    ext_input_i  : in    std_logic_vector(63 downto 0);
    ext_output_o : out   std_logic_vector(63 downto 0);
    gpio_io     : inout std_logic_vector(27 downto 0);
    debug       : inout std_logic_vector((nr_of_dbgports-1) downto 0)
    );
end wb_sl_gpio;


architecture RTL of wb_sl_gpio is

  constant INPUT    : std_logic                                 := '0';
  constant OUTPUT   : std_logic                                   := not INPUT;
  constant LEDOFF   : std_logic                                  := '0';
  constant LEDON    : std_logic                                  := not LEDOFF;
  constant INPORT   : natural                                   := 0;
  constant INBUFF   : natural                                   := 1;
  signal wishbone   : rec_wishbone_signal                           := wishbone_default;
  signal irqi       : std_logic_vector((nr_of_irqiports-1) downto 0) := (others => '0');
  signal irqo       : std_logic_vector((nr_of_irqoports-1) downto 0) := (others => '0');
  signal led        : std_logic_vector(7 downto 0)                  := (others => '0');
  signal ext_oe     : std_logic_vector(7 downto 0)                   := (others => INPUT);
  type arr_std64 is array(natural range <>) of std_logic_vector(63 downto 0);
  signal ext_input  : arr_std64(1 downto 0)                         := (others => (others => '0'));
  signal ext_output : std_logic_vector(63 downto 0)                 := (others => '0');
  type arr_std28 is array(natural range <>) of std_logic_vector(27 downto 0);
  type rec_gpio is record
    oe     : std_logic_vector(27 downto 0);
    input  : arr_std28(1 downto 0);
    output : std_logic_vector(27 downto 0);
  end record;
  constant gpio_default : rec_gpio :=
    (
      oe     => (others => INPUT),
      input  => (others => (others => '0')),
      output => (others => '0')
      );
  signal gpio : rec_gpio := gpio_default;

begin
  -- connect ports to internal signals
  -------------------------------------
  -- ports of WISHBONE bus
  wishbone.syscon   <= SYSCON_I;
  wishbone.master   <= MASTER_I;
  SLAVE_O           <= wishbone.slave;
  -- ports of interrupt bus
  irqi              <= IRQ_I;
  IRQ_O             <= irqo;
  -- LEDs
  led_o             <= led;
  -- signals for 64-bit GPIO over bus transceiver circuits on the plugin-board
  ext_oe_o          <= ext_oe;
  ext_input(INPORT) <= ext_input_i;
  ext_output_o      <= ext_output;
  -- port of 28-bit GPIO at the internal expansion connector
  process(gpio)
  begin
    for i in gpio.output'range loop
      if gpio.oe(i) = OUTPUT then
        gpio_io(i) <= gpio.output(i);
```

```
    else
      gpio_io(i) <= 'Z';
    end if;
  end loop;
end process;
gpio.input(INPORT) <= gpio_io;

-- architecture implementation
------------------------------
-- WISHBONE handshake, internal 28-bit GPIO and external 64-bit GPIO
-- input-, output- and output-enable-registers, LED control register
process(wishbone, ext_input, ext_oe, gpio, led)
begin
  -- clocked process part
  ----------------------
  if wishbone.syscon.rst = '1' then
    wishbone.slave.ack <= '0';
    ext_oe            <= (others => INPUT);
    ext_output          <= (others => '0');
    gpio.oe           <= (others => INPUT);
    gpio.output         <= (others => '0');
    led               <= (others => LEDOFF);
  elsif wishbone.syscon.clk'event and wishbone.syscon.clk = '1' then
    -- WISHBONE handshake acknowledge response for registers
    if wishbone.slave.ack = '1' then
      wishbone.slave.ack <= '0';
    elsif (wishbone.master.cyc and wishbone.master.stb) = '1' then
      wishbone.slave.ack <= '1';
    end if;
    -- WISHBONE write address-decoding
    if (wishbone.master.cyc and wishbone.master.stb and wishbone.master.we) = '1' then
      case wishbone.master.adr(rng_gpio_adr) is
        -- output register port 0, port 1, port 2 and port 3 (external GPIO)
        when gpio_ext0_offset(rng_gpio_adr) =>
          ext_output(31 downto 0) <= wishbone.master.dat;
          -- output register port 4, port 5, port 6 and port 7 (external GPIO)
        when gpio_ext1_offset(rng_gpio_adr) =>
          ext_output(63 downto 32) <= wishbone.master.dat;
          -- output-enable register for all ports
          -- you can configure the whole port as an out- or input, but not single bits!
        when gpio_extoe_offset(rng_gpio_adr) =>
          ext_oe <= wishbone.master.dat(rng_gpio_extoe);
          -- output register for internal GPIO
        when gpio_int_offset(rng_gpio_adr) =>
          gpio.output <= wishbone.master.dat(rng_gpio_int);
          -- output-enable register for internal GPIO
          -- you can configure each bit as an out- or input
        when gpio_intoe_offset(rng_gpio_adr) =>
          gpio.oe <= wishbone.master.dat(rng_gpio_intoe);
          -- LED control register
        when gpio_led_offset(rng_gpio_adr) =>
          led <= wishbone.master.dat(rng_gpio_led);
        when others => null;
      end case;
    end if;
    -- route GPIO inputs through flip-flops
    ext_input(INBUFF)  <= ext_input(INPORT);
    gpio.input(INBUFF) <= gpio.input(INPORT);
  end if;
  -- combinatorial process part
  -----------------------------
  -- WISHBONE read address-decoding
  wishbone.slave.dat <= (others => '0');  -- default value
  case wishbone.master.adr(rng_gpio_adr) is
    -- input register port 0, port 1, port 2 and port 3 (external GPIO)
    when gpio_ext0_offset(rng_gpio_adr) =>
      wishbone.slave.dat <= ext_input(INBUFF)(31 downto 0);
      -- input register port 4, port 5, port 6 and port 7 (external GPIO)
```

```
    when gpio_ext1_offset(rng_gpio_adr) =>
      wishbone.slave.dat <= ext_input(INBUFF)(63 downto 32);
      -- output-enable register for all ports
    when gpio_extoe_offset(rng_gpio_adr) =>
      wishbone.slave.dat(rng_gpio_extoe) <= ext_oe;
      -- input register for internal GPIO
    when gpio_int_offset(rng_gpio_adr) =>
      wishbone.slave.dat(rng_gpio_int) <= gpio.input(INBUFF);
      -- output-enable register for internal GPIO
    when gpio_intoe_offset(rng_gpio_adr) =>
      wishbone.slave.dat(rng_gpio_intoe) <= gpio.oe;
      -- LED control register
    when gpio_led_offset(rng_gpio_adr) =>
      wishbone.slave.dat(rng_gpio_led) <= led;
    when others => null;
   end case;
  end process;
end RTL;
```

Code 5 Wb_sl_gpio.vhd

In the GPIO Module, a register can be either written (send out information), or be read. As far as the ports of the PIB are concerned the state (Input / Output) of each port is defined by ext_oe register.

3.4 Testing Wishbone Bus Architecture with Modelsim

Having already explained the main WISHBONE components the next step in order to clarify how the WISHBONE architecture works is to develop a testbench that would actually simulate the local bus. In other words the testbench will replace the local bus. In such a way the exact dataflow can be produced, which is already depicted in Figure 5 for both cases of write and read cycle.

In Figure 9 follows an overview of how the whole system looks like starting from PC(local bus) up to the CON9 connector.

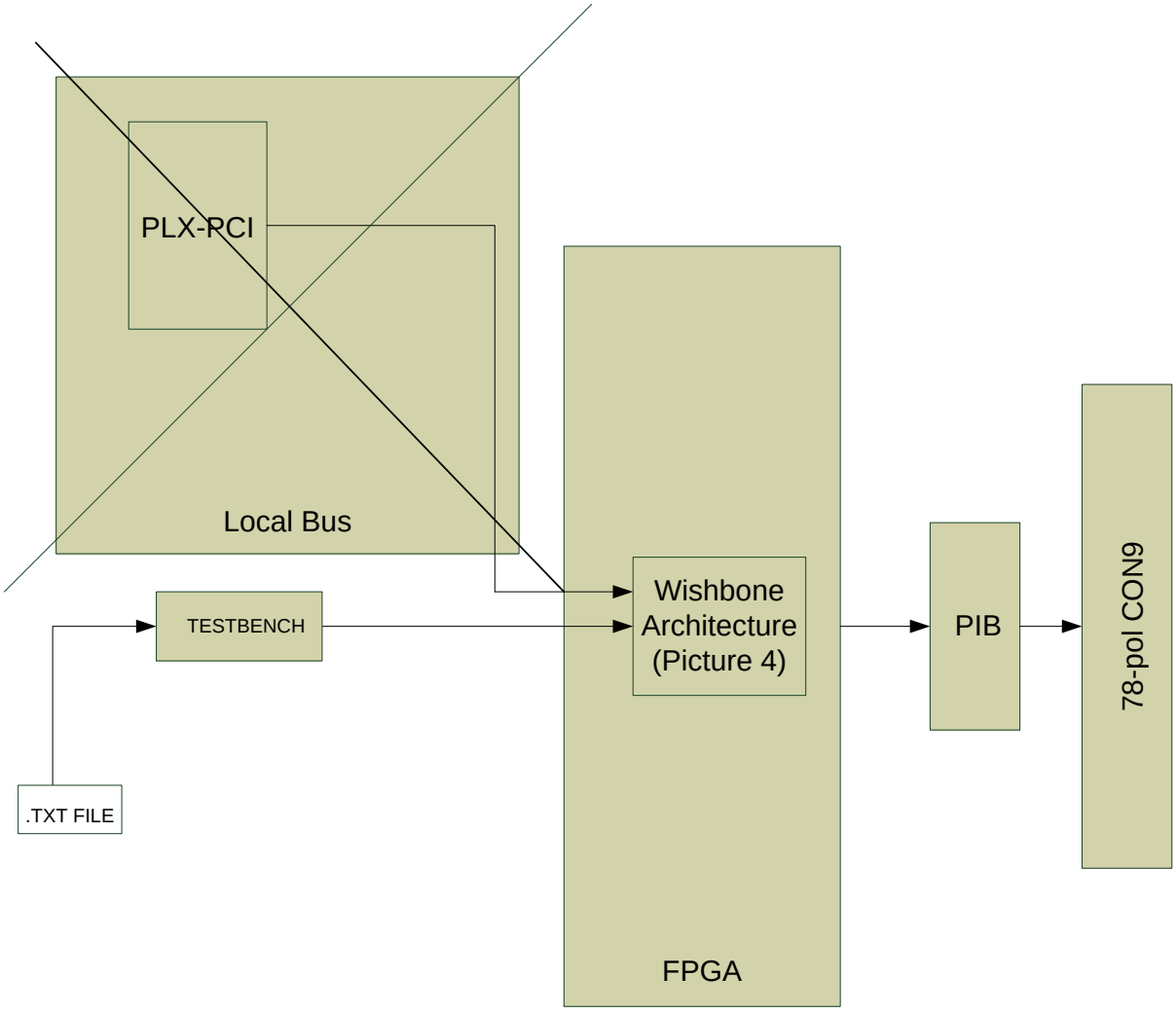


Figure 9 Local Bus Testbench

Obviously, the main goal for writing this testbench is to communicate directly with the WISHBONE bus and in this way to conceive how it works. Using this testbench the engineer is independent from PLX/PCI as he himself can produce the local bus signals which are sent to the WISHBONE architecture and stimulate its components.

The code of the testbench, which will also be used later in the simulation of the MSS module, is presented below.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.iuic_fpga_tb_package.all;
library std;
use std.textio.all;

entity cmd_if is
  generic(
```

```
    g_cmd_file : string := "C:\Users\toshiba\Desktop\commandfile.txt";
    clk_period : time   := 20 ns
    );
end cmd_if;


architecture behave of cmd_if is
  signal clk                                  : std_logic := '0';
  signal counter                                : integer   := 1;
  signal lhold, lholda, rst, lwr, ads, ready      : std_logic;
  signal lad                                  : std_logic_vector(31 downto 0);
  signal led                                  : std_logic_vector(7 downto 0);
  signal gpio                                   : std_logic_vector(92 downto 0);
  component pcis3base_top is
    generic
     (
       nr_of_dbgports : positive := 1
       );
    port
     (
       pin_sysclk_i          : in    std_logic;
       pin_plx_lreset_n_i    : in    std_logic;
       pin_plx_lhold_i       : in    std_logic;
       pin_plx_lholda_o      : out   std_logic;
       pin_plx_ads_n_i       : in    std_logic;
       pin_plx_lw_r_n_i      : in    std_logic;
       pin_plx_lad_io        : inout std_logic_vector(31 downto 0);
       pin_plx_ready_n_o     : out   std_logic;
       pin_plx_linti_n_o     : out   std_logic;
       pin_led_o             : out   std_logic_vector(7 downto 0);
       pin_gpiomoduleport_io : inout std_logic_vector(92 downto 0);
       pin_gpio_io           : inout std_logic_vector(27 downto 0);
       pin_debug_io          : inout std_logic_vector((nr_of_dbgports-1) downto 0)
       );
  end component;
begin
--clk frequency=50 Mhz
  clk <= not clk after clk_period/2;
---------------------------------------
-- Process to generate stimulus ...
---------------------------------------
  stimuli  : pcis3base_top port map(clk, rst, lhold, lholda, ads, lwr, lad, ready, open, led, gpio, open, open);
  stimuli : process
    file INFILE           : text open read_mode is g_cmd_file;
    variable CMD          : string(1 to 5);
    variable IN_LINE      : line;
    variable STRING_IN    : string (1 to 255);
    variable string_in_len : integer := 0;
    variable a_str        : string (1 to 8);
    variable d_str        : string (1 to 8);
    variable ADR          : std_logic_vector(31 downto 0);
    variable DATA         : std_logic_vector(31 downto 0);
  begin
    ---------------------------------------------------------------------------------------------
    -- Initialization of output signals
    ---------------------------------------------------------------------------------------------
    lhold     <= '0';
    rst       <= '0';
    lwr       <= '1';
    ads       <= '1';
    lad       <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    wait for 100 ns;
    ---------------------------------------------------------------------------------------------
    -- Commandfile interpreter
    ---------------------------------------------------------------------------------------------
    programm : while not endfile(INFILE) loop
      -- Read a line from the commandfile
      str_read(INFILE, STRING_IN, string_in_len);
```

```
    counter <= counter+1;
    if ((counter mod 4 = 1 and counter /= 1) or counter = 10) then
      wait for 2000 ns;
    end if;
    INLOOP :
      -- Jump over comments and empty lines
    while (STRING_IN(1) = '-') loop
      report STRING_IN(1 to string_in_len);
      str_read(INFILE, STRING_IN, string_in_len);
    end loop INLOOP;
    CMD := STRING_IN(1 to 5);
    report "COMMAND-FILE:  " & STRING_IN(1 to string_in_len);
    ------------------------------------------------------------------------
    --   Write Command
    ------------------------------------------------------------------------
    if (CMD(1 to 2) = "wr") then
      a_str := STRING_IN(4 to 11);
      d_str := STRING_IN(13 to 20);
      ADR   := str2vector(a_str);
      DATA  := str2vector(d_str);
      report "MSS write   ADR: " & a_str & "  Data: " & d_str;
      lhold <= '1';
      ads   <= '0';
      lad   <= ADR;
      wait for clk_period;
      ads   <= '1';
      lad   <= DATA;
     wait for 8*clk_period;
      lad   <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      wait for clk_period;
    end if;
    ------------------------------------------------------------------------
    --   Read Command
    ------------------------------------------------------------------------
    if (CMD(1 to 2) = "rd") then
      a_str := STRING_IN(4 to 11);
      ADR   := str2vector(a_str);
      lhold <= '1';
      lwr   <= '0';
      wait for clk_period;
      ads   <= '0';
      lad   <= ADR;
      wait for clk_period;
      lad   <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      ads   <= '1';
      wait for 8*clk_period;
      lad   <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      wait for clk_period;
      lwr   <= '1';
     wait for clk_period;
    end if;
   end loop programm;
   wait;  -- wait forever if cmdfile is completely read
 end process STIMULI;
end architecture behave;
```

Code 6 Testbench.vhd

Using this testbench a text file is read, which contains 2 type of commands, write and read. For reading each line of this file, it is used the function *str_read()* which has three operands. The first one is the name of the text file, the second one is a line of this file and the third one the length of this line. The code of this function is included in package *iuic_fpga_tb_package* and follows below.

```
-- read variable length string from input file

 procedure str_read(file in_file :     text;
                    res_string   : out string;
                    res_len      : out integer) is

  variable lead_ws     : boolean := true;
  variable ws_count    : integer := 0;
  variable res_len_cnt : integer := 0;
  variable l           : line;
  variable c           : character;
  variable is_string   : boolean;

 begin
  readline(in_file, l);
  -- clear the contents of the result string
  for i in res_string'range loop
    res_string(i) := '+';
  end loop;
  res_len_cnt := 0;
  res_len     := 0;
  ws_count    := 0;

  -- read all characters of the line, up to the length
  -- of the results string
  for i in res_string'range loop
    read(l, c, is_string);
    if not is_string then             -- found end of line
      res_string(i - ws_count) := '#';
      exit;
    else
      if lead_ws then

        if ((c = ' ') or (c = ht)) then
          ws_count := ws_count + 1;
        else
          lead_ws  := false;
          res_string(i - ws_count) := c;
          res_len_cnt:= res_len_cnt + 1;
        end if;

      else

        if ((c = ' ') or (c = ht)) then
          lead_ws := true;
        end if;

        res_string(i - ws_count) := c;
        res_len_cnt := res_len_cnt + 1;

      end if;
    end if;

  end loop;

  res_len := res_len_cnt;

 end str_read;
```

Code 7 Str_read()

In the testbench source file above, the code of the top module is simplified, as all the unused slave devices are erased, meaning the SDRAM, SPI FLASH and TIMER slave devices. Obviously, the SYSCON device is also erased as the testbench itself provides all the other devices with the clock signal. Furthermore, the clock frequency used in the testbench is equal

to the clock frequency of the clock oscillator supplied to the FPGA. In both cases (wr & rd) of assembly commands written in the text file the source code generates the whole signal values sequence already showed in Figure 5. It will be presented step by step below in the simulation environment of Modelsim, how the local bus signals values are translated to the corresponding WISHBONE bus signals.

First of all, it is mentioned in Code 8, which assembly commands are written in the"commandfile.txt" which will be read by the testbench.

```
wr  20000008  000000FF
wr  20000000  04030201
wr  20000004  08070605
rd  20000004
rd  20000000
```

Code 8 Commandfile.txt (GPIO)

Taking a look inside the code of the WISHBONE package defined in Code 1 it can be understood that the addresses written inside the text file are the addresses which correspond to the GPIO module. It has to be mentioned that the addresses in "Commandife.txt" are the local bus addresses which will be translated to the corresponding WISHBONE bus addresses inside the master device connected to the WISHBONE bus. For instance, the local bus address 20000008 is equal to the WISHBONE bus address 08000002 as in general, the local bus address is equal to the WISHBONE bus address when the latter is left shifted two times.

It is high time to present the simulation, which will give an overview of the local bus-WISHBONE relationship.

Step 1

```
wr  20000008  000000FF
```

The testbench reads the first command and passes on its information to the **wishbone** and **ext_oe** signals.

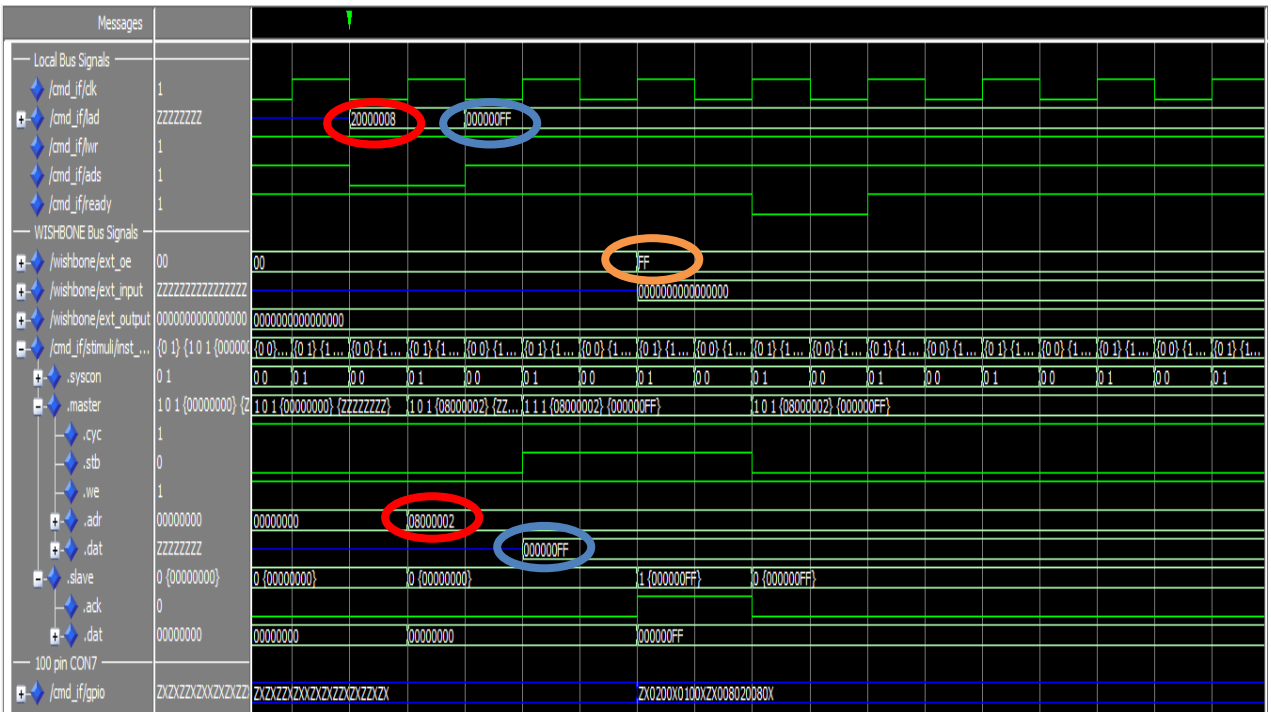


Figure 10 Local Bus-Wishbone Testbench pic.1

There are 10 signals included in the waveform. **Clk** is the clock signal, the next 4 signals (**lwr,lad,ads,ready**) belong to the local bus and the last 5 signals belong to the WISHBONE module. **Ext_oe** is a 8 bit std_logic_vector signal which shows the I/O state of each of the 8 ports in the PIB (1 for output, 0 for input). It has to be clarified that meaning output is the direction from FPGA towards the ports of the PIB and meaning input is the opposite direction. **Ext_input** is a 64 bit std_logic_vector signal which has the values of the 64 IOs when they receive information from outside through the CON9 connector. Starting from bit 0, the first 8 bits (bit0-bit7) is the information sent through Port 0 towards FPGA, the next 8 bits (bit8-bit15) is the information sent through Port 1 etc. **Ext_output** is also a 64 bit std_logic_vector signal, which respectively has the values, which will be sent to the 64 IOs from the FPGA. **Gpio** is a 93 bit std_logic_vector signal, which shows the value of each of the 93 pins of the CON7 connector between the FPGA and the PIB. However, the most important signal in the waveform above is the **wishbone** as it is the signal with which all WISHBONE modules communicate.

It is obvious that the command cycle depicted in Figure 10 is a write cycle as the **lwr** signal has always the value '1'. Observing the **lad** signal, it can be immediately conceived that the address is equal to 2000_0008 and the data package is equal to 0000_00FF. What is more, it is displayed above, how the **lad** signal's value is multiplexed, as in case when the control signal **ads** is equal to '0' the lad signal's value is passed on the **wishbone.master.adr,** while in case when the signal **ads** is equal to '1' the lad signal's value is passed on the **wishbone.master.dat**. As already mentioned, the address value passed on the **wishbone.master.adr (0800_0002)** is equal to the value carried by signal **lad (2000_0008)** right shifted two times. The address is multiplexed correctly as the data package (0000_00FF) is passed on register **ext_oe** which corresponds to address 0800_0002. It was stated earlier, that register **ext_oe** shows if a port is behaving as input or as output. As all bits of this signal are set to '1' all ports of PIB are outputs.

Step 2

```
wr   20000004  08070605
```

The testbench reads the third command and passes on its information to the **wishbone** and **ext_output** signals.

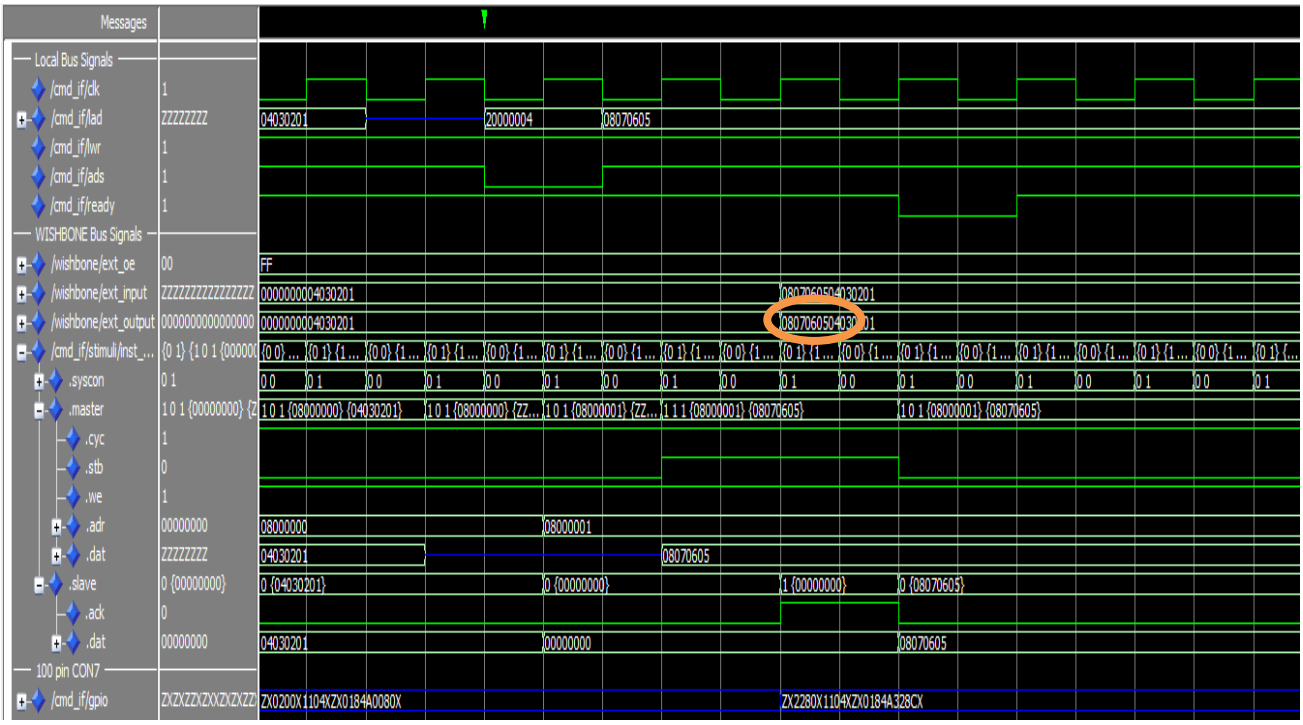


Figure 11 Local Bus-Wishbone Testbench pic.3

Exactly as before, the command cycle depicted in Figure 11 is a write cycle as the **lwr** signal has always the value '1'. The **lad** signal carries now however a different address (2000_0004) , which corresponds to the 32 MSBs of register **ext_output.** The data packages 05, 06, 07 and 08 are sent in Ports 4,5,6,7 respectively.

Step 3

```
rd   20000004
```

The testbench reads the fourth command and passes on its information to the **wishbone** and **lad** signals.

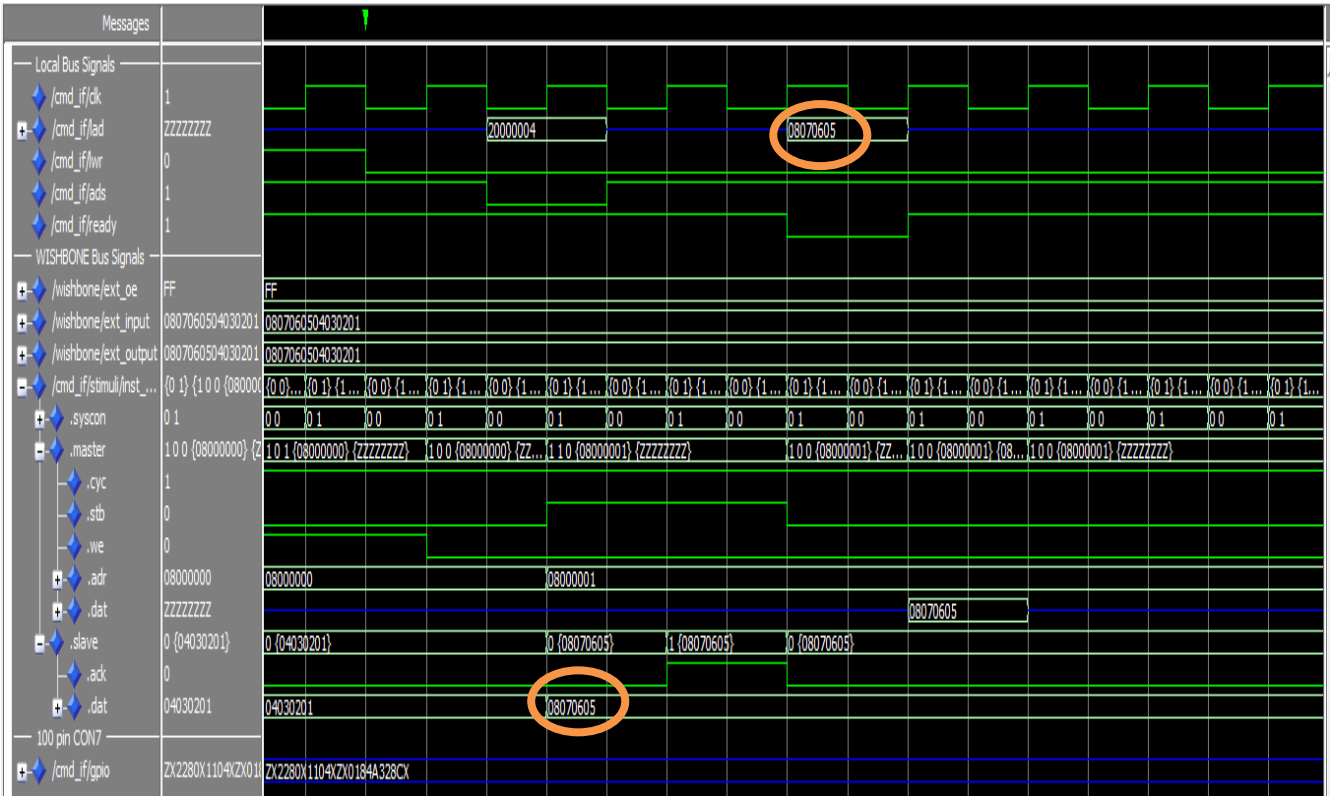


Figure 12 Local Bus-Wishbone Testbench pic.4

The command cycle depicted in Figure 12 is a read cycle as the **lwr** signal has the value '0'. The **lad** signal carries the address "2000_0004" which matches to the 32 MSBs of register **ext_output.** Inside the master module connected to the WISHBONE bus, which is described in Code 4, the value of signal **wishbone.slave.dat** is passed on the local bus signal **lad.** Apparently, in this case the value of of signal **wishbone.slave.dat** is equal to the information stored in address "0800_0001" of WISHBONE.

Step 4

```
rd   20000000
```

The testbench reads the fifth command and passes on its information to the **wishbone** and **lad** signals.

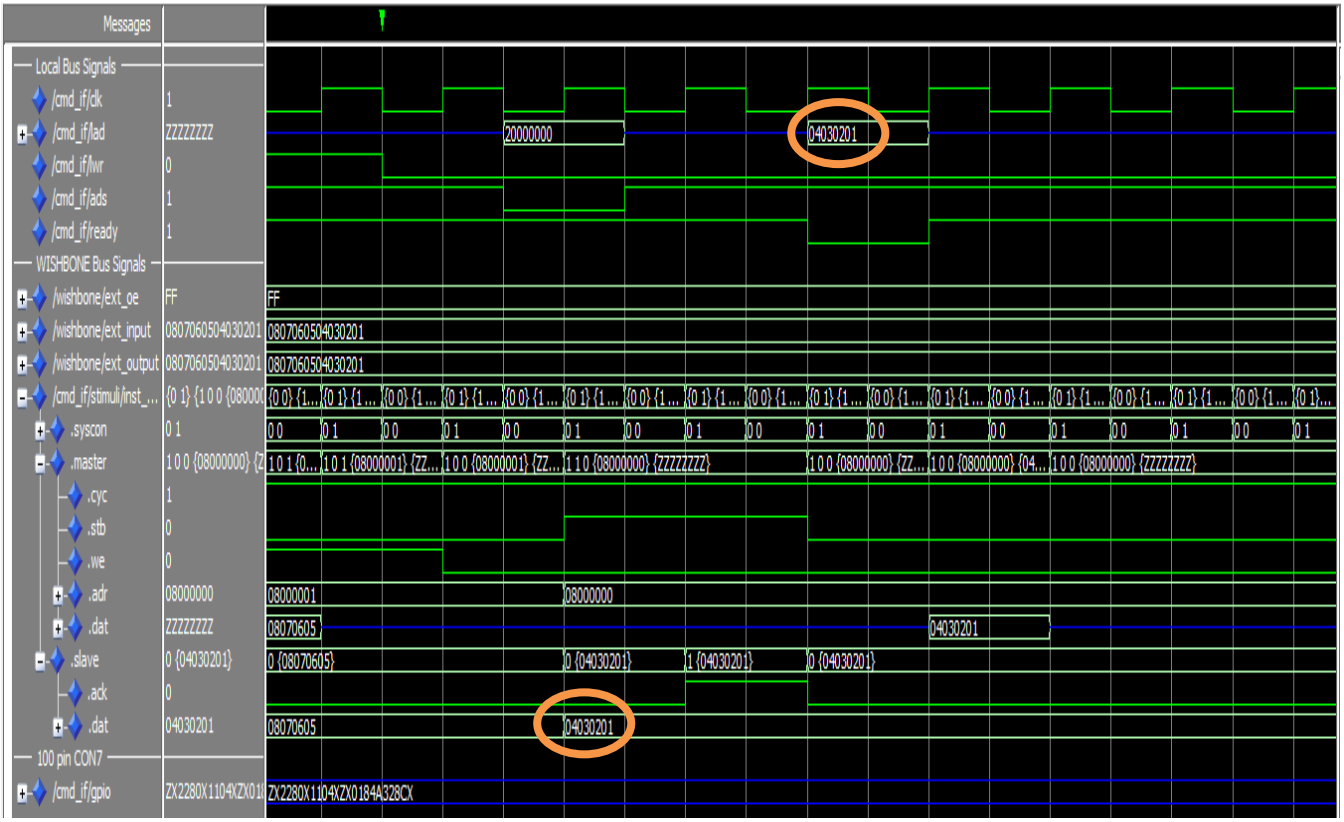


Figure 13 Local Bus-Wishbone Testbench pic.5

The command cycle depicted in Figure 13 is also a read cycle as the **lwr** signal has the value '0'. The **lad** signal carries the address "2000_0000" which matches to the 32 LSBs of register **ext_output.** The value of signal **wishbone.slave.dat,** which corresponds now to address "0800_0000" of WISHBONE is passed on the local bus signal **lad** when the control signal **ready** takes the value '0'**.** The value of signal **ready** is also defined in the master module **(**Code 4**),** and has always the opposite value in compare with signal **wishbone.slave.ack.** It should be underlined, that there is always one clock cycle delay in the information passed on from signal **wishbone.slave.ack** to signal **ready**.

4. MSS Module implementation

4.1 General Information about ERC32 microprocessor

ERC32 is a radiation-tolerant 32-bit RISC processor developed for space applications. Two versions have been manufactured, the ERC32 Chip Set (Part Names: TSC691, TSC692, TSC693), and the ERC32 Single Chip (Part Name: TSC695). These implementations follow SPARC V7 specifications. The radiation-tolerant feature of ERC32 makes it very popular in space electronics applications, as it is one of the few microprocessors, which are available in radiation tolerant technology.[7]

ERC32 microprocessor has been developed under the ESA contract at the end of the 90's and it is now commercialised by Atmel. This microprocessor is implemented on the 0.5 micron Radiation Tolerant CMOS process of Atmel.

There were several design revisions during the development and evaluation phase of ERC32 microprocessor, as there have been found several deficiencies of the design. The final iteration of ERC32 that is nowadays commercialised by Atmel is the F iteration. Although some of the previous E iterations have been in field, it is strongly advised to discontinue their use.[8]

4.2 MSS Communication Interface Timing

In this subchapter, it will be described how the timing interface of ERC32 Microprocessor looks like. From this point on, the name MSS will be used instead of any other specific microprocessor's name. The Write and Read Timing of MSS are displayed below:

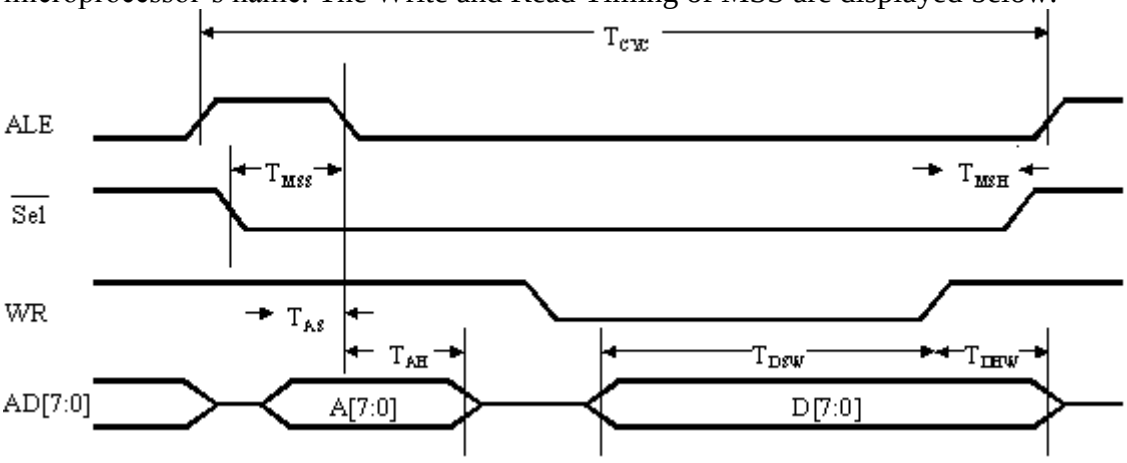


Figure 14 Write Timing of MSS Communication Interface

[7] http://en.wikipedia.org/wiki/ERC32

[8] http://microelectronics.esa.int/erc32/Hardware%20and%20Documentation%20Status%20of%20the%20ERC32%20Single%20Chip%20i1r1a.pdf

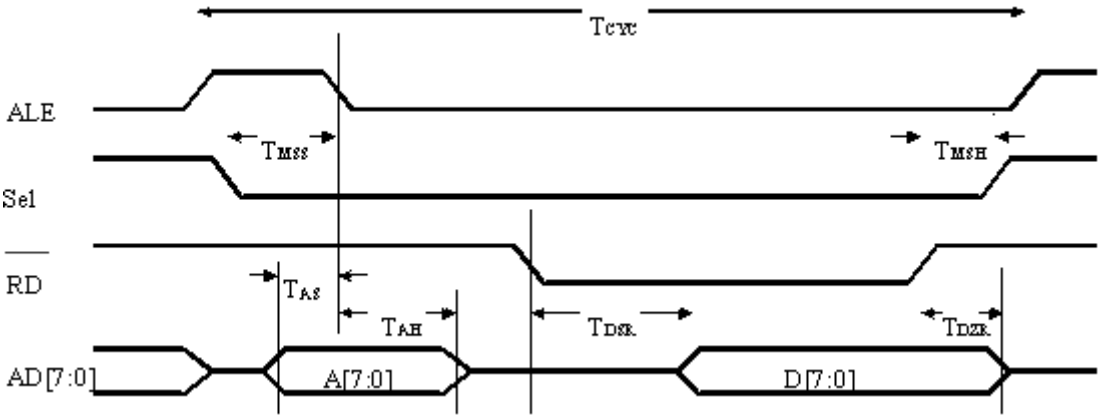


Figure 15 Read Timing of MSS Communication Interface

The timing constants definition follows in the next table:

Symbol	Parameter	Min number of clock cycles	Max number of clock cycles
Tcyc	access cycle time	4,0	-
Tmss	select active to ALE inactive time	0,5	-
Tmsh	access inactive to select inactive time	0,5	-
Tas	address to ALE low set up time	0,5	-
Tah	address hold time	0,5	-
Tdsw	write high data set up time	1,5	-
Tdhw	write high data hold time	0,5	-
Tdsr	read low to data valid time	-	1,0
Tdzr	read high data hold time	0	-
Twr	WR_N pulse length	-	1
Tale	ALE pulse length	-	1

Table 13 MSS timing constants

The control bus timing values are set, taking first into account that the MSS clock cycle duration is 66,7 ns = 15 Mhz frequency. It should be underlined that the precise adjustment of the timing parameters is beyond the scope of the research carried out. That means, that there will not be any change in the clock frequency of 50 Mhz which is used by the FPGA.

From Figure 15 and Figure 16 can be concluded, that the MSS Communication Interface consists of 4 "1-bit"control signals(ale, sel, wr, rd). It should be highlighted that the multiplexed address and data information has length **16 bits** and not 8 bits as it is shown in the two figures above. As a result, the whole MSS bus has 20 bits length.

4.3 Overview of MSS Module

Within this chapter the MSS Module is described. Having already given in subchapter 3.2.1 in Figure 4 an image of the WISHBONE architecture, it is clear that up to this point the WISHBONE structure consists of one master device and four slave devices. The MSS module

will be the fifth slave device which will be connected to the WISHBONE bus. As the MSS module is the fifth slave device in WISHBONE architecture, Table 12 must be updated with the new slave device addresses. The updated table follows in Table 14.

Slave Device	Start address	End address
SDRAM	0000_0000	007F_FFFF
FLASH	0400_0000	0401_FFFF
GPIO	0800_0000	0800_0005
TIMER	0C00_0000	0C00_0001
MSS	1000_0000	1000_0007

Table 14 Updated WISHBONE slave devices address field

The MSS slave device following the example of GPIO slave device can function either in Master or in Slave mode, meaning that it can send out or receive data packages. As the only way of communication is through the PIB ports, 20 of the 64 TTL IOs must be reserved for the MSS module. For testing purposes, inside the MSS module another device is developed, which is the DUT. Consequently, inside the MSS module there are two devices, the MSS and the DUT, which speak with each other through the PIB ports. Obviously, when the one device is functioning as master, the other functions as slave and the opposite. The GPIO module cannot speak anymore with all 8 ports installed on the PIB but only with 2 of them as the other 6 ports are reserved for the MSS module. Figure 16 shows the new structure of WISHBONE and what is more, it makes evident with which ports of the PIB, speaks each slave device.

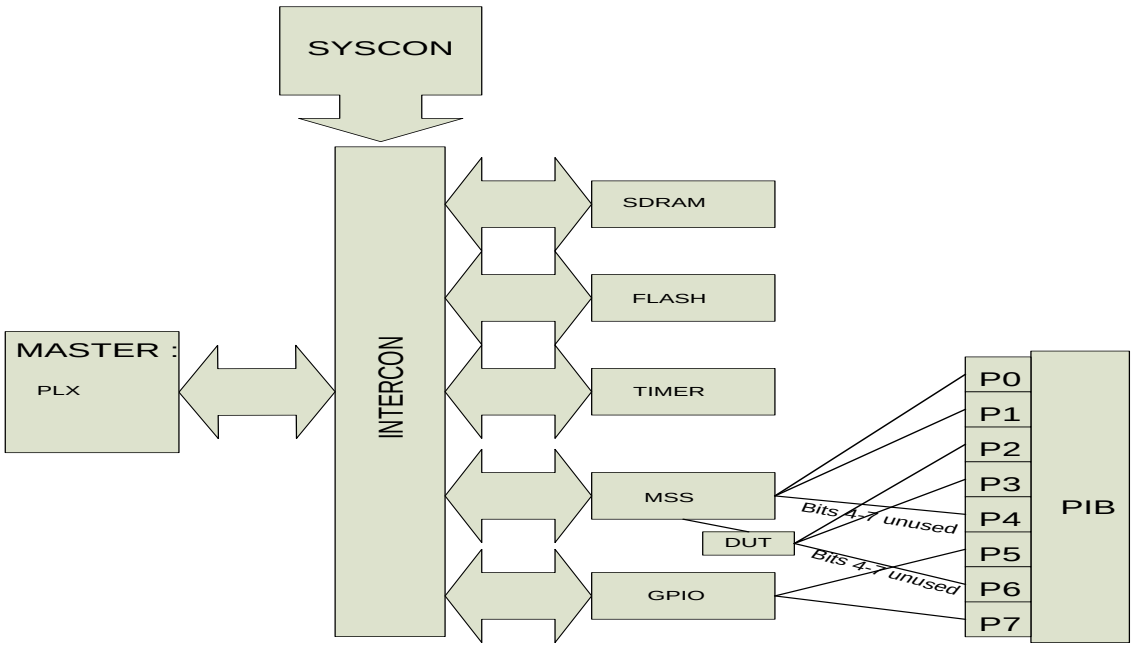


Figure 16 Updated WISHBONE Architecture

Bits 4-7 of Port 4 (P2) and Port 6 (P6) are unused as the MSS module has 20 bits length. Each mode of MSS module will be analyzed separately below.

MSS bus is defined by a vector signal called mss. Depending on the state of MSS module the mss signal is either an input (slave mode) or an output (master mode) signal. Its structure remains the same in both cases and is shown in Table 15.

Bits	19	18	17	16	15-0
Signal Name	Sel	Ale	Wr	Rd	Address/Data

Table 15 Mss vector bits definition

4.4 Development of MSS Module

4.4.1 MSS-Master

In this subchapter, these elements will be described, which the MSS slave device needs in order to operate in master mode. The MSS module will now simulate a real microprocessor and will generate its data bus, which is displayed in Figure 14 and Figure 15.

First of all, some registers will be used, in order to store the information which will be used, either for the stimulation of the control signals of MSS bus or as the address/data, which will be embedded into the MSS bus. Table 16 includes all registers which are used from the MSS module in master mode.

Name	Address	Description
Adr_o	1000_0000	Address on MSS bus
Data_ma_o	1000_0001	Data package on MSS bus, which is sent out (Write Command)
Data_ma_i	1000_0002	Received data package on MSS bus (Read Command)
Cmd_o	1000_0003	Command executed
Conf_o	1000_0004	Configuration information

Table 16 MSS module registers used in Master mode

Using the registers above, the information carried by local bus or generated inside MSS module can be transmitted to the PIB ports. The first three registers involve information which will be embedded in MSS bus. That means that a connection between the first three registers and the PIB ports has to be implemented so that the MSS bus information can reach the CON9 connector.

As the multiplexed address/data information on MSS bus has length 16 bits, the registers **Adr_o**, **Data_ma_o**, **Data_ma_i** are vectors of 16 bits. Register **Adr_o** contains the address to be included in MSS bus, register **Data_ma_o** contains the data package which will be included in MSS bus, in case a 'Write' command is executed. Register **Data_ma_i** contains the 16 bits information package, which is received either through connector CON9 or from MSS_Regs, in case a 'Read' command is executed. In the next subchapter register **Data_ma_i** will be discussed more analytically. The other two registers are used as arbitrators of the FSM, which implements the MSS timing. The first one (register **Cmd_o**) contains the command which is executed and the second one (register **Conf_o**) contains the configuration

which sets the duration of each stage of the command. Register **Cmd_o** has 8 bits length and can take specific values which determine, first of all, if the MSS module is in use or not, if it is in master or slave mode and last but not least, which command (Write/Read) will be executed. Table 17 explains the function of register **Cmd_o**. Register **Conf_o** has 24 bits length and its use can be understood looking at Figure 18 in subchapter 4.5.1. The 20 LSBs of **Conf_o** are used for the MSS timing, while the 4 MSBs have debugging use.

Bits	Use definition
0	If '0'/'1' MSS module executes read/write command.
1	Unused
2	Unused
3	Unused
4	Unused
5	If '0'/'1' MSS module is functioning in Master/Slave mode.
6	If '1' then MSS module is functioning
7	Unused

Table 17 Cmd_o register bits definition

Apart from the previous registers, which have a direct relationship with the MSS timing, another two registers are implemented which are used as debugging registers showing how many clock cycles a MSS command takes (Register **Counter_Cycles**) and how many orders have already been executed (Register **Counter_Orders**). These debugging registers can fuction **only** in master mode. A short description of these two registers follows in Table 18.

Name	Address	Description
Counter_Cycles	1000_0006	Number of clock cycles per order
Counter_Orders	1000_0007	Number of orders executed

Table 18 MSS module debugging registers

4.4.2 MSS-Slave

In slave mode, the only register from Table 16, which may be used is register **Data_ma_i.** As it is depicted in Figure 17, this register can receive data either from MSS-Master or from MSS-Slave. The source depends on the configuration register **Conf_o.** If the 4 MSBs of this register are set to '1', then register **Data_ma_i** receives a 16-bit data package from MSS-Slave. In any other case the source remains the MSS-Master. As already said, the register **Data_ma_i** receives data only in case of a 'Read' command. The option to receive data from MSS-Slave has been developed, as in this way it can be judged, if command 'Read' works properly, as up to this point there is no PCB connected to the PCI card, which would normally be the source of the data package.

The only registers, which are constructed especially for MSS-Slave are the **MSS_Regs**. These are a bank of 128 registers, where the data received from the PCB can be saved. The choice of the correct register relies on the bits 4 to 10 of WISHBONE address. Some features about this register bank follow in Table 19.

Name	Address	Description
PCB_Regs	1000_0005	Register Bank

Table 19 MSS module registers used in Slave mode

4.4.3 DUT

The registers, which are constructed especially for the DUT are the **PCB_Regs** and the **Data_Pcb_in**. The first one is a bank of 128 registers. In this register bank the data received from the MSS (master mode) can be saved. The second register stores the data package received from MSS-Slave in a read command. The choice of the correct member of the register bank relies on the bits 4 to 10 of WISHBONE address. Some features about these registers follow in Table 20.

Name	Address	Description
PCB_Regs	1000_0008	Register Bank for data received from MSS
Data_Pcb_in	1000_0009	Received data package from MSS (Read Command)

Table 20 MSS module registers used in DUT

4.5 Design of MSS Module

As the basic elements, created inside the MSS module, are already defined in the previous subsection, it is high time to present the design of MSS slave device, on which its VHDL implementation will be based afterwards. Figure 17 gives an outline of the design of MSS module.

The distinction between WISHBONE and MSS slave device is made clear in Figure 17. What is more, the function of all registers is shown, figuring out at the same time which register is related to which of the three devices inside the MSS slave device.

It should be stressed out, that one of the main differences between master and slave mode is the fact that slave mode is always in use in contrast to master mode. It was shown in Table 17, that master mode is active only if the third MSB of register **cmd_o** is set to ‘0’. As far as the slave mode is concerned, the same bit of register **cmd_o** defines from where the MSS-Slave receives the twenty MSS bus input bits. If the master mode is active (**cmd_o**(5)=’0’) the MSS bus generated from MSS-Master is sent not only to the PIB ports but also to the MSS-Slave. If the master mode is inactive (**cmd_o**(5)=’1’) the MSS bus input information to the MSS-Slave comes from the PIB ports.

The fact that the two devices inside the MSS slave device are correlated, gives to the user the opportunity to control the content of the register bank members, which belong to MSS-Slave, as he is able through MSS-Master to write a data package to a specific register, which will also be sent to the DUT. The connection of the MSS-Master with the DUT device is achieved over the PIB ports thanks to the use of a loop-back connector which creates a short-circuit between the ports 0-2,1-3,4-6 and 5-7 of the PIB.

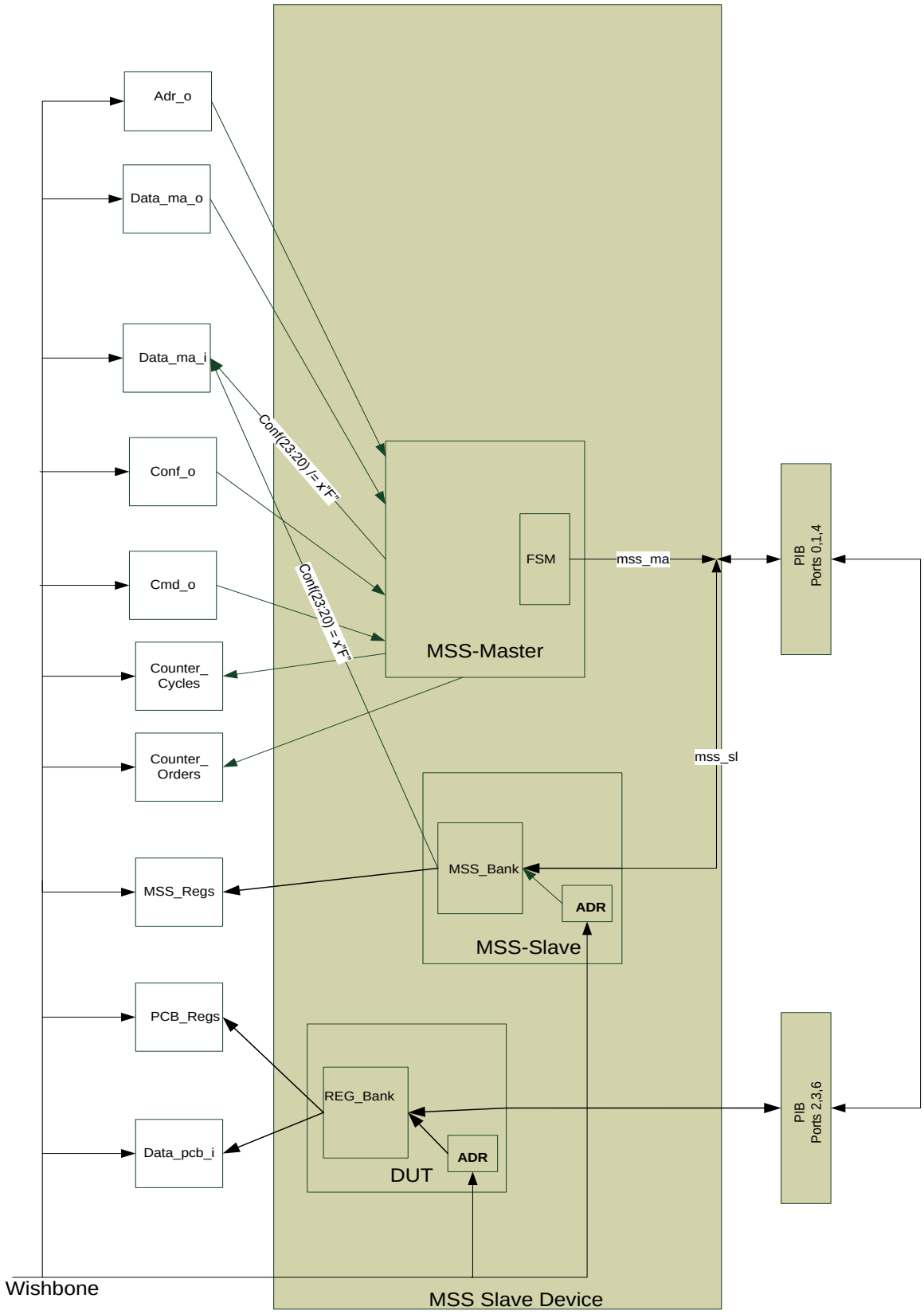


Figure 17 MSS Module Design

4.5.1 MSS-Master

Within this subchapter will be explained, how the MSS works in master mode. A design of MSS-Master will clarify its function.

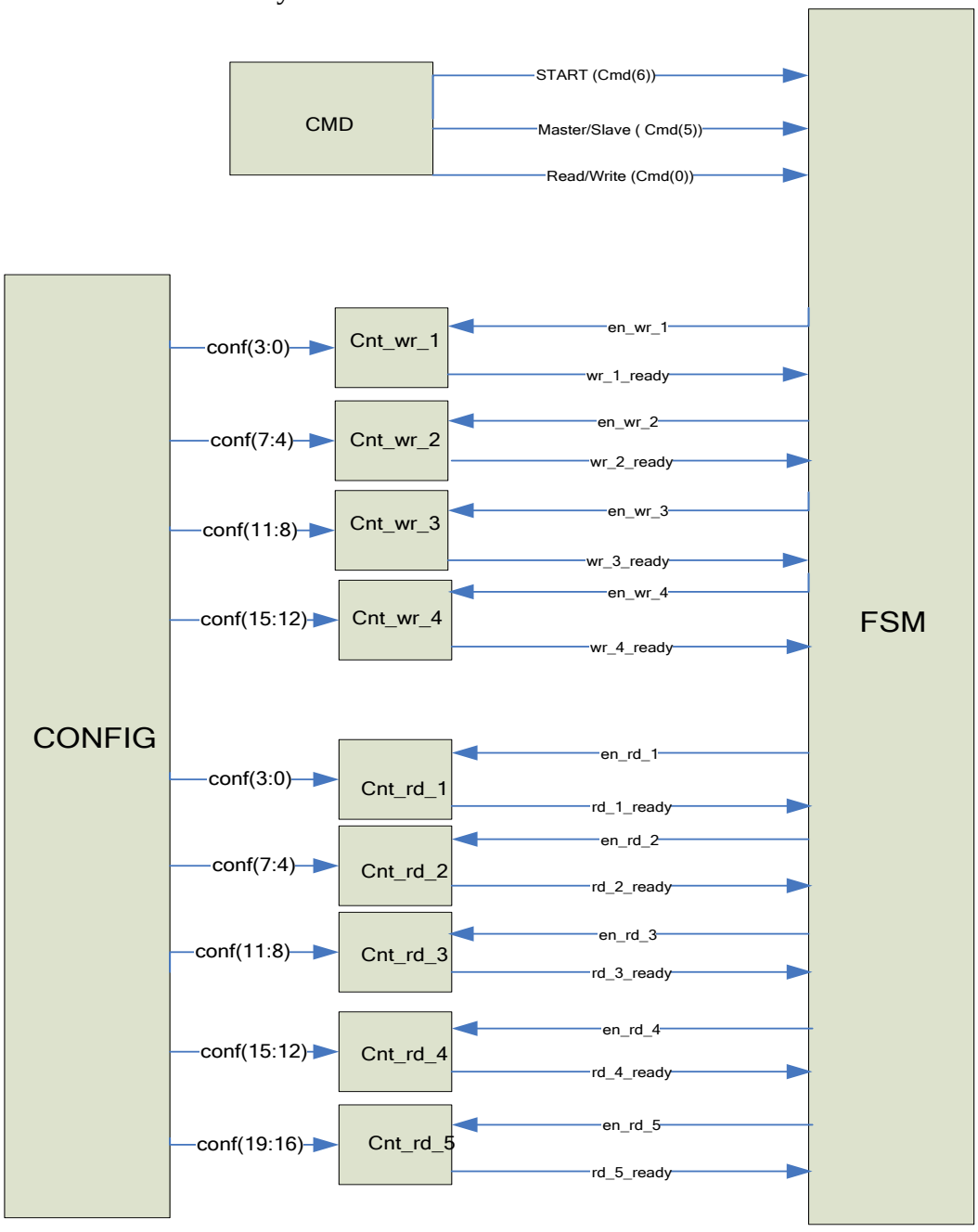


Figure 18 MSS-Master Design

Looking at the figure above, it is obvious, that register **Cmd_o** plays a crucial role in MSS module, as it is the register which determines the function of FSM (Finite State Machine) and defines the state of MSS module (Master/Slave).

Each of the two commands (Write/Read) is divided in a number of steps or, in other words, in a number of FSM states. In each state there is a change in the value of one or more bits of the

MSS bus. More specifically, ‘Write’ command consists of 4 states, while ‘Read’ command consists of 5 states. Each of these states and the function of FSM as well, are presented in the Figure 19.

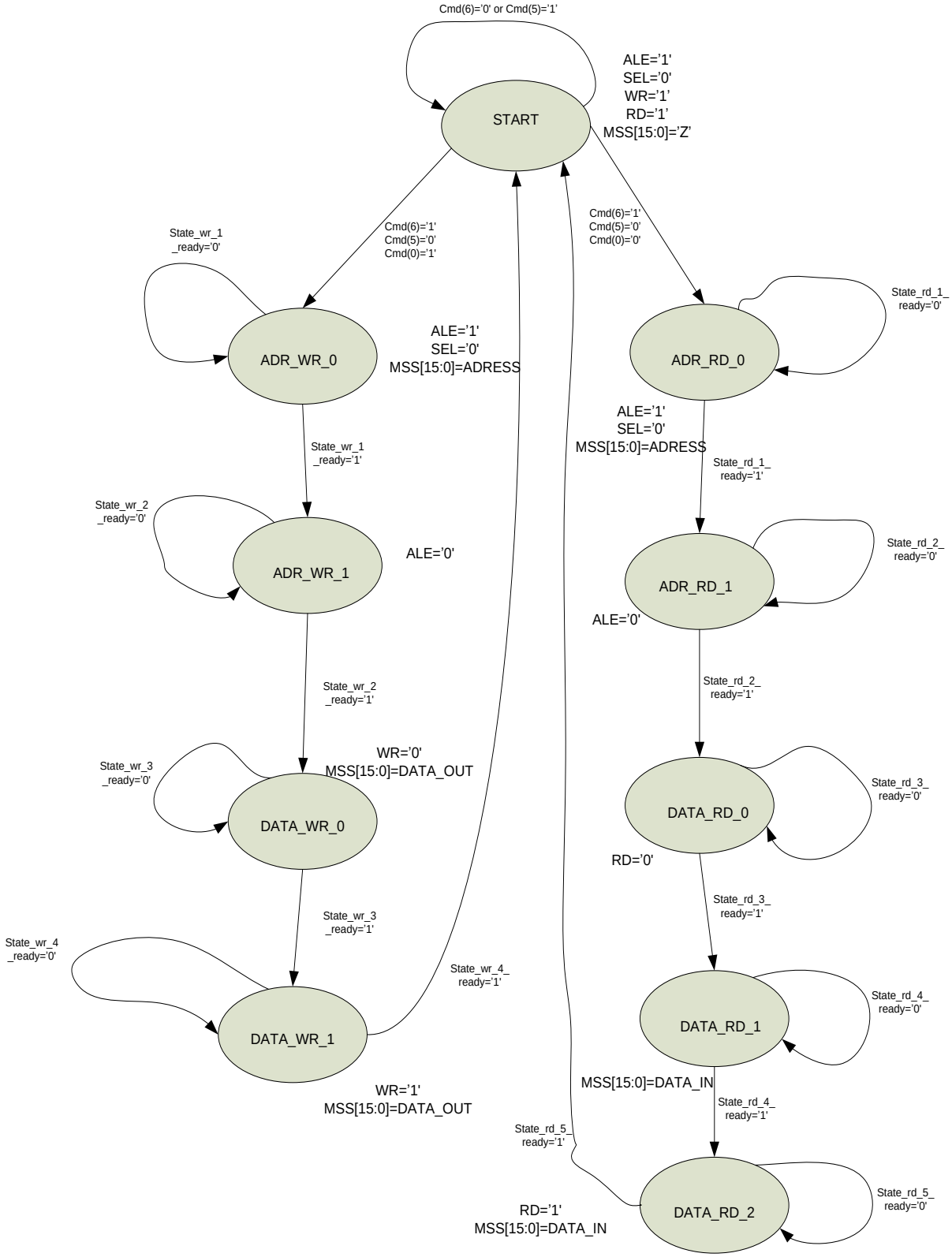


Figure 19 FSM Design

Looking at Figure 18, it is made clear how register **Conf_o** influences the MSS timing. Its 24 bits are divided in 6 parts of 4 bits each. The last 5 parts which belong to the 20 LSBs of **Conf_o** set the duration of a 'Read' command, while the last 4 parts which belong to the 16 LSBs of **Conf_o** set the duration of a 'Write' command. As it is already mentioned, the 4 MSBs of **Conf_o** define the source of the data package which will be stored in register **Data_ma_i.**

For each state of the FSM there is a separate counter defined, which is initialised from the corresponding 4 bits part of register **Conf_o.** Each counter signal is counting down till zero only if an enable signal is received from the FSM. Obviously, only one state of the FSM can be activated in every single clock cycle. When a counter is equal to zero, a ready signal is sent back to the FSM, informing the Finite State Machine that the corresponding state to which the ready signal belongs, has already lasted the exact number of clock cycles the **Conf_o** register defines.

In Figure 19, the signals-members of MSS bus, whose value is changed, are mentioned near each state of FSM. The FSM stays in state 'START' until the moment when the next command (Write/Read) comes. It should be highlighted, that the FSM functions only in case the MSS slave device works in master mode, or ,in other words, only if the third MSB of register **Cmd_o** is set to '0'.

4.5.2 MSS-Slave

MSS-Slave design is more simplistic in comparison with the MSS-Master design. In Slave mode, the MSS module receives a 20 bit vector signal which represents the Microprocessor bus. The responsibility of MSS-Slave is to translate the values of the control signals of MSS bus according to the diagrams which were presented in Figure 14 and Figure 15. By the correct translation of the control signals, the MSS-Slave is enabled to execute the corresponding operation these signals imply.

As stated earlier, MSS-Slave is always in use regardless of the value of register **Cmd_o.** The value of register **Cmd_o** regulates only the source from which the MSS-Slave receives the MSS bus bits.

Figure 21 shows a simplified image of MSS-Slave design.

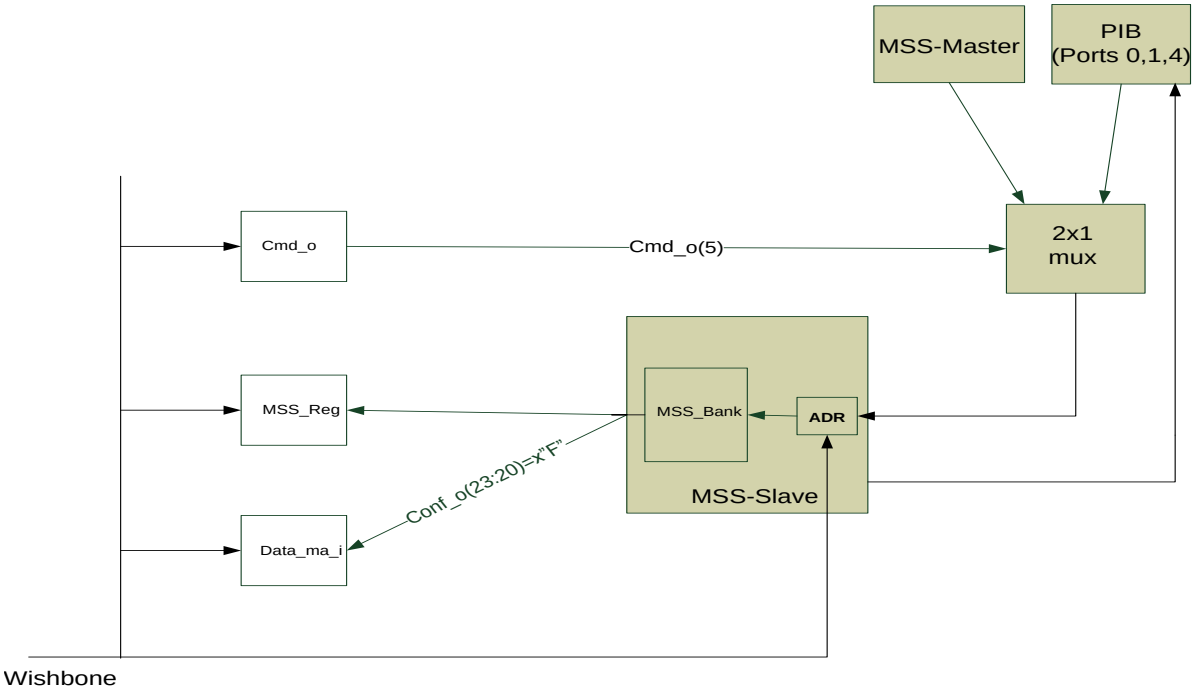


Figure 20 MSS-Slave Design

4.5.2 DUT

The device under test functions similar as the Master device as it generates the MSS timing which is sent towards the MSS master device. So the Figure 18 and Figure 19 can also describe the function of DUT.

Figure 21 shows a simplified image of DUT design.

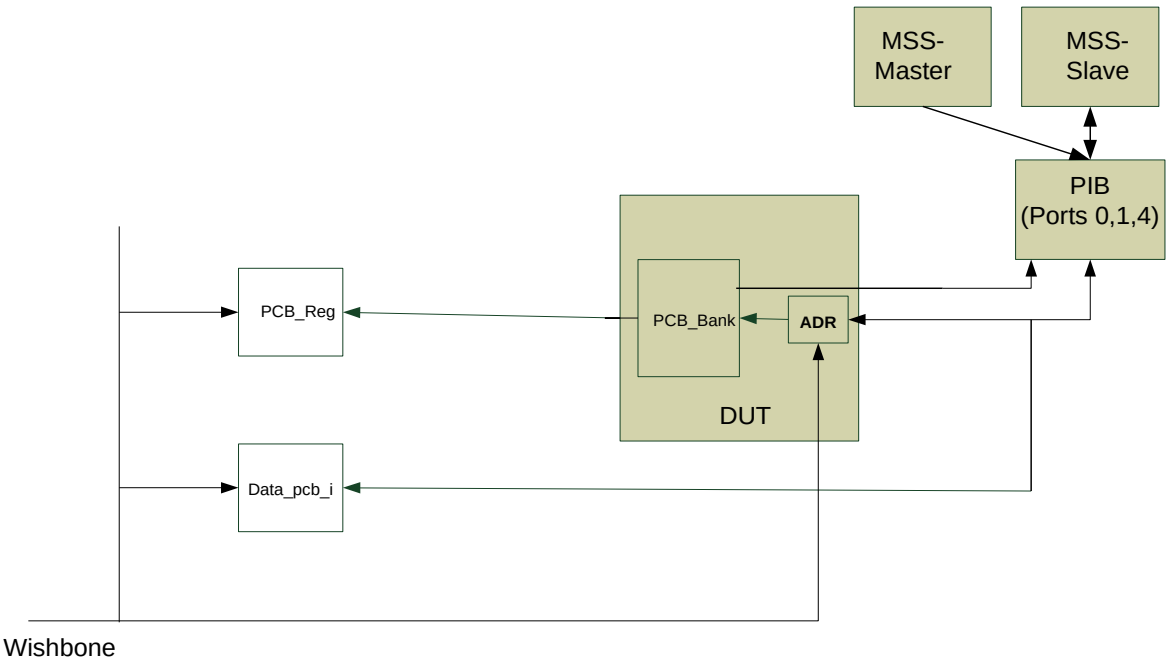


Figure 21 DUT Design

4.6 VHDL Implementation of MSS Module

In this subchapter the basic points of the VHDL implementation of MSS module will be analyzed. Apart from the new slave device's code, all changes inside the other modules of WISHBONE will be explained, so that the MSS slave device can fit in the WISHBONE architecture correctly.

4.6.1 Wishbone Module

First of all, starting from file "**wishbone.vhd**", all the changes inside WISHBONE module will be mentioned. As the code of file "**wishbone.vhd**" is already referred in 3.3.1, within this subchapter only the changes in the code will be underlined.

The changes inside the code of file "**wishbone.vhd**" follow below.

```
-- basic system configuration
  ----------------------------

  -- nr of devices
  constant nr_of_masters : positive := 1;
  subtype rng_masters is natural range (nr_of_masters-1) downto 0;
  constant nr_of_slaves : positive := 5;  --5 Slave Devices connected to the Wishbone Master Devices
  subtype rng_slaves is natural range (nr_of_slaves-1) downto 0;

  -- nr of address- and data-lines
  constant adr_width : positive := 32;
  subtype rng_adr is natural range (adr_width-1) downto 0;
  constant dat_width : positive := 32;
  subtype rng_dat is natural range (dat_width-1) downto 0;

  -- interrupt settings
  ---------------------

  -- nr of modules connected to the interrupt bus
  constant nr_of_irqdevs : positive := 6;
  subtype rng_irqdevs is natural range (nr_of_irqdevs-1) downto 0;

-- external GPIO, internal GPIO and LEDs
  ----------------------------------------
component wb_sl_gpio is
    generic
    (
      nr_of_irqiports : positive := 1;
      nr_of_irqoports : positive := 1;
      nr_of_dbgports : positive := 1
    );
    port
    (
      SYSCON_I : in rec_syscon_port;
      MASTER_I : in rec_master_port;
      SLAVE_O : out rec_slave_port;

      IRQ_I : in std_logic_vector((nr_of_irqiports-1) downto 0);
      IRQ_O : out std_logic_vector((nr_of_irqoports-1) downto 0);

      led_o : out std_logic_vector(7 downto 0);

      ext_gpio_oe_o : out std_logic_vector(7 downto 0);
      ext_gpio_input_i : in std_logic_vector(15 downto 0);
      ext_gpio_output_o : out std_logic_vector(15 downto 0);
```

```
    gpio_io : inout std_logic_vector(27 downto 0);

    debug : inout std_logic_vector((nr_of_dbgports-1) downto 0)
  );
end component;


signal ext_gpio_oe : std_logic_vector(7 downto 0) := (others => '0');
signal ext_gpio_input : std_logic_vector(15 downto 0) := (others => '0');
signal ext_gpio_output : std_logic_vector(15 downto 0) := (others => '0');


-- external MSS BUS
----------------------------------------
type reg128_bank_int is array(0 to 127) of std_logic_vector (15 downto 0);


constant reg128_bank_default :reg128_bank_int:=(others =>(others =>'0'));

-- define MSS base address
constant mss_baseadr : std_logic_vector(rng_adr) := x"1000_0000";
-- valid address bits
subtype rng_mss_adr is natural range 2 downto 0;
-- address_reg
constant mss_adress_offset : std_logic_vector(rng_adr) := x"0000_0000";
-- define bitmask for address
subtype rng_mss_adress is natural range 15 downto 0;
-- data_reg_out
constant mss_data_out_offset : std_logic_vector(rng_adr) := x"0000_0001";
-- data_reg_in
constant mss_data_in_offset : std_logic_vector(rng_adr) := x"0000_0002";
-- define bitmask for data
subtype rng_mss_data is natural range 15 downto 0;
-- cmd_reg
constant mss_cmd_offset : std_logic_vector(rng_adr) := x"0000_0003";
-- define bitmask for cmd
subtype rng_mss_cmd is natural range 7 downto 0;
-- conf_reg
constant mss_conf_offset : std_logic_vector(rng_adr) := x"0000_0004";
-- define bitmask for cmd
subtype rng_mss_conf is natural range 23 downto 0;
-- define Microprocessor Bus width
subtype rng_mss_length is natural range 19 downto 0;
  -- SLAVE MODE registers
constant mss_bank_offset : std_logic_vector(rng_adr) := x"0000_0005";
 -- PCB registers
constant pcb_bank_offset : std_logic_vector(rng_adr) := x"0000_0008";
-- data read from MSS-Slave
constant mss_data_slave_in_offset : std_logic_vector(rng_adr) := x"0000_0009";




-- pragma SW_ON
--#define MSS_BASEADR 0x40000000
--#define MSS_ADR_OFFSET 0x00000000
--#define MSS_ADRESS_MASK 0x0000FFFF
--#define MSS_DATA_OUT_OFFSET 0x00000004
--#define MSS_DATA_IN_OFFSET 0x00000008
--#define MSS_DATA_MASK 0x0000FFFF
--#define MSS_CMD_OFFSET 0x0000000C
--#define MSS_CMD_MASK 0x000000FF
--#define MSS_CONF_OFFSET 0x00000010
--#define MSS_CONF_MASK 0x00FFFFFF
--#define MSS_BANK_OFFSET 0x00000014
--#define MSS_CNT_CYCLE_OFFSET 0x00000018
--#define MSS_CNT_ORDERS_OFFSET 0x0000001C
--#define PCB_BANK_OFFSET 0x00000020
```

```
 --#define MSS_DATA_SLAVE_IN_OFFSET 0x00000024
 constant sl_mss_id : natural := TO_INTEGER(unsigned(mss_baseadr(rng_slave_select)));
 component wb_sl_mss is
  generic
  (
    nr_of_irqiports : positive := 1;
    nr_of_irqoports : positive := 1;
    nr_of_dbgports : positive := 1
  );
  port
  (
  SYSCON_I : in rec_syscon_port;
  MASTER_I : in rec_master_port;
  SLAVE_O  : out rec_slave_port;

  IRQ_I    : in std_logic_vector((nr_of_irqiports-1) downto 0);
  IRQ_O    : out std_logic_vector((nr_of_irqoports-1) downto 0);

  --MSS extra registers for storing info sent from C# Interface
   adr_o   : out std_logic_vector(15 downto 0);
  data_ma_o  : out std_logic_vector(15 downto 0);
  data_ma_i : in std_logic_vector(15 downto 0);
  cmd_o   : out std_logic_vector(7 downto 0);
  conf_o  : out std_logic_vector(23 downto 0);

  --Signals/Members of MSS bus as SLAVE Device
 sel_n_i : in std_logic;
 ale_i   : in std_logic;
 wr_n_i  : in std_logic;
 rd_n_i  : in std_logic;
 data_sl_o : out std_logic_vector(15 downto 0);

  --Signals/Members of MSS bus as MASTER Device
 sel_n_o : out std_logic;
 ale_o   : out std_logic;
 wr_n_o  : out std_logic;
 rd_n_o  : out std_logic;

 --Register Bank
 pcb_reg_i   : in reg128_bank_int;
 reg_o   : out reg128_bank_int;

--MSS bus
 mss_bus_i :in std_logic_vector(19 downto 0);
 mss_bus_o :out std_logic_vector(19 downto 0)
);
 end component;
 constant irqdev_mss_id : natural := 5;

-- external PCB BUS
 ----------------------------------------

 component pcb_test is

  port
  (
  SYSCON_I : in rec_syscon_port;
  --MSS Registers information is sent also to the DUT
  pcb_adr_i : in std_logic_vector(15 downto 0);
  pcb_data_ma_i : in  std_logic_vector(15 downto 0);
  pcb_cmd_i : in std_logic_vector(7 downto 0);
  pcb_conf_i : in std_logic_vector(23 downto 0);
  --PCB Regs are sent to MSS
  pcb_reg_o : out reg128_bank_int;
  --Data package read from PCB Regs
  data_read_o: out std_logic_vector(15 downto 0);
  --PCB Bus
  pcb_bus_i : in std_logic_vector(19 downto 0);
```

```
  pcb_bus_o : out std_logic_vector(19 downto 0)
  );

end component;
```

Code 9 WISHBONE.vhd with MSS

Inside the updated file "**wishbone.vhd**" the number of slave devices is now equal to five. The number of master devices doesn't change and remains equal to one. Another constant which is updated is the one, which defines the number of devices connected to the interrupt bus. Inside the code above there are two new constants. The first one (sl_mss_id) is the identity of mss slave device, so that the mss slave device can be either activated or inactivated. The second one (irqdev_mss_id) is the identity of mss slave device in the interrupt bus.

Furthermore, three signals in entity "**wb_sl_gpio**" have been renamed, in order to be able to set apart which module is speaking to the ports of the PIB. These three signals are the **"ext_oe", "ext_input"** and **"ext_output**", which receive respectively the names **"ext_gpio_oe", "ext_gpio_input"** and **"ext_gpio_output**". What is more, it is defined that the GPIO module speaks now to only 2 ports of the PIB. Two new entities are defined under the names "**wb_sl_mss**" and "**pcb_test**". The first one implements the MSS module. There are nine new constants defined, which represent the nine addresses related to the MSS module. Moreover, the data type **reg128_bank** represents the register bank which consists of 128 registers. The second new entity implements the DUT device.

4.6.2 Top Module

The source file "**pcis3base_top.vhd**" must also be updated in order to contain the instantiation of the MSS module.

The changes inside the code of file "**pcis3base_top.vhd**" follow below.

```
-- MSS address
signal adr                : std_logic_vector(15 downto 0) := (others => '0');
-- Data read from DUT
signal data_ma_in         : std_logic_vector(15 downto 0) := (others => '0');
-- Data sent to registers
signal data_ma_out        : std_logic_vector(15 downto 0) := (others => '0');
-- Data read from MSS-Slave registers
signal data_sl_out, data_read : std_logic_vector(15 downto 0) := (others => '0');
-- command register
signal cmd                : std_logic_vector(7 downto 0)  := (others => '0');
-- configuration register
signal conf               : std_logic_vector(23 downto 0) := (others => '0');
-- MSS and DUT bus
signal mss_ma, mss_sl     : std_logic_vector(19 downto 0) := (others => '0');
signal pcb_ma, pcb_sl     : std_logic_vector(19 downto 0) := (others => '0');
-- MSS Slave and PCB register banks
signal reg_bank, pcb_reg_bank : reg128_bank_int           := reg128_bank_default;
-- PIB ports directions
signal ext_mss_gpio_oe    : std_logic_vector(7 downto 0)  := (others => '0');
-- Communication of MSS-master and DUT device with the ports
signal ext_mss_input      : std_logic_vector(23 downto 0) := (others => '0');
signal ext_mss_output     : std_logic_vector(23 downto 0) := (others => '0');
signal ext_pcb_input      : std_logic_vector(23 downto 0);
signal ext_pcb_output     : std_logic_vector(23 downto 0) := (others => '0');
-- MSS bus control signals
signal wr_ma, wr_sl       : std_logic                     := '1';
signal rd_ma, rd_sl       : std_logic                     := '1';
```

```
 signal ale_ma, ale_sl        : std_logic                  := '0';
 signal sel_ma, sel_sl        : std_logic                  := '1';


inst_wb_sl_mss : wb_sl_mss
   generic map
   (
     nr_of_irqiports => nr_of_irqs,
     nr_of_irqoports => nr_of_irqs,
     nr_of_dbgports  => 1
     )
   port map
   (
     SYSCON_I => intercon.slaves.syscon(sl_mss_id),
     MASTER_I => intercon.slaves.master(sl_mss_id),
     SLAVE_O  => intercon.slaves.slave(sl_mss_id),

     IRQ_I => irqs,
     IRQ_O => irqos(irqdev_mss_id),

     adr_o     => adr,
     data_ma_o => data_ma_out,
     data_ma_i => data_ma_in,
     cmd_o     => cmd,
     conf_o    => conf,

     sel_n_i     => sel_sl,
     ale_i       => ale_sl,
     wr_n_i      => wr_sl,
     rd_n_i      => rd_sl,
     data_sl_o   => data_sl_out,
     data_read_i => data_read,

     sel_n_o => sel_ma,
     ale_o   => ale_ma,
     wr_n_o  => wr_ma,
     rd_n_o  => rd_ma,

     reg_o     => reg_bank,
     pcb_reg_i => pcb_reg_bank,

     mss_bus_o => mss_ma,
     mss_bus_i => mss_sl

     );

  inst_pcb_test : pcb_test
    port map(

     SYSCON_I => intercon.slaves.syscon(sl_mss_id),

     pcb_adr_i     => adr,
     pcb_data_ma_i => data_ma_out,
     pcb_cmd_i     => cmd,
     pcb_conf_i    => conf,

     pcb_reg_o   => pcb_reg_bank,
     data_read_o => data_read,

     pcb_bus_i => pcb_sl,
     pcb_bus_o => pcb_ma

     );

-- connect internal signals to the 100 pol. extension port
  ------------------------------------------------------------

  -- you will find further information about the signal assignment
```

```
-- in the the PIB64IO documentation. there is a table with the columns
-- "HDL Pin", "HDL Direction" and "Name", which should give you
-- an idea, how the connection of the entity port "pin_gpiomoduleport_io" to
-- the plugin-board is done

-- ports 0,1,4 => MSS-Master/MSS-Slave device module
-- ports 2,3,6 => DUT module
-- ports 5,7 => GPIO module

-- define each of the 93 I/Os as inputs, outputs or bidirs (default: all inputs)
gpiomoduleport.oe <=
  (
    1       => OUTPUT,
    2 to 5   => ext_mss_gpio_oe(7),
    6 to 9   => ext_mss_gpio_oe(6),
    10 to 11 => OUTPUT,
    12 to 15 => ext_mss_gpio_oe(5),
    16 to 19 => ext_mss_gpio_oe(4),
    20 to 21 => OUTPUT,
    22 to 25 => ext_mss_gpio_oe(3),
    26 to 29 => ext_mss_gpio_oe(2),
    30 to 31 => OUTPUT,
    32 to 35 => ext_mss_gpio_oe(0),
    36 to 39 => ext_mss_gpio_oe(1),
    40       => OUTPUT,
    50       => OUTPUT,
    51 to 54 => ext_mss_gpio_oe(1),
    55 to 58 => ext_mss_gpio_oe(0),
    59 to 60 => OUTPUT,
    61 to 64 => ext_mss_gpio_oe(2),
    65 to 68 => ext_mss_gpio_oe(3),
    69       => OUTPUT,
    71       => OUTPUT,
    72 to 75 => ext_mss_gpio_oe(4),
    76 to 79 => ext_mss_gpio_oe(5),
    80 to 81 => OUTPUT,
    82 to 85 => ext_mss_gpio_oe(6),
    86 to 88 => ext_mss_gpio_oe(7),
    90       => ext_mss_gpio_oe(7),
    91       => OUTPUT,
    others   => INPUT
    );

gpiomoduleport.output(1)  <= ext_mss_gpio_oe(7);  -- port 7 direction
gpiomoduleport.output(10) <= '0';
gpiomoduleport.output(11) <= ext_mss_gpio_oe(5);  -- port 5 direction
gpiomoduleport.output(20) <= '0';
gpiomoduleport.output(21) <= ext_mss_gpio_oe(3);  -- port 3 direction
gpiomoduleport.output(30) <= '0';
gpiomoduleport.output(31) <= ext_mss_gpio_oe(0);  -- port 0 direction
gpiomoduleport.output(40) <= '0';
gpiomoduleport.output(50) <= ext_mss_gpio_oe(1);  -- port 1 direction
gpiomoduleport.output(59) <= '0';
gpiomoduleport.output(60) <= ext_mss_gpio_oe(2);  -- port 2 direction
gpiomoduleport.output(69) <= '0';
gpiomoduleport.output(71) <= ext_mss_gpio_oe(4);  -- port 4 direction
gpiomoduleport.output(80) <= '0';
gpiomoduleport.output(81) <= ext_mss_gpio_oe(6);  -- port 6 direction
gpiomoduleport.output(91) <= '0';

gpiomoduleport.output(86) <= ext_gpio_output(1*8+7);    -- P7.7
gpiomoduleport.output(87) <= ext_gpio_output(1*8+6);    -- P7.6
gpiomoduleport.output(88) <= ext_gpio_output(1*8+5);    -- P7.5
gpiomoduleport.output(90) <= ext_gpio_output(1*8+4);    -- P7.4
ext_gpio_input(1*8+7)    <= gpiomoduleport.input(86);  -- P7.7
ext_gpio_input(1*8+6)    <= gpiomoduleport.input(87);  -- P7.6
ext_gpio_input(1*8+5)    <= gpiomoduleport.input(88);  -- P7.5
ext_gpio_input(1*8+4)    <= gpiomoduleport.input(90);  -- P7.4
```

```
--register data_ma_in receives a 16 bit vector signal either from the register bank or from PIB ports
-------------------------------------------------------------------
data_read_register : process(intercon.syscon)
-------------------------------------------------------------------
  begin
    if intercon.syscon.rst = '1' then
      data_ma_in <= (others => '0');
    elsif rising_edge(intercon.syscon.clk) then
      if rd_ma = '0' then
        if conf(23 downto 20) /= x"F" then
          data_ma_in <= ext_mss_input (15 downto 0);
        else

          data_ma_in <= reg_bank(conv_integer(adr(6 downto 0)));

        end if;
      end if;
    end if;
  end process data_read_register;

  -------------------------------------------------------------------
 MSS_Ports_DUT_communication : process(
                                       mss_ma,
                                         rd_ma,
                                         rd_sl,
                                         sel_sl,
                                         cmd,
                                         ext_mss_input,
                                         data_sl_out,
                                         ext_gpio_oe,
                                         conf,
                                         ext_pcb_input,
                                         pcb_ma)
  -------------------------------------------------------------------
  begin
  --Init
  ------------------------------------------------
    ext_mss_gpio_oe            <= "00010011";
    ext_mss_output(23 downto 0) <= x"0" & mss_ma;
    ext_pcb_output(23 downto 0) <= x"0" & pcb_ma;
    sel_sl                    <= ext_mss_input(19);
    ale_sl                    <= ext_mss_input(18);
    wr_sl                     <= ext_mss_input(17);
    rd_sl                     <= ext_mss_input(16);
    mss_sl(19 downto 0)        <= ext_mss_input (19 downto 0);
    pcb_sl(19 downto 0)        <= ext_pcb_input (19 downto 0);
  ------------------------------------------------

  --MSS speaks to the (0,1,4) PIB ports
  --DUT speaks to the (2,3,6) PIB ports
  --Start bit
  ---------------------------------------
    if cmd(6) = '1' then  -- Start Bit is set to '1' for the very first command sent.
  ---------------------------------------

  --Master Mode
  ---------------------------------------
      if cmd(5) = '0' then
        ext_mss_output(23 downto 0) <= x"0" & mss_ma;
        if cmd(0) = '1' then          -- Write
          ext_mss_gpio_oe <= "00010011";  -- Ports 0,1,4 are used to send the 20 MSS bits to the external 78-pin-HD-Sub
        else
          if rd_ma = '0' then
            if conf(23 downto 20) /= x"F" then  -- Read
              ext_mss_gpio_oe <= "00011100";  -- Ports 0,1 are used to receive the 16 Mss LSB bits from the external 78-pin-
HD-Sub and Port 4 to send the control signals
```

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

  ---------------------------------------

  --Slave Mode
  ---------------------------------------
    else
      ext_mss_gpio_oe <= "01001100";
      if rd_sl = '0' then
        ext_mss_gpio_oe            <= "00100011";
        ext_mss_output(15 downto 0) <= data_sl_out;
      end if;
    end if;

  ---------------------------------------

  --GPIO speaks to the (5 and 7) PIB ports
  ---------------------------------------
   else
    ext_mss_gpio_oe(7) <= ext_gpio_oe(7);
    ext_mss_gpio_oe(5) <= ext_gpio_oe(5);
   end if;
  ---------------------------------------
  end process;

-------------------------------------------------------------------------------
  CON9_connection :process(ext_mss_output, ext_gpio_output, ext_pcb_output, gpiomoduleport)
-------------------------------------------------------------------
  begin
   for i in 0 to 3 loop
    gpiomoduleport.output(2+i)  <= ext_gpio_output(1*8+i);     -- P7.0-3
    ext_gpio_input(1*8+i)       <= gpiomoduleport.input(2+i);
    gpiomoduleport.output(6+i)  <= ext_pcb_output(2*8+i);      -- P6.0-3
    ext_pcb_input(2*8+i)        <= gpiomoduleport.input(6+i);
    gpiomoduleport.output(12+i) <= ext_gpio_output(0*8+i);     -- P5.0-3
    ext_gpio_input(0*8+i)       <= gpiomoduleport.input(12+i);
    gpiomoduleport.output(16+i) <= ext_mss_output(2*8+i);      -- P4.0-3
    ext_mss_input(2*8+i)        <= gpiomoduleport.input(16+i);
    gpiomoduleport.output(22+i) <= ext_pcb_output(0*8+i);      -- P3.0-3
    ext_pcb_input(0*8+i)        <= gpiomoduleport.input(22+i);
    gpiomoduleport.output(26+i) <= ext_pcb_output(1*8+i);      -- P2.0-3
    ext_pcb_input(1*8+i)        <= gpiomoduleport.input(26+i);
    gpiomoduleport.output(36+i) <= ext_mss_output(1*8+i);      -- P1.0-3
    ext_mss_input(1*8+i)        <= gpiomoduleport.input(36+i);
    gpiomoduleport.output(32+i) <= ext_mss_output(0*8+i);      -- P0.0-3
    ext_mss_input(0*8+i)        <= gpiomoduleport.input(32+i);
    gpiomoduleport.output(58-i) <= ext_mss_output(0*8+(4+i));  -- P0.4-7
    ext_mss_input(0*8+(i+4))    <= gpiomoduleport.input(58-i);
    gpiomoduleport.output(54-i) <= ext_mss_output(1*8+(4+i));  -- P1.4-7
    ext_mss_input(1*8+(i+4))    <= gpiomoduleport.input(54-i);
    gpiomoduleport.output(64-i) <= ext_pcb_output(1*8+(4+i));  -- P2.4-7
    ext_pcb_input(1*8+(4+i))    <= gpiomoduleport.input(64-i);
    gpiomoduleport.output(68-i) <= ext_pcb_output(0*8+(4+i));  -- P3.4-7
    ext_pcb_input(0*8+(4+i))    <= gpiomoduleport.input(68-i);
    gpiomoduleport.output(75-i) <= ext_mss_output(2*8+(4+i));  -- P4.4-7
    ext_mss_input(23 downto 20) <= (others => '0');
    gpiomoduleport.output(79-i) <= ext_gpio_output(0*8+(4+i));  -- P5.4-7
    ext_gpio_input(0*8+(4+i))   <= gpiomoduleport.input(79-i);
    gpiomoduleport.output(85-i) <= ext_pcb_output(2*8+(4+i));  -- P6.4-7
    ext_pcb_input(23 downto 20) <= (others => '0');
   end loop;
  end process CON9_connection;
 end RTL;
```

Code 10 Pcis3base_top.vhd with MSS

First of all, as already mentioned in file "**wishbone.vhd**", there are some signals whose name should be changed, so that the communication of the two slave modules (**gpio,mss)** with the PIB ports can be better understood. Apart from the three signals written in the previous subchapter, there is another one signal, whose name is **"ext_oe**", which is now renamed to the name **"ext_mss_gpio_oe".**

Moreover, there are two new processes added inside the file "**pcis3base_top.vhd**", which implement the connection of the MSS module with the six PIB ports.

4.6.3 GPIO Module

The source file "**wb_sl_gpio.vhd**" must also be updated as the GPIO module speaks now with only 2 of the 8 ports. In other words, the connection of the GPIO slave device with the PIB ports must be updated.

The changes inside the code of file "**wb_sl_gpio.vhd**" follow below.

```
-- output register  port 5 and port 7 (external GPIO)
        when gpio_ext1_offset(rng_gpio_adr) =>
         ext_gpio_output(15 downto 0) <= wishbone.master.dat(15 downto 0);
  -- output register port 5 and port 7 (external GPIO)
     when gpio_ext1_offset(rng_gpio_adr) =>
       wishbone.slave.dat(15 downto 0) <= ext_gpio_input(INBUFF)(15 downto 0);
```

Code 11 Changes in Wb_sl_gpio.vhd

4.6.4 MSS Module

4.6.4.1 MSS-Master and MSS-Slave

Within this subchapter, the code of MSS-Master and MSS-Slave will be referred. The code of file "**wb_sl_mss.vhd**" follows below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

use work.wishbone.all;


entity wb_sl_mss is
  generic
    (
      nr_of_irqiports : positive := 1;
      nr_of_irqoports : positive := 1;
      nr_of_dbgports  : positive := 1
      );
  port
    (
      -- WISHBONE Interconnection signals
      SYSCON_I    : in  rec_syscon_port;
      MASTER_I    : in  rec_master_port;
      SLAVE_O     : out rec_slave_port;
      IRQ_I       : in  std_logic_vector((nr_of_irqiports-1) downto 0);
      IRQ_O       : out std_logic_vector((nr_of_irqoports-1) downto 0);
```

```
    -- MSS extra registers for storing info sent from C# Interface
    adr_o       : out std_logic_vector(15 downto 0);
    data_ma_o   : out std_logic_vector(15 downto 0);
    data_ma_i   : in  std_logic_vector(15 downto 0);
    cmd_o       : out std_logic_vector(7 downto 0);
    conf_o      : out std_logic_vector(23 downto 0);
    -- Signals/Members of MSS bus as SLAVE Device
    sel_n_i     : in  std_logic;
    ale_i       : in  std_logic;
    wr_n_i      : in  std_logic;
    rd_n_i      : in  std_logic;
    data_sl_o   : out std_logic_vector(15 downto 0);
    data_read_i : in  std_logic_vector(15 downto 0);
    -- Signals/Members of MSS bus as MASTER Device
    sel_n_o     : out std_logic;
    ale_o       : out std_logic;
    wr_n_o      : out std_logic;
    rd_n_o      : out std_logic;
    -- Register Bank
    reg_o       : out reg128_bank_int;
    pcb_reg_i   : in  reg128_bank_int;
    -- MSS bus
    mss_bus_i   : in  std_logic_vector(19 downto 0);
    mss_bus_o   : out std_logic_vector(19 downto 0)
    );
end wb_sl_mss;


architecture RTL of wb_sl_mss is

  type STATE_TYPE is (
    START,
    ADR_WR_0,
    ADR_WR_1,
    DATA_WR_0,
    DATA_WR_1,
    ADR_RD_0,
    ADR_RD_1,
    DATA_RD_0,
    DATA_RD_1,
    DATA_RD_2
    );

  -- Current and next state signals
  signal current_state : STATE_TYPE;
  signal next_state    : STATE_TYPE;

  constant INPUT                        : std_logic                             := '0';
  constant OUTPUT                       : std_logic                             := not INPUT;
  -- WISHBONE bus communication signal
  signal wishbone                       : rec_wishbone_signal                   := wishbone_default;
  -- Interrupt bus communication signals
  signal irqi                           : std_logic_vector((nr_of_irqiports-1) downto 0) := (others => '0');
  signal irqo                           : std_logic_vector((nr_of_irqoports-1) downto 0) := (others => '0');
  -- FSM enable and ready signals
  signal en_wr_1, en_wr_2               : std_logic                             := '0';
  signal en_wr_3, en_wr_4               : std_logic                             := '0';
  signal en_rd_1, en_rd_2               : std_logic                             := '0';
  signal en_rd_3, en_rd_4               : std_logic                             := '0';
  signal en_rd_5                        : std_logic                             := '0';
  signal state_wr_1_ready, state_wr_2_ready : std_logic                         := '0';
  signal state_wr_3_ready, state_wr_4_ready : std_logic                         := '0';
  signal state_rd_1_ready, state_rd_2_ready : std_logic                         := '0';
  signal state_rd_3_ready, state_rd_4_ready : std_logic                         := '0';
  signal state_rd_5_ready               : std_logic                             := '0';
  -- FSM control signal which shows if a whole command cycle is carried out
  signal done                           : std_logic                             := '0';
  -- MSS address register
```

```
  signal adr                          : std_logic_vector(15 downto 0)          := (others => '0');
  -- MSS-Master data registers
  signal data_out, data_in            : std_logic_vector(15 downto 0)          := (others => '0');
  -- MSS-Slave data register
  signal data_sl_out                  : std_logic_vector(15 downto 0)          := (others => '0');
  -- DUT data register
  signal data_read_in                 : std_logic_vector(15 downto 0)          := (others => '0');
  -- MSS-Slave register address
  signal adress_reg                   : std_logic_vector(15 downto 0);
  -- MSS-Slave register content
  signal data_reg                     : std_logic_vector(15 downto 0)          := (others => '0');
  -- MSS command register
  signal cmd                          : std_logic_vector(7 downto 0)           := (others => '0');
  -- FIFO implementing the write cycle in SLAVE mode
  signal wr_fifo                      : std_logic_vector(3 downto 0);
  -- MSS configuration register
  signal conf                         : std_logic_vector(23 downto 0)          := (others => '0');
  -- MSS debugging registers
  signal counter_orders, counter_cycles     : std_logic_vector(15 downto 0)    := (others => '0');
  -- MSS timing counter signals
  -- Read Counters
  signal state_rd_1_cnt, state_rd_2_cnt     : integer                          := 0;
  signal state_rd_3_cnt, state_rd_4_cnt     : integer                          := 0;
  signal state_rd_5_cnt               : integer                          := 0;
  -- Write Counters
  signal state_wr_1_cnt, state_wr_2_cnt     : integer                          := 0;
  signal state_wr_3_cnt, state_wr_4_cnt     : integer                          := 0;
  -- MSS Slave and DUT register banks
  signal reg_bank, pcb_reg_bank       : reg128_bank_int                  := reg128_bank_default;
  -- MSS Control Signals
  signal wr_in, rd_in, sel_in         : std_logic                        := '1';
  signal wr_out, rd_out, sel_out      : std_logic                        := '1';
  signal ale_in, ale_out              : std_logic                        := '0';
  -- MSS bus
  signal mss_in, mss_out              : std_logic_vector(19 downto 0)          := (others => 'Z');

begin

  -- connect ports to internal signals
  -------------------------------------

  -- ports of WISHBONE bus
  wishbone.syscon <= SYSCON_I;
  wishbone.master <= MASTER_I;
  SLAVE_O         <= wishbone.slave;

  -- ports of interrupt bus
  irqi  <= IRQ_I;
  IRQ_O <= irqo;

  --ADR/CMD Interface
  adr_o       <= adr;
  data_ma_o   <= data_out;
  data_in     <= data_ma_i;
  data_sl_o   <= data_sl_out;
  data_read_in <= data_read_i;
  cmd_o       <= cmd;
  conf_o      <= conf;

  -- MSS-Slave bus signals
  wr_in  <= wr_n_i;
  rd_in  <= rd_n_i;
  sel_in <= sel_n_i;
  ale_in <= ale_i;
  mss_in <= mss_bus_i;

  -- MSS-Master bus signals
  wr_n_o   <= wr_out;
```

```
  rd_n_o    <= rd_out;
  sel_n_o   <= sel_out;
  mss_bus_o <= mss_out;
  ale_o     <= ale_out;

  -- MSS-Slave and DUT register bank
  reg_o       <= reg_bank;
  pcb_reg_bank <= pcb_reg_i;

  -- The 4 Control Signals' values are passed to the MSS bus
  mss_out(19) <= sel_out;
  mss_out(18) <= ale_out;
  mss_out(17) <= wr_out;
  mss_out(16) <= rd_out;


--*******************************************************************--
-------------COMMUNICATION OF WISHBONE WITH MSS MODULE-----------------
--*******************************************************************--
-----------------------------------------------------------------------
  Wishbone_bus : process(
    counter_orders,
    counter_cycles,
    wishbone.syscon.rst,
    wishbone.syscon.clk,
    wishbone.master.adr,
    cmd,
    data_in,
    data_out,
    conf,
    adr,
    reg_bank,
    pcb_reg_bank,
    data_read_in)
-----------------------------------------------------------------------
  begin
    if wishbone.syscon.rst = '1' then
      wishbone.slave.ack <= '0';
    elsif wishbone.syscon.clk'event and wishbone.syscon.clk = '1' then
      -- WISHBONE handshake acknowledge response for registers
      if wishbone.slave.ack = '1' then
        wishbone.slave.ack <= '0';
      elsif (wishbone.master.cyc and wishbone.master.stb) = '1' then
        wishbone.slave.ack <= '1';
      end if;
      -- WISHBONE write address-decoding
      if (wishbone.master.cyc and wishbone.master.stb and wishbone.master.we) = '1' then
        cmd <= (others => '0');
        case wishbone.master.adr(rng_mss_adr) is
          -- MSS Address
          when mss_adress_offset(rng_mss_adr) =>
            adr <= wishbone.master.dat(rng_mss_adress);
            -- Data package sent towards the PIB ports
          when mss_data_out_offset(rng_mss_adr) =>
            data_out <= wishbone.master.dat(rng_mss_data);
            -- Command executed
          when mss_cmd_offset(rng_mss_adr) =>
            cmd <= wishbone.master.dat(rng_mss_cmd);
            -- Timing of the command cycle
          when mss_conf_offset(rng_mss_adr) =>
            conf <= wishbone.master.dat(rng_mss_conf);
          when others => null;
        end case;
      end if;
    end if;

-- combinatorial process part
----------------------------
```

```
  -- WISHBONE read address-decoding
  wishbone.slave.dat <= (others => '0');
  case wishbone.master.adr(rng_mss_adr) is
    -- Data package read in a read command cycle in Master mode
    when mss_data_in_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_data) <= data_in;
      -- MSS Address
    when mss_adress_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_adress) <= adr;
      -- Data package sent towards the PIB ports
    when mss_data_out_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_data) <= data_out;
      -- Command executed
    when mss_cmd_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_cmd) <= cmd;
      -- Timing of the command cycle
    when mss_conf_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_conf) <= conf;
      -- MSS-Slave Register Bank
    when mss_bank_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_data) <= reg_bank(conv_integer(wishbone.master.adr(10 downto 4)));
      -- Number of Cycles per order
    when mss_cnt_cycle_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_data) <= counter_cycles;
      -- Number of orders already executed
    when mss_cnt_orders_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_data) <= counter_orders;
      -- DUT Register Bank
    when pcb_bank_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_data) <= pcb_reg_bank(conv_integer(wishbone.master.adr(10 downto 4)));
      -- Data package read in a read command cycle in Slave mode
    when mss_data_slave_in_offset(rng_mss_adr) =>
      wishbone.slave.dat(rng_mss_data) <= data_read_in;
    when others => null;
  end case;
end process Wishbone_bus;
--**********************************************************************--
----------END OF COMMUNICATION OF WISHBONE WITH MSS MODULE---------
--**********************************************************************--

--**********************************************************************--
----------MSS AS SLAVE DEVICE----------
--**********************************************************************--

-----------------------------------------------------------------
adr_latch : process(wishbone.syscon.rst, ale_in)
-----------------------------------------------------------------
begin
  if wishbone.syscon.rst = '1' then
    adress_reg <= (others => '0');
  elsif falling_edge(ale_in) then
    adress_reg <= mss_in(15 downto 0);
  end if;
end process adr_latch;

-----------------------------------------------------------------
data_latch : process(wishbone.syscon.rst, wr_in)
-----------------------------------------------------------------
begin
  if wishbone.syscon.rst = '1' then
    data_reg <= (others => '0');
  elsif rising_edge(wr_in) then
    if sel_in = '0' then
      data_reg <= mss_in(15 downto 0);
    end if;
  end if;
end process data_latch;
```

```
-------------------------------------------------------------
read_mss_slave_reg_bank : process(adress_reg, reg_bank)
-------------------------------------------------------------
begin
  data_sl_out <= reg_bank(conv_integer(adress_reg(6 downto 0)));
end process read_mss_slave_reg_bank;

-------------------------------------------------------------
write_fifo_timing : process(wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if wishbone.syscon.rst = '1' then
    wr_fifo <= (others => '1');
  elsif rising_edge(wishbone.syscon.clk) then
    wr_fifo(2 downto 0) <= wr_fifo(3 downto 1);
    wr_fifo(3)          <= wr_in or sel_in;
  end if;
end process write_fifo_timing;

-------------------------------------------------------------
write_mss_slave_reg_bank : process(wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if wishbone.syscon.rst = '1' then
    reg_bank <= (others => (others => '0'));
  elsif rising_edge(wishbone.syscon.clk) then
    if wr_fifo(3 downto 2) = "10" then
      reg_bank(conv_integer(adress_reg(6 downto 0))) <= data_reg;
    end if;
  end if;
end process write_mss_slave_reg_bank;

--************************************************************************--
----------------------END OF MSS AS SLAVE DEVICE--------------------
--************************************************************************--


--************************************************************************--
-----------------------MSS AS MASTER DEVICE-------------------------
--************************************************************************--

-------------------------------------------------------------
fsm_en_start_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    done <= '1';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    if cmd = x"00" and done = '1' then
      done <= '0';
    else
      if state_wr_4_ready = '1' or state_rd_5_ready = '1' then
        done          <= '1';
        counter_orders <= counter_orders+1;
      end if;
    end if;
  end if;
end process fsm_en_start_clock;

-------------------------------------------------------------
fsm_en_wr_1_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_wr_1_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_wr_1_ready <= '0';
    if en_wr_1 = '1' then
```

```
      state_wr_1_cnt    <= state_wr_1_cnt-1;
      if state_wr_1_cnt <= 0 then
        state_wr_1_ready <= '1';
      end if;
    else
      state_wr_1_cnt <= conv_integer (conf(3 downto 0))-1;
    end if;
  end if;
end process fsm_en_wr_1_clock;

-------------------------------------------------------------------
fsm_en_wr_2_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_wr_2_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_wr_2_ready <= '0';
    if en_wr_2 = '1' then
      state_wr_2_cnt    <= state_wr_2_cnt-1;
      if state_wr_2_cnt <= 0 then
        state_wr_2_ready <= '1';
      end if;
    else
      state_wr_2_cnt <= conv_integer (conf(7 downto 4))-1;
    end if;
  end if;
end process fsm_en_wr_2_clock;

-------------------------------------------------------------------
fsm_en_wr_3_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_wr_3_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_wr_3_ready <= '0';
    if en_wr_3 = '1' then
      state_wr_3_cnt    <= state_wr_3_cnt-1;
      if state_wr_3_cnt <= 0 then
        state_wr_3_ready <= '1';
      end if;
    else
      state_wr_3_cnt <= conv_integer (conf(11 downto 8))-1;
    end if;
  end if;
end process fsm_en_wr_3_clock;

-------------------------------------------------------------------
fsm_en_wr_4_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_wr_4_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_wr_4_ready <= '0';
    if en_wr_4 = '1' then
      state_wr_4_cnt    <= state_wr_4_cnt-1;
      if state_wr_4_cnt <= 0 then
        state_wr_4_ready <= '1';
      end if;
    else
      state_wr_4_cnt <= conv_integer (conf(15 downto 12))-1;
    end if;
  end if;
end process fsm_en_wr_4_clock;

-------------------------------------------------------------------
```

```
fsm_en_rd_1_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_1_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_1_ready <= '0';
    if en_rd_1 = '1' then
      state_rd_1_cnt    <= state_rd_1_cnt-1;
      if state_rd_1_cnt <= 0 then
        state_rd_1_ready <= '1';
      end if;
    else
      state_rd_1_cnt <= conv_integer (conf(3 downto 0))-1;
    end if;
  end if;
end process fsm_en_rd_1_clock;


-------------------------------------------------------------------
fsm_en_rd_2_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_2_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_2_ready <= '0';
    if en_rd_2 = '1' then
      state_rd_2_cnt    <= state_rd_2_cnt-1;
      if state_rd_2_cnt <= 0 then
        state_rd_2_ready <= '1';
      end if;
    else
      state_rd_2_cnt <= conv_integer (conf(7 downto 4))-1;
    end if;
  end if;
end process fsm_en_rd_2_clock;


-------------------------------------------------------------------
fsm_en_rd_3_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_3_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_3_ready <= '0';
    if en_rd_3 = '1' then
      state_rd_3_cnt    <= state_rd_3_cnt-1;
      if state_rd_3_cnt <= 0 then
        state_rd_3_ready <= '1';
      end if;
    else
      state_rd_3_cnt <= conv_integer (conf(11 downto 8))-1;
    end if;
  end if;
end process fsm_en_rd_3_clock;


-------------------------------------------------------------------
fsm_en_rd_4_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_4_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_4_ready <= '0';
    if en_rd_4 = '1' then
      state_rd_4_cnt    <= state_rd_4_cnt-1;
      if state_rd_4_cnt <= 0 then
        state_rd_4_ready <= '1';
```

```
      end if;
    else
      state_rd_4_cnt <= conv_integer (conf(15 downto 12))-1;
    end if;
  end if;
end process fsm_en_rd_4_clock;

-------------------------------------------------------------------
fsm_en_rd_5_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_5_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_5_ready <= '0';
    if en_rd_5 = '1' then
      state_rd_5_cnt    <= state_rd_5_cnt-1;
      if state_rd_5_cnt <= 0 then
        state_rd_5_ready <= '1';
      end if;
    else
      state_rd_5_cnt <= conv_integer (conf(19 downto 16))-1;
    end if;
  end if;
end process fsm_en_rd_5_clock;

-------------------------------------------------------------
clocked_proc : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    current_state <= START;
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    if cmd = "00" then
      counter_cycles <= (others => '0');
    else
      if (sel_out = '0') then
        counter_cycles <= counter_cycles+1;
      end if;
    end if;
    current_state <= next_state;
  end if;
end process clocked_proc;

-------------------------------------------------------------
nextstate_proc : process (
  current_state,
  cmd,
  done,
  adr,
  data_out,
  data_in,
  state_wr_1_ready,
  state_wr_2_ready,
  state_wr_3_ready,
  state_wr_4_ready,
  state_rd_1_ready,
  state_rd_2_ready,
  state_rd_3_ready,
  state_rd_4_ready,
  state_rd_5_ready)
-------------------------------------------------------------
begin
  mss_out(15 downto 0) <= (others => 'Z');  -- The 15 LSB of Mss have multiplexed the adress and data
  ale_out              <= '0';       --ALE control signal
  sel_out              <= '1';       --SEL control signal
  wr_out               <= '1';       --WR  control signal
  rd_out               <= '1';       --RD  control signal
```

```
en_wr_1             <= '0';
en_wr_2             <= '0';
en_wr_3             <= '0';
en_wr_4             <= '0';
en_rd_1             <= '0';
en_rd_2             <= '0';
en_rd_3             <= '0';
en_rd_4             <= '0';
en_rd_5             <= '0';
next_state          <= START;
case current_state is
 when START =>
  if cmd(6) = '0' then
   next_state <= START;
  else
   --FSM functions only if MSS is in Master mode(cmd(5)='0')
   if cmd(5) = '0' then
    if done = '0' then
     if cmd(0) = '1' then
      --WRITE TIMING OF MSS COMMUNICATION INTEFACE
      next_state <= ADR_WR_0;
     else
      --READ TIMING OF MSS COMMUNICATION INTEFACE
      next_state <= ADR_RD_0;
     end if;
    else
     next_state <= START;
    end if;
   end if;
  end if;

  --START OF WRITE TIMING

  --Tmss=Tas

 when ADR_WR_0 =>
  if state_wr_1_ready = '1' then
   en_wr_2             <= '1';
   -------------------
   mss_out(15 downto 0) <= adr;
   sel_out             <= '0';
   -------------------
   next_state          <= ADR_WR_1;
  else
   en_wr_1             <= '1';
   -------
   ale_out             <= '1';
   mss_out(15 downto 0) <= adr;
   sel_out             <= '0';
   -------------------
   next_state          <= ADR_WR_0;
  end if;

  --Tah

 when ADR_WR_1 =>
  if state_wr_2_ready = '1' then
   en_wr_3             <= '1';
   -------
   wr_out              <= '0';
   mss_out(15 downto 0) <= data_out;
   -------------------
   sel_out             <= '0';
   -------------------
   next_state          <= DATA_WR_0;
  else
   en_wr_2             <= '1';
   -------------------
```

```
    sel_out             <= '0';
    mss_out(15 downto 0) <= adr;
    ------------------
    next_state          <= ADR_WR_1;
  end if;

  --Tdsw

when DATA_WR_0 =>
  if state_wr_3_ready = '1' then
    en_wr_4             <= '1';
    ------------------
    mss_out(15 downto 0) <= data_out;
    sel_out             <= '0';
    ------------------
    next_state          <= DATA_WR_1;
  else
    en_wr_3             <= '1';
    -------
    wr_out              <= '0';
    mss_out(15 downto 0) <= data_out;
    ------------------
    sel_out             <= '0';
    ------------------
    next_state          <= DATA_WR_0;
  end if;

  --Tmsh=Tdhw

when DATA_WR_1 =>
  if state_wr_4_ready = '1' then
    next_state <= START;
  else
    en_wr_4             <= '1';
    ------------------
    sel_out             <= '0';
    mss_out(15 downto 0) <= data_out;
    ------------------
    next_state          <= DATA_WR_1;
  end if;


  --START OF READ TIMING

  --Tmss=Tas

when ADR_RD_0 =>
  if state_rd_1_ready = '1' then
    en_rd_2             <= '1';
    ------------------
    ale_out             <= '0';
    sel_out             <= '0';
    mss_out(15 downto 0) <= adr;
    ------------------
    next_state          <= ADR_RD_1;
  else
    en_rd_1             <= '1';
    -------
    ale_out             <= '1';
    sel_out             <= '0';
    mss_out(15 downto 0) <= adr;
    ------------------
    next_state          <= ADR_RD_0;
  end if;

  --Tah

when ADR_RD_1 =>
```

```
  if state_rd_2_ready = '1' then
    en_rd_3    <= '1';
    -------
    rd_out     <= '0';
    -------------------
    sel_out    <= '0';
    -------------------
    next_state <= DATA_RD_0;
  else
    en_rd_2             <= '1';
    -------------------
    sel_out             <= '0';
    mss_out(15 downto 0) <= adr;
    -------------------
    next_state          <= ADR_RD_1;
  end if;

  --Tdsr

when DATA_RD_0 =>
  if state_rd_3_ready = '1' then
    en_rd_4             <= '1';
    -------
    mss_out(15 downto 0) <= data_in;
    -------------------
    rd_out              <= '0';
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_RD_1;
  else
    en_rd_3    <= '1';
    -------
    rd_out     <= '0';
    -------------------
    sel_out    <= '0';
    -------------------
    next_state <= DATA_RD_0;
  end if;

  --Tx

when DATA_RD_1 =>
  if state_rd_4_ready = '1' then
    en_rd_5             <= '1';
    -------------------
    mss_out(15 downto 0) <= data_in;
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_RD_2;
  else
    en_rd_4             <= '1';
    -------
    mss_out(15 downto 0) <= data_in;
    -------------------
    rd_out              <= '0';
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_RD_1;
  end if;

  --Tmsh=Tdzr

when DATA_RD_2 =>
  if state_rd_5_ready = '1' then
    next_state <= START;
  else
    en_rd_5             <= '1';
    -------------------
```

```
        mss_out(15 downto 0) <= data_in;
        sel_out              <= '0';
        -------------------
        next_state           <= DATA_RD_2;
      end if;
    end case;
  end process nextstate_proc;
--*****************************************************************--
-------------------------END OF MSS AS MASTER DEVICE-------------------
--*****************************************************************--

end RTL;
```

Code 12 Wb_sl_mss.vhd

Code 12 is divided in three main parts. In the first one, the communication of WISHBONE with the registers of the new module is implemented. In the second one the MSS-Slave device is developed, while in third one the MSS-Master device is implemented. Each of the last two parts is further divided into other processes each of which implements a specific function.

4.6.4.2 DUT

Within this subchapter, the code of DUT device will be referred. The code of file "**pcb_test.vhd**" follows below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

use work.wishbone.all;


entity pcb_test is
  port
    (
      -- Clock and reset signal
      SYSCON_I      : in  rec_syscon_port;
      -- Information from MSS registers
      pcb_adr_i     : in  std_logic_vector(15 downto 0);
      pcb_data_ma_i : in  std_logic_vector(15 downto 0);
      pcb_cmd_i     : in  std_logic_vector(7 downto 0);
      pcb_conf_i    : in  std_logic_vector(23 downto 0);
      -- PCB Register bank
      pcb_reg_o     : out reg128_bank_int;
      -- Data received from MSS Slave
      data_read_o   : out std_logic_vector(15 downto 0);
      -- PCB Bus
      pcb_bus_i     : in  std_logic_vector(19 downto 0);
      pcb_bus_o     : out std_logic_vector(19 downto 0)
      );
end pcb_test;

architecture RTL of pcb_test is

  type STATE_TYPE is (
    START,
    ADR_WR_0,
    ADR_WR_1,
    DATA_WR_0,
    DATA_WR_1,
```

```
  ADR_RD_0,
  ADR_RD_1,
  DATA_RD_0,
  DATA_RD_1,
  DATA_RD_2
  );

 -- Current and next state signals
 signal current_state : STATE_TYPE;
 signal next_state    : STATE_TYPE;

 constant INPUT                      : std_logic                 := '0';
 constant OUTPUT                     : std_logic                 := not INPUT;
 -- WISHBONE bus communication signal
 signal wishbone                     : rec_wishbone_signal       := wishbone_default;
 -- FSM enable and ready signals
 signal en_wr_1, en_wr_2             : std_logic                 := '0';
 signal en_wr_3, en_wr_4             : std_logic                 := '0';
 signal en_rd_1, en_rd_2             : std_logic                 := '0';
 signal en_rd_3, en_rd_4             : std_logic                 := '0';
 signal en_rd_5                      : std_logic                 := '0';
 signal state_wr_1_ready, state_wr_2_ready : std_logic           := '0';
 signal state_wr_3_ready, state_wr_4_ready : std_logic           := '0';
 signal state_rd_1_ready, state_rd_2_ready : std_logic           := '0';
 signal state_rd_3_ready, state_rd_4_ready : std_logic           := '0';
 signal state_rd_5_ready             : std_logic                 := '0';
 -- FSM control signal which shows if a whole command cycle is carried out
 signal done                         : std_logic                 := '0';
 -- DUT address register
 signal adr                          : std_logic_vector(15 downto 0) := (others => '0');
 -- DUT data package
 signal data_pcb_out, data_sl_out    : std_logic_vector(15 downto 0) := (others => '0');
 -- DUT register address
 signal adress_reg, pcb_adr_in       : std_logic_vector(15 downto 0);
 -- DUT register data
 signal data_reg, pcb_data_in        : std_logic_vector(15 downto 0);
 -- Data package read from DUT
 signal data_read_out                : std_logic_vector(15 downto 0);
 -- Command carried out
 signal cmd                          : std_logic_vector(7 downto 0)  := (others => '0');
 -- FIFO implementing the write cycle in SLAVE mode
 signal wr_fifo                      : std_logic_vector(3 downto 0);
 -- DUT configuration register
 signal conf                         : std_logic_vector(23 downto 0) := (others => '0');
 -- DUT timing counter signals
 -- Read Counters
 signal state_rd_1_cnt, state_rd_2_cnt  : integer                 := 0;
 signal state_rd_3_cnt, state_rd_4_cnt  : integer                 := 0;
 signal state_rd_5_cnt               : integer                   := 0;
 -- Write Counters
 signal state_wr_1_cnt, state_wr_2_cnt  : integer                 := 0;
 signal state_wr_3_cnt, state_wr_4_cnt  : integer                 := 0;
 -- DUT register bank
 signal pcb_reg_bank                 : reg128_bank_int           := reg128_bank_default;
 -- DUT Control Signals
 signal wr_in, rd_in, sel_in         : std_logic                 := '1';
 signal wr_out, rd_out, sel_out      : std_logic                 := '1';
 signal ale_in, ale_out              : std_logic                 := '0';
 -- DUT bus
 signal pcb_bus_in, pcb_bus_out      : std_logic_vector(19 downto 0) := (others => 'Z');


begin
 -- connect ports to internal signals
 -------------------------------------
 -- ports of WISHBONE bus
 wishbone.syscon <= SYSCON_I;
 -- MSS registers
```

```
 pcb_adr_in      <= pcb_adr_i;
 pcb_data_in     <= pcb_data_ma_i;
 cmd             <= pcb_cmd_i;
 conf            <= pcb_conf_i;
 -- Data package received from DUT device
 data_read_o     <= data_read_out;
 -- DUT bus
 pcb_bus_in      <= pcb_bus_i;
 pcb_bus_o       <= pcb_bus_out;
 -- DUT address bank
 pcb_reg_o       <= pcb_reg_bank;
 -- The 4 MSBs values are passed to the DUT control signals
 sel_in          <= pcb_bus_in(19);
 ale_in          <= pcb_bus_in(18);
 wr_in           <= pcb_bus_in(17);
 rd_in           <= pcb_bus_in(16);
 -- The 4 Control Signals' values are passed to the DUT bus
 pcb_bus_out(19) <= sel_out;
 pcb_bus_out(18) <= ale_out;
 pcb_bus_out(17) <= wr_out;
 pcb_bus_out(16) <= rd_out;

--*********************************************************************--
----------COMMUNICATION OF DUT DEVICE WITH REGISTERS----------------
--*********************************************************************--
-------------------------------------------------------------------
 DUT_adr_data_reg : process (cmd, adress_reg, pcb_bus_in, pcb_adr_in, data_sl_out)
-------------------------------------------------------------------
 begin
   if cmd(5) = '0' then
     adr          <= adress_reg;
     data_pcb_out <= data_sl_out;
   else
     adr          <= pcb_adr_in;
     data_pcb_out <= pcb_bus_in(15 downto 0);
   end if;
 end process DUT_adr_data_reg;
--*********************************************************************--
------------------------DUT AS SLAVE DEVICE-------------------------
--*********************************************************************--

-------------------------------------------------------------------
 adr_latch : process(wishbone.syscon.rst, ale_in)
-------------------------------------------------------------------
 begin
   if wishbone.syscon.rst = '1' then
     adress_reg <= (others => '0');
   elsif falling_edge(ale_in) then
     adress_reg <= pcb_bus_in (15 downto 0);
   end if;
 end process adr_latch;

-------------------------------------------------------------------
 data_latch : process(wishbone.syscon.rst, wr_in)
-------------------------------------------------------------------
 begin
   if wishbone.syscon.rst = '1' then
     data_reg <= (others => '0');
   elsif rising_edge(wr_in) then
     if sel_in = '0' then
       data_reg <=pcb_bus_in (15 downto 0);
     end if;
   end if;
 end process data_latch;

-------------------------------------------------------------------
 read_mss_slave_reg_bank : process(adress_reg, reg_bank)
-------------------------------------------------------------------
```

```
begin
  data_sl_out <= pcb_reg_bank(conv_integer(adress_reg(6 downto 0)));
end process read_mss_slave_reg_bank;

-------------------------------------------------------------
write_fifo_timing : process(wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if wishbone.syscon.rst = '1' then
    wr_fifo <= (others => '1');
  elsif rising_edge(wishbone.syscon.clk) then
    wr_fifo(2 downto 0) <= wr_fifo(3 downto 1);
    wr_fifo(3)          <= wr_in or sel_in;
  end if;
end process write_fifo_timing;

-------------------------------------------------------------
write_dut_slave_reg_bank : process(wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if wishbone.syscon.rst = '1' then
    reg_bank <= (others => (others => '0'));
  elsif rising_edge(wishbone.syscon.clk) then
   if cmd(5)='0' then
    if wr_fifo(3 downto 2) = "10" then
     pcb_ reg_bank(conv_integer(adress_reg(6 downto 0))) <= data_reg;
    end if;
   end if;
  end if;
end process write_mss_slave_reg_bank;

--*****************************************************************--
-----------------------END OF DUT AS SLAVE DEVICE-------------------
--*****************************************************************--


--*****************************************************************--
-----------------------DUT AS MASTER DEVICE-------------------------
--*****************************************************************--

-------------------------------------------------------------
fsm_en_start_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    done <= '1';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    if cmd = x"00" and done = '1' then
      done <= '0';
    else
      if state_wr_4_ready = '1' or state_rd_5_ready = '1' then
        done          <= '1';
        counter_orders <= counter_orders+1;
      end if;
    end if;
  end if;
end process fsm_en_start_clock;

-------------------------------------------------------------
fsm_en_wr_1_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_wr_1_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_wr_1_ready <= '0';
    if en_wr_1 = '1' then
      state_wr_1_cnt    <= state_wr_1_cnt-1;
```

```
      if state_wr_1_cnt <= 0 then
        state_wr_1_ready <= '1';
      end if;
    else
      state_wr_1_cnt <= conv_integer (conf(3 downto 0))-1;
    end if;
  end if;
end process fsm_en_wr_1_clock;

-------------------------------------------------------------------
fsm_en_wr_2_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_wr_2_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_wr_2_ready <= '0';
    if en_wr_2 = '1' then
      state_wr_2_cnt    <= state_wr_2_cnt-1;
      if state_wr_2_cnt <= 0 then
        state_wr_2_ready <= '1';
      end if;
    else
      state_wr_2_cnt <= conv_integer (conf(7 downto 4))-1;
    end if;
  end if;
end process fsm_en_wr_2_clock;

-------------------------------------------------------------------
fsm_en_wr_3_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_wr_3_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_wr_3_ready <= '0';
    if en_wr_3 = '1' then
      state_wr_3_cnt    <= state_wr_3_cnt-1;
      if state_wr_3_cnt <= 0 then
        state_wr_3_ready <= '1';
      end if;
    else
      state_wr_3_cnt <= conv_integer (conf(11 downto 8))-1;
    end if;
  end if;
end process fsm_en_wr_3_clock;

-------------------------------------------------------------------
fsm_en_wr_4_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_wr_4_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_wr_4_ready <= '0';
    if en_wr_4 = '1' then
      state_wr_4_cnt    <= state_wr_4_cnt-1;
      if state_wr_4_cnt <= 0 then
        state_wr_4_ready <= '1';
      end if;
    else
      state_wr_4_cnt <= conv_integer (conf(15 downto 12))-1;
    end if;
  end if;
end process fsm_en_wr_4_clock;

-------------------------------------------------------------------
fsm_en_rd_1_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
```

```
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_1_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_1_ready <= '0';
    if en_rd_1 = '1' then
      state_rd_1_cnt    <= state_rd_1_cnt-1;
      if state_rd_1_cnt <= 0 then
        state_rd_1_ready <= '1';
      end if;
    else
      state_rd_1_cnt <= conv_integer (conf(3 downto 0))-1;
    end if;
  end if;
end process fsm_en_rd_1_clock;

-------------------------------------------------------------------
fsm_en_rd_2_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_2_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_2_ready <= '0';
    if en_rd_2 = '1' then
      state_rd_2_cnt    <= state_rd_2_cnt-1;
      if state_rd_2_cnt <= 0 then
        state_rd_2_ready <= '1';
      end if;
    else
      state_rd_2_cnt <= conv_integer (conf(7 downto 4))-1;
    end if;
  end if;
end process fsm_en_rd_2_clock;

-------------------------------------------------------------------
fsm_en_rd_3_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_3_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_3_ready <= '0';
    if en_rd_3 = '1' then
      state_rd_3_cnt    <= state_rd_3_cnt-1;
      if state_rd_3_cnt <= 0 then
        state_rd_3_ready <= '1';
      end if;
    else
      state_rd_3_cnt <= conv_integer (conf(11 downto 8))-1;
    end if;
  end if;
end process fsm_en_rd_3_clock;

-------------------------------------------------------------------
fsm_en_rd_4_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_4_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_4_ready <= '0';
    if en_rd_4 = '1' then
      state_rd_4_cnt    <= state_rd_4_cnt-1;
      if state_rd_4_cnt <= 0 then
        state_rd_4_ready <= '1';
      end if;
```

```
    else
      state_rd_4_cnt <= conv_integer (conf(15 downto 12))-1;
    end if;
  end if;
end process fsm_en_rd_4_clock;

-------------------------------------------------------------------
fsm_en_rd_5_clock : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    state_rd_5_ready <= '0';
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    state_rd_5_ready <= '0';
    if en_rd_5 = '1' then
      state_rd_5_cnt    <= state_rd_5_cnt-1;
      if state_rd_5_cnt <= 0 then
        state_rd_5_ready <= '1';
      end if;
    else
      state_rd_5_cnt <= conv_integer (conf(19 downto 16))-1;
    end if;
  end if;
end process fsm_en_rd_5_clock;

-------------------------------------------------------------
clocked_proc : process (wishbone.syscon.rst, wishbone.syscon.clk)
-------------------------------------------------------------
begin
  if (wishbone.syscon.rst = '1') then
    current_state <= START;
  elsif (wishbone.syscon.clk'event and wishbone.syscon.clk = '1') then
    if cmd = "00" then
      counter_cycles <= (others => '0');
    else
      if (sel_out = '0') then
        counter_cycles <= counter_cycles+1;
      end if;
    end if;
    current_state <= next_state;
  end if;
end process clocked_proc;

-------------------------------------------------------------
nextstate_proc : process (
  current_state,
  cmd,
  done,
  adr,
  pcb_data_in,
  data_pcb_out,
  state_wr_1_ready,
  state_wr_2_ready,
  state_wr_3_ready,
  state_wr_4_ready,
  state_rd_1_ready,
  state_rd_2_ready,
  state_rd_3_ready,
  state_rd_4_ready,
  state_rd_5_ready)
  ---------------------------------------------------------------------------
begin
  pcb_bus_out (15 downto 0) <= (others => 'Z');  -- The 15 LSB of Mss have multiplexed the adress and data
  ale_out             <= '0';       --ALE control signal
  sel_out             <= '1';       --SEL control signal
  wr_out               <= '1';       --WR  control signal
  rd_out              <= '1';       --RD  control signal
  en_wr_1               <= '0';
```

```
en_wr_2             <= '0';
en_wr_3             <= '0';
en_wr_4             <= '0';
en_rd_1             <= '0';
en_rd_2             <= '0';
en_rd_3             <= '0';
en_rd_4             <= '0';
en_rd_5             <= '0';
next_state          <= START;
case current_state is
   when START =>
   if cmd(6) = '0' then
     next_state <= START;
   else
     if cmd(5) = '0' then
       if done = '0' then
         if cmd(0) = '0' then
           -- READ TIMING OF DUT COMMUNICATION INTEFACE
           next_state <= ADR_RD_0;
         end if;
       else
         next_state <= START;
       end if;
     else
       if done = '0' then
         if cmd(0) = '1' then
           -- WRITE TIMING OF DUT COMMUNICATION INTEFACE
           next_state <= ADR_WR_0;
         else
           -- READ TIMING OF DUT COMMUNICATION INTEFACE
           next_state <= ADR_RD_0;
         end if;
       else
         next_state <= START;
       end if;
     end if;
   end if;
   --START OF WRITE TIMING

   --Tmss=Tas

  when ADR_WR_0 =>
    if state_wr_1_ready = '1' then
      en_wr_2             <= '1';
      -------------------
      pcb_bus_out(15 downto 0) <= adr;
      sel_out             <= '0';
      -------------------
      next_state          <= ADR_WR_1;
    else
      en_wr_1             <= '1';
      -------
      ale_out             <= '1';
      pcb_bus _out(15 downto 0) <= adr;
      sel_out             <= '0';
      -------------------
      next_state          <= ADR_WR_0;
    end if;

    --Tah

  when ADR_WR_1 =>
    if state_wr_2_ready = '1' then
      en_wr_3             <= '1';
      -------
      wr_out              <= '0';
      pcb_bus _out(15 downto 0) <= pcb_data_int;
      -------------------
```

```
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_WR_0;
   else
    en_wr_2             <= '1';
    -------------------
    sel_out             <= '0';
    mss_out(15 downto 0) <= adr;
    -------------------
    next_state          <= ADR_WR_1;
   end if;

   --Tdsw

  when DATA_WR_0 =>
   if state_wr_3_ready = '1' then
    en_wr_4             <= '1';
    -------------------
    pcb_bus _out(15 downto 0) <= pcb_data_in;
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_WR_1;
   else
    en_wr_3             <= '1';
    -------
    wr_out              <= '0';
    pcb_bus _out(15 downto 0) <= pcb_data_in;
    -------------------
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_WR_0;
   end if;

   --Tmsh=Tdhw

  when DATA_WR_1 =>
   if state_wr_4_ready = '1' then
    next_state <= START;
   else
    en_wr_4             <= '1';
    -------------------
    sel_out             <= '0';
    pcb_bus _out(15 downto 0) <= pcb_data_in;
    -------------------
    next_state          <= DATA_WR_1;
   end if;


   --START OF READ TIMING

   --Tmss=Tas

  when ADR_RD_0 =>
   if state_rd_1_ready = '1' then
    en_rd_2             <= '1';
    -------------------
    ale_out             <= '0';
    sel_out             <= '0';
    pcb_bus _out(15 downto 0) <= adr;
    -------------------
    next_state          <= ADR_RD_1;
   else
    en_rd_1             <= '1';
    -------
    ale_out             <= '1';
    sel_out             <= '0';
    pcb_bus _out(15 downto 0) <= adr;
    -------------------
```

```
    next_state          <= ADR_RD_0;
  end if;

  --Tah

when ADR_RD_1 =>
  if state_rd_2_ready = '1' then
    en_rd_3    <= '1';
    -------
    rd_out     <= '0';
    -------------------
    sel_out    <= '0';
    -------------------
    next_state <= DATA_RD_0;
  else
    en_rd_2             <= '1';
    -------------------
    sel_out             <= '0';
    pcb_bus _out(15 downto 0) <= adr;
    -------------------
    next_state          <= ADR_RD_1;
  end if;

  --Tdsr

when DATA_RD_0 =>
  if state_rd_3_ready = '1' then
    en_rd_4             <= '1';
    -------
    pcb_bus _out(15 downto 0) <= data_pcb_out;
    -------------------
    rd_out              <= '0';
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_RD_1;
  else
    en_rd_3    <= '1';
    -------
    rd_out     <= '0';
    -------------------
    sel_out    <= '0';
    -------------------
    next_state <= DATA_RD_0;
  end if;

  --Tx

when DATA_RD_1 =>
  if state_rd_4_ready = '1' then
    en_rd_5             <= '1';
    -------------------
    pcb_bus _out(15 downto 0) <= data_pcb_out;
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_RD_2;
  else
    en_rd_4             <= '1';
    -------
    pcb_bus _out(15 downto 0) <= data_pcb_out;
    -------------------
    rd_out              <= '0';
    sel_out             <= '0';
    -------------------
    next_state          <= DATA_RD_1;
  end if;

  --Tmsh=Tdzr
```

```
    when DATA_RD_2 =>
      if state_rd_5_ready = '1' then
        next_state <= START;
      else
        en_rd_5             <= '1';
        -------------------
        pcb_bus _out(15 downto 0) <= data_pcb_out;
        sel_out             <= '0';
        -------------------
        next_state          <= DATA_RD_2;
      end if;
   end case;
 end process nextstate_proc;
--*************************************************************************--
-------------------------END OF MSS AS MASTER DEVICE--------------------
--*************************************************************************--

end RTL;
```

Code 13 Pcb_test.vhd

It is obvious, that the code inside the file of "**pcb_test.vhd**" is similar to the code which implements the MSS-Master and MSS-Slave device. The main difference is that the FSM inside the DUT can function either the DUT is in master (MSS is in slave mode) or in slave mode.

4.7 Testing MSS Module with Modelsim

Having already explained, how the MSS module is developed, the next step in order to clarify how the MSS slave device works, is to use the testbench already referred in Code 6, which simulates the local bus. In such a way it can be understood how the MSS module works either in master or in slave mode.

First of all, the master mode will be explained. The testbench will read the following text file, which includes assembly commands.

```
Master
wr   40000000  0000FF7F
wr   40000004  0000FFF0
wr   40000010  00062222
wr   4000000C  00000041
wr   40000000  00000F71
wr   40000004  0000FFFF
wr   4000000C  00000041
wr   40000000  00000F7F
wr   4000000C  00000040
Slave
wr   40000000  0000FF7E
wr   40000004  0000FFF0
wr   40000010  00062222
wr   4000000C  00000061
wr   40000000  0000FF7D
wr   40000004  0000FFF2
wr   40000010  00062222
wr   4000000C  00000061
```

Code 14 MSS Commandfile.txt

Taking a look inside the code of the WISHBONE package defined in Code 9 it can be understood that the addresses written inside the text file are the addresses which correspond to the MSS module. It has to be mentioned that the addresses in "MSS Commandife.txt" are the local bus addresses which will be translated to the corresponding WISHBONE bus addresses inside the master device connected to the WISHBONE bus. For instance, the local bus address 40000004 is equal to the WISHBONE bus address 10000001 as in general, the local bus address is equal to the WISHBONE bus address when the latter is left shifted two times.

It is time to present the simulation, which will give an overview of the WISHBONE-MSS relationship.

Step 1

```
wr   40000000  0000FF7F
wr   40000004  0000FFF0
wr   40000010  00062222
wr   4000000C  00000041
```

The testbench reads the first command group and passes on its information to register 127 of MSS-Slave and DUT register bank.

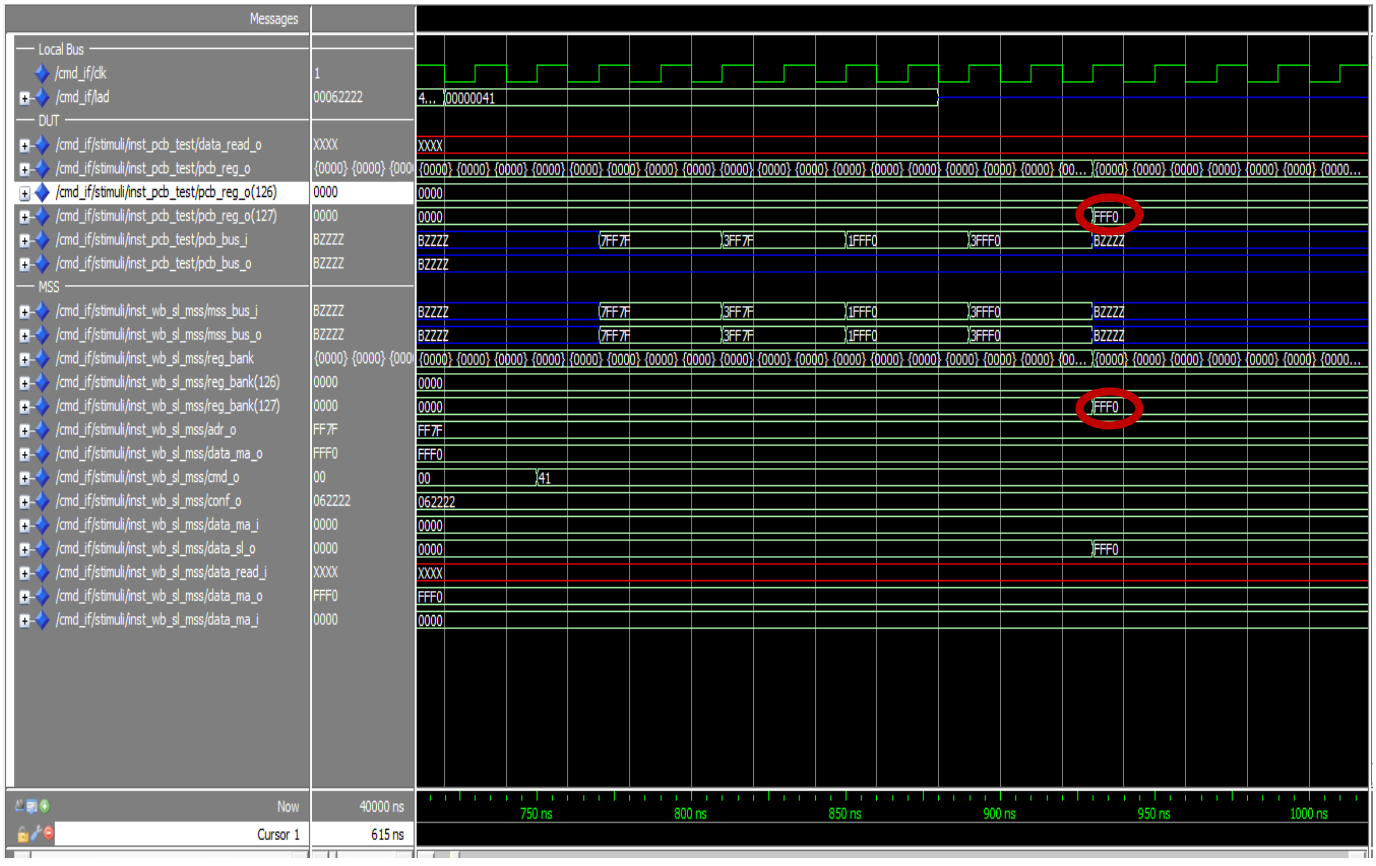


Figure 22 Wishbone-MSS Testbench pic.1

In the figure above it is shown a write command in master mode. That means that the address 7F not only of the DUT register bank but also of the MSS-Slave register bank will store the data package FFF0.

Step 2

```
wr   40000000  00000F7F
wr   4000000C  00000040
```

The testbench reads the second command group and reads the address 7F of DUT register bank.

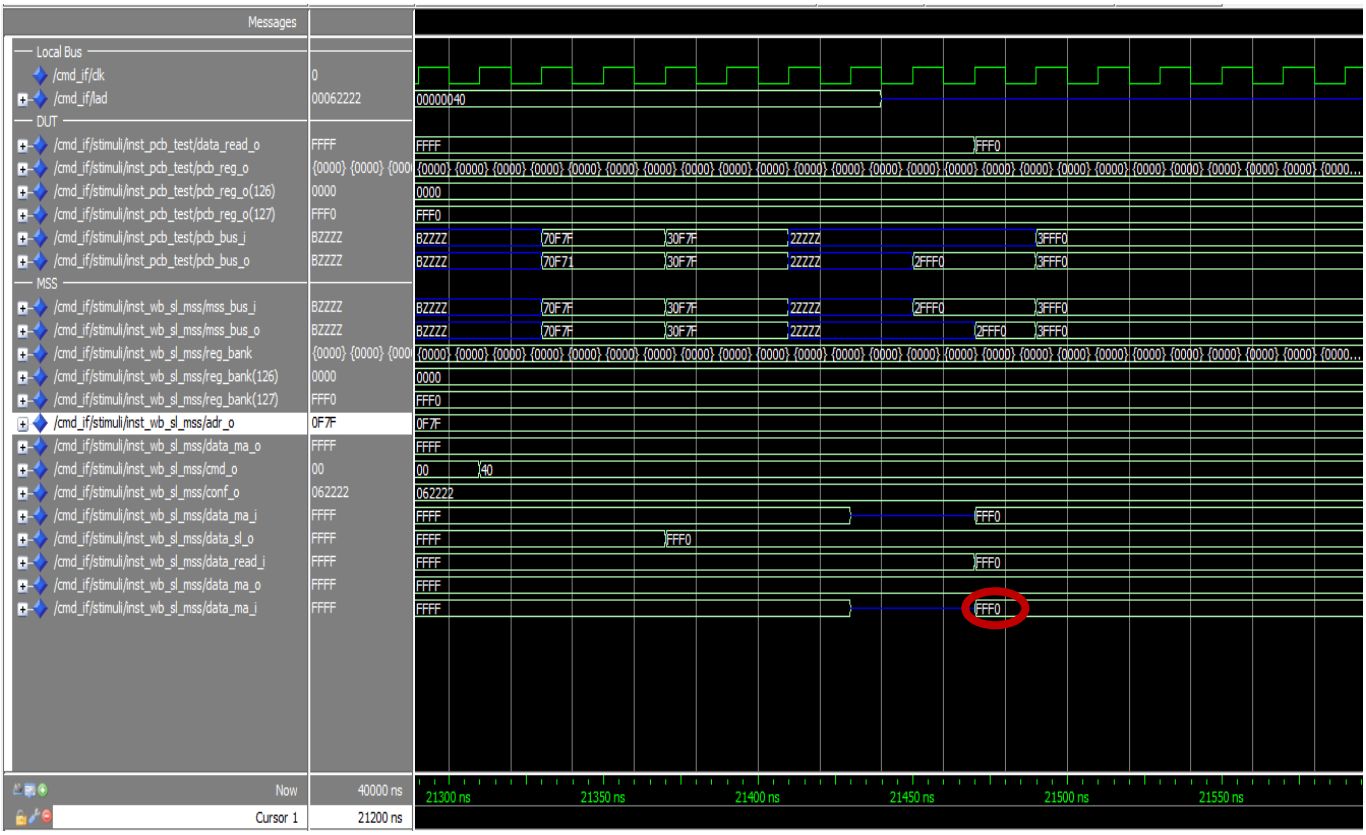


Figure 23 Wishbone-MSS Testbench pic.2

In the figure above it is shown a read command in master mode. Inside the red cycle is shown that the register **data_ma_i** receives the data package that was read during the execution of the read command cycle.

Step 3

```
wr   40000000  0000FF7E
wr   40000004  0000FFF0
wr   40000010  00062222
wr   4000000C  00000061
```

The testbench reads the third group of commands and passes on its information to register 126 of MSS-Slave register bank.

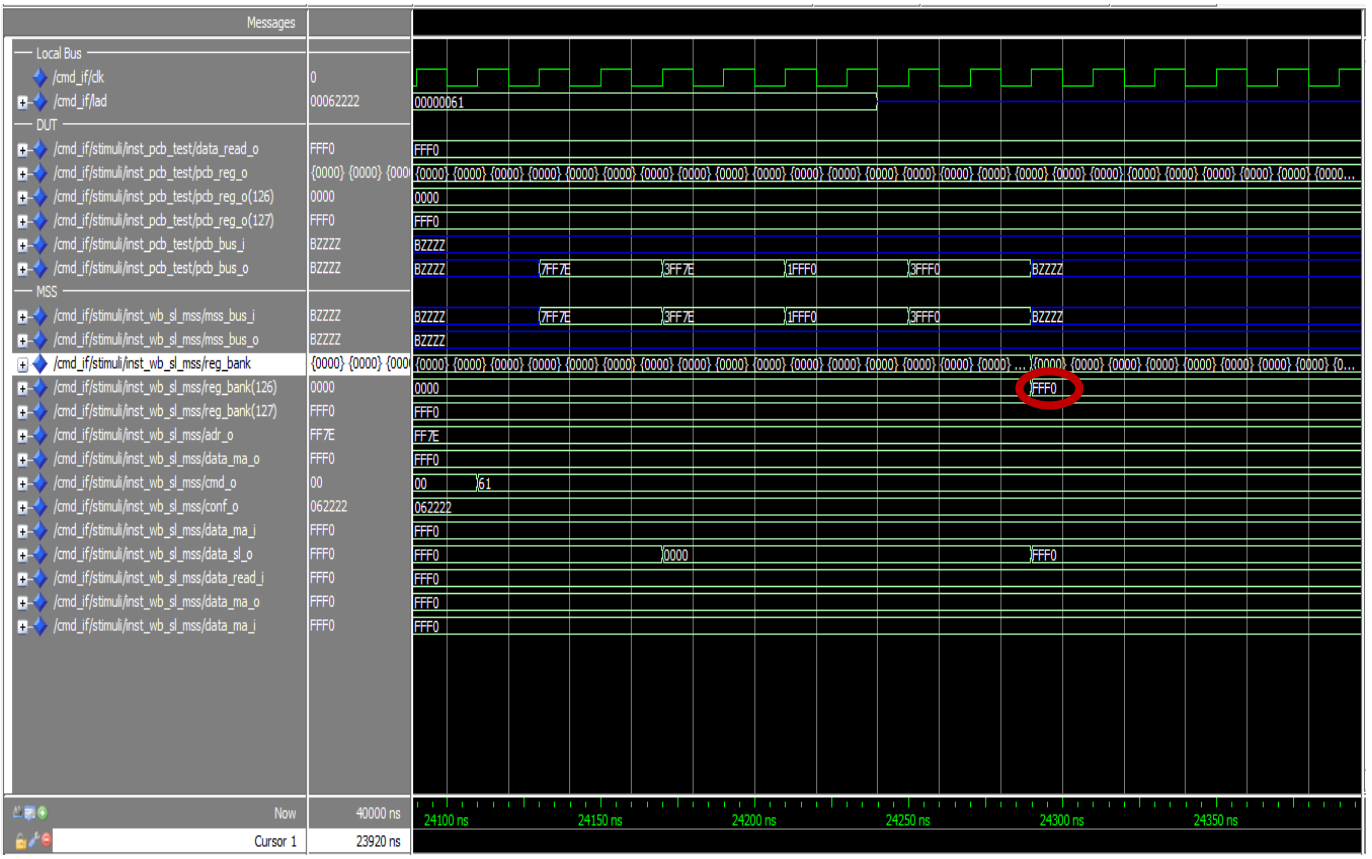


Figure 24 Wishbone-MSS Testbench pic.3

In the figure above a write command in slave mode is shown. It has to be stressed out, that only the register 126 of MSS-Slave register bank received the data package as the DUT functions in this case as a master device.

5. Software Interface implementation

In this chapter, the communication interface with the PCIS3BASE card will be described. The communication interface is developed in C#. The development of the software Interface didn't start from scratch but was based on another implementation with which the communication with the 8 PIB ports was possible. Starting from that point, the MSS Graphic User Interface (GUI) was developed in such a way, that the user is able to give the MSS commands through a text file, which will be executed by the MSS module.

5.1 Design of the Software Interface

Two GUIs are implemented. The first one is used as the main GUI used from the user and the second one is used for debugging purposes. It should be taken into consideration that it is not possible to use both GUIs simultaneously as there is only one socket connection established and there is a conflict when both software interfaces try to connect with the FPGA on the PCIS3BASE.

5.1.1 Main Software Interface

First of all, the main features of the main software Interface will be described. On these features the software Interface will be based to accomplish the communication between the PC-user and the CON9 connector through the MSS module. This GUI will not demand from the user to speak separately to each register of the MSS module. As input it will accept only a text file which will contain assembly commands. Reading this text file, the information gathered is related automatically to the corresponding MSS registers which are responsible for passing on finally the information to the CON9 pins. What is more, depending on the requirements of the user, the assembly commands inside the text file can be executed either one by one or all together with one button click. Obviously, some extra utilities must be added inside the GUI which will give the opportunity to the user to control the function of the MSS module.

- ✓ The user should be able to choose the state of the MSS module.
- ✓ The user should be able to control with a self-test button if the MSS module works properly.
- ✓ The user should be able to see the content of register **data_ma_i** in case the MSS-Slave device receives a data package.
- ✓ The user should be able to see the content of register **data_read_i** in case the DUT device receives a data package.
- ✓ Four text fields must be added, which will show either the content of a specific PCB/MSS-Slave register (a member of the register bank) or a field of registers in the PCB/MSS-Slave register banks starting from a specific address and ending to another specific address.

- ✓ Another two text fields must be added, which will be used for debugging purposes as they will show the content of registers **counter_orders** (number of orders) and **counter_cycles** (number of clock cycles of the last order)**.**

A figure follows below, showing an overview of the Main GUI.

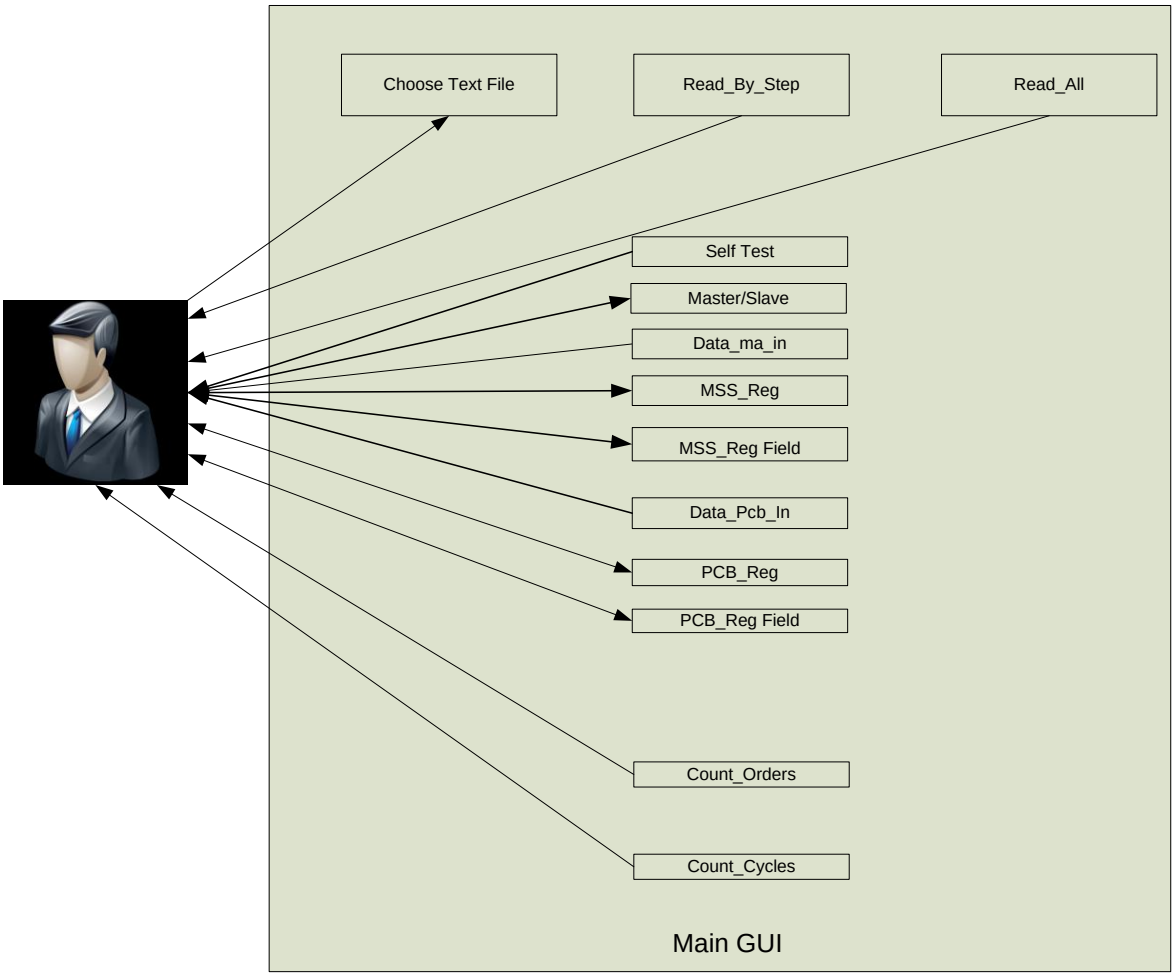


Figure 25 Main GUI overview

For instance, the user can use the *Choose Text File* button to upload a text file and that's why the arrow is directed towards the *Choose Text File* button. The button *Data_Ma _In* is used only to read the content of register **Data_ma_in** and therefore, the arrow is directed towards the user. If the user can write and read a text field, such as PCB_Reg, the arrow is bidirectional.

5.1.2 Debugging Software Interface

In this GUI the user is able to control the content of every single register inside the MSS module. Apart from some of the text fields mentioned above, there are some extra text fields which show the content of registers **Adr_o, Data_ma_o, Conf_o** and **Cmd_o.**

Another figure follows below, which gives an overview of the Debugging GUI.

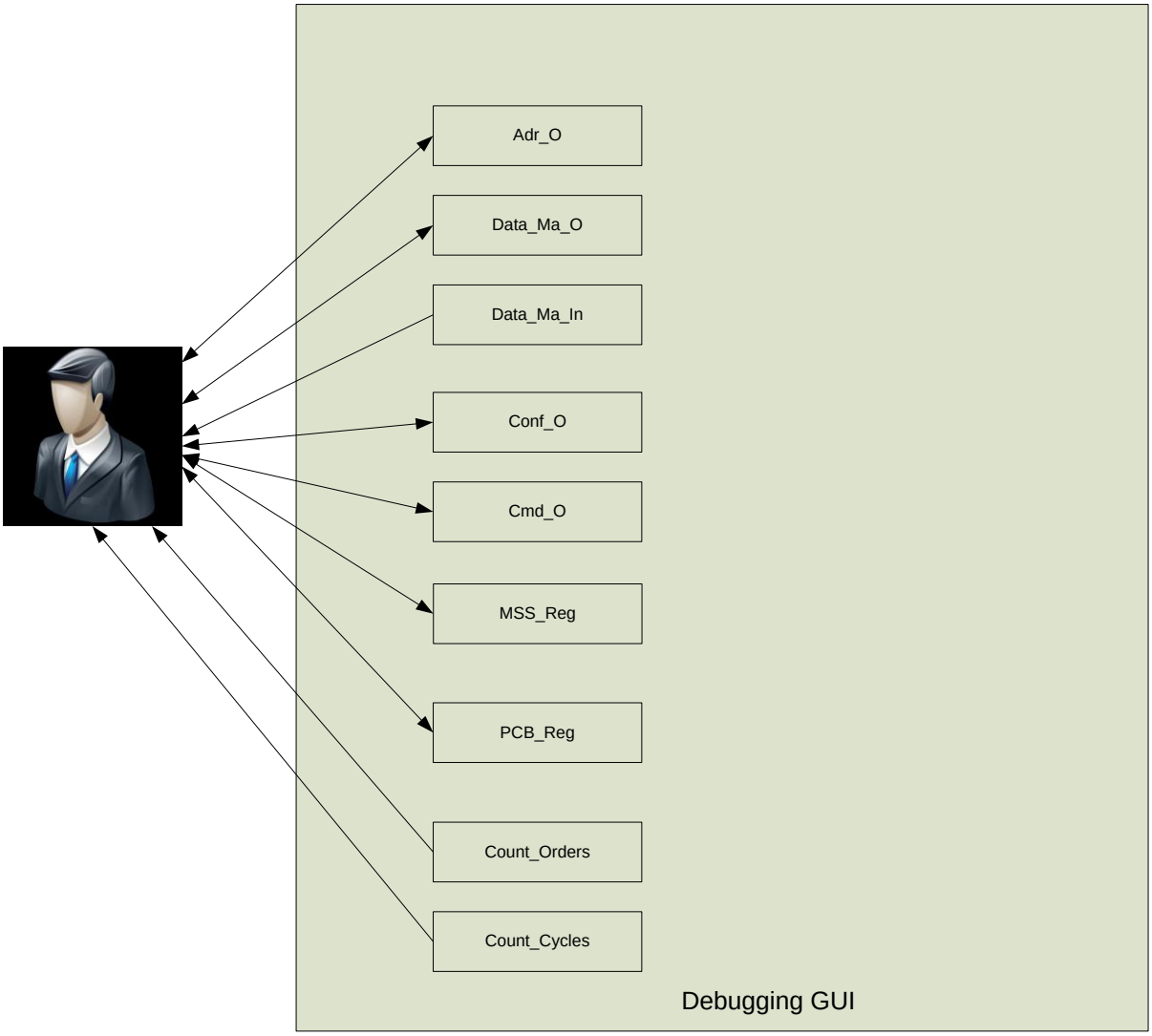


Figure 26 Debugging GUI overview

5.2 Implementation in C#

Within this subchapter both GUIs as well as the code structure will be explained.

5.2.1 Code structure

The main components of the API will be shown below and the function of each source file will be explained.

- **wishbone_pcis3base.cs** : Contains all the constants which represent specific addresses of the slave devices.
- **UdmsHandler.cs :** Implements the communication with the PIB ports.
- **s3base.cs :** The FPGA design is read and the communication with the MSS registers is implemented
- **Program.cs :** Includes the method main() and the GUI is called.
- **GPIO_Tool_MainWindow.cs :** Implements the function of every button of the GUI which doesn't belong to the Table defined in GPIO_Tool_MainWindow.Designer.cs
- **GPIO_Tool_MainWindow.Designer.cs :** Defines the design of the GUI.
- **GPIO_Commands_UserControl.cs :** Implements the function of the buttons which belong to the Table of the GUI.
- **GPIO_Commands_UserControl.Designer.cs :** Defines the elements of each line of the Table in the GUI.

5.2.2 Main Software Interface

The figure shows the GUI which is generated from the main API.

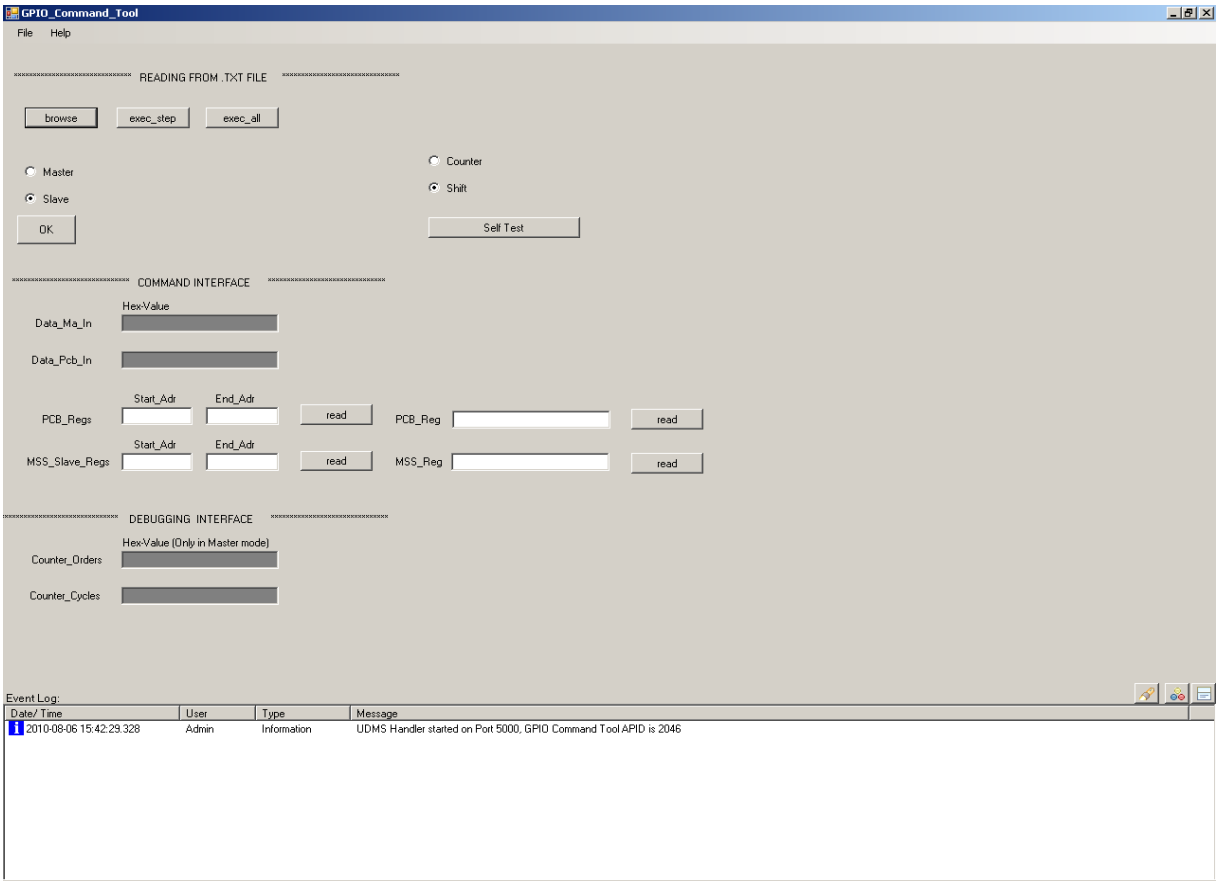


Figure 27 Main GUI

By default the MSS module is set to slave mode. The user is able to change the mode of the MSS module by using the radio-button. Moreover, pressing the *browse* button a new window pops out which gives the opportunity to choose a text file from the computer's file system.

The figure below shows exactly what happens when the *browse* button is pressed.

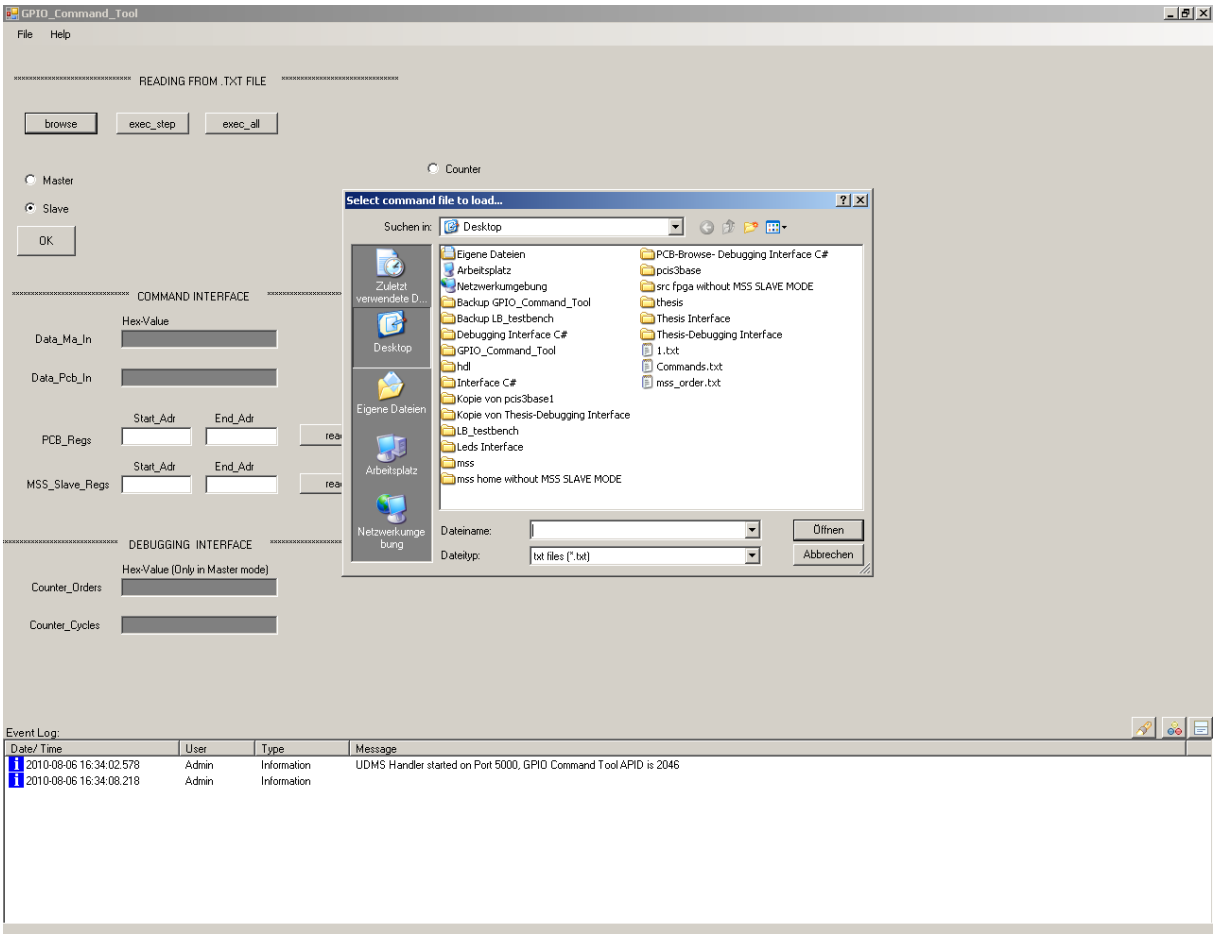


Figure 28 Pressing Browse button

The button *read_step* executes one by one the commands included inside the text file while the button *read_all* executes all the commands in one click.

Data_ma_in is a text field which shows the content of register **Data_ma_in.** The same happens also with the text fields Cnt_Cycle, Cnt Orders, PCB_Reg, MSS_Reg which show the content of registers **counter_orders, counter_cycles** and a specific member of the register banks of MSS-Slave and DUT respectively. MSS_Slave_Regs and DUT_Regs show the content of a field of registers inside the register banks of MSS-Slave and DUT respectively .

More information about the use of the main GUI follows in the next figure.

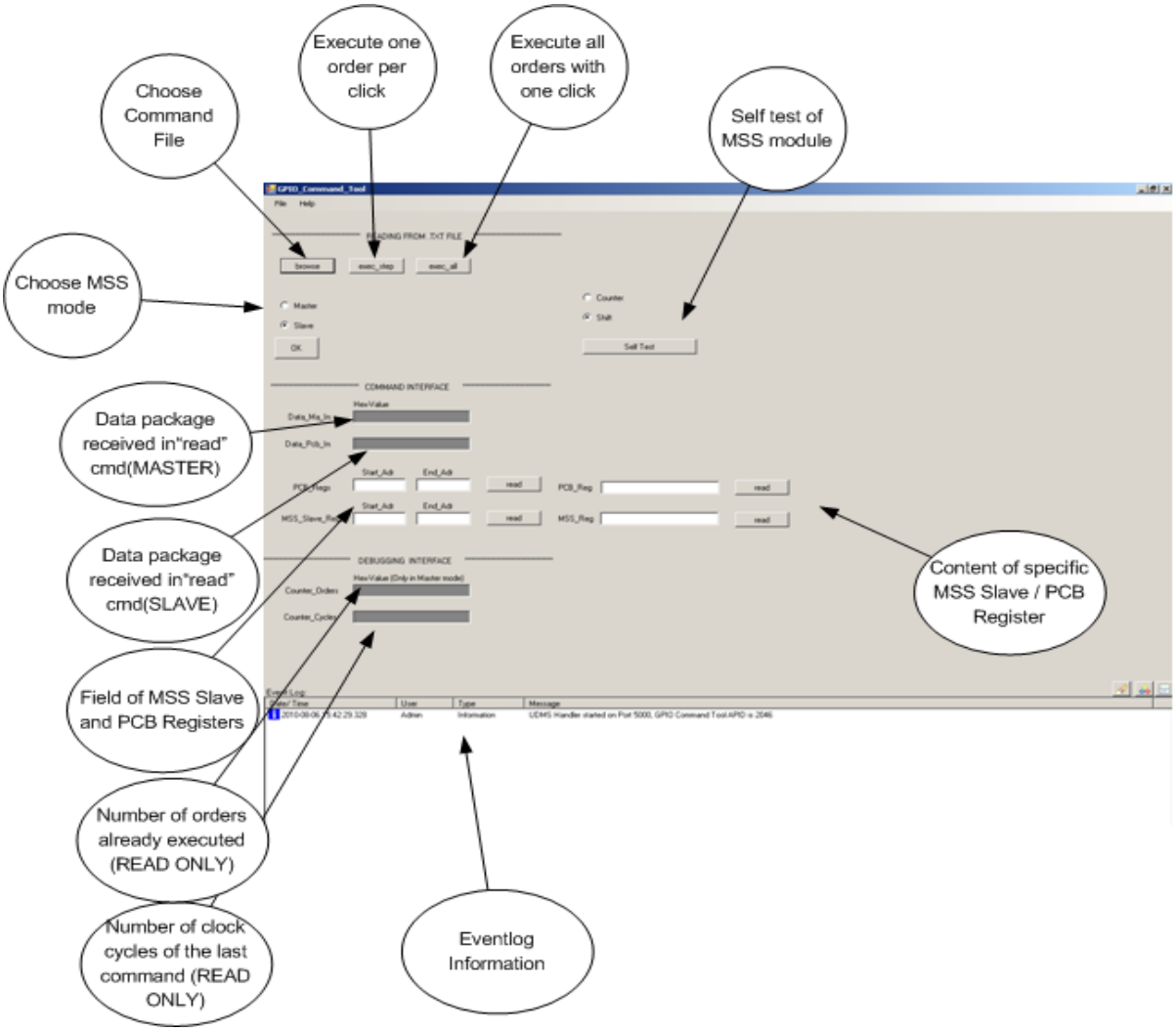


Figure 29 Use of Main GUI

5.2.3 Debugging Software Interface

The next figure shows the GUI which is generated from the API which is, in turn, developed for debugging purposes.

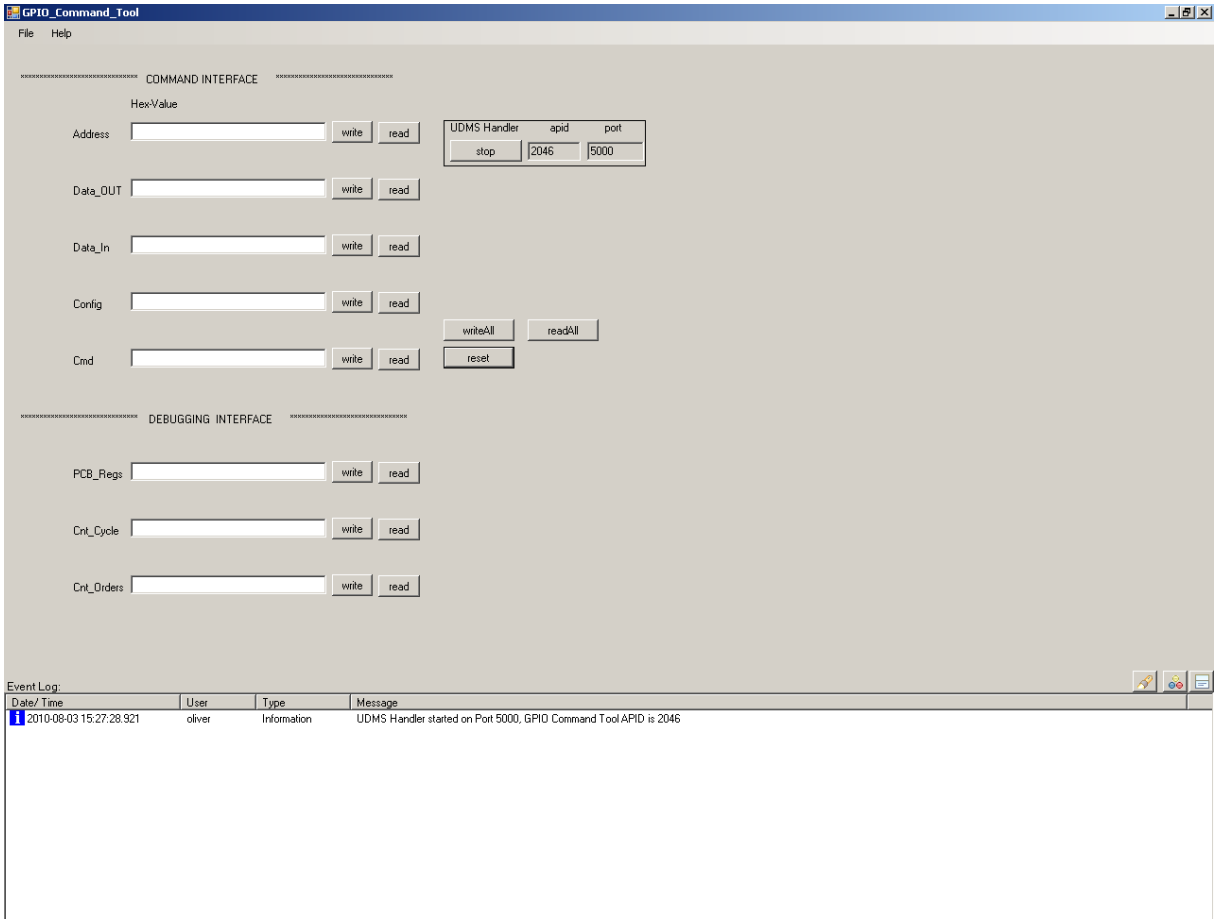


Figure 30 Debugging GUI

Looking at the figure above is obvious that the user has now access to every single register of the MSS module. Consequently, he is able either to write a register or read the content of a register whose content may have been earlier influenced by a command executed from the Main GUI.

6. MSS module on Spartan-3 FPGA

In chapter 4, is already explained, how the MSS module works in the simulation environment of Modelsim. Now, within the chapter 6, it is time to take advantage of Spartan-3 FPGA resources and implement the MSS slave device in the FPGA. For the implementation the Xilinx enviroment ISE 10.1 will be used.

6.1 Code structure in Xilinx ISE

In the programming enviroment of Xilinx ISE the following code structure is used:

- Pcis3base_top.vhd
  - ✓ Wb_intercon.vhd
  - ✓ Wb_syscon.vhd
  - ✓ Wb_ma_plx.vhd
  - ✓ Wb_sl_sdr.vhd
  - ✓ Wb_sl_flash.vhd
  - ✓ Wb_sl_gpio.vhd
  - ✓ Wb_sl_mss.vhd
  - ✓ Pcb_test.vhd
  - ✓ Wb_sl_timer.vhd
  - ✓ Pcis3base.ucf

6.2 FPGA implementation report

In the table below the number of several types of hardware elements which are used inside the Spartan-3 FPGA is depicted.

Adders/Subtractors	**8**
3-bit subtractor	1
32-bit adder	1
5-bit subtractor	10
7-bit adder	1
Counters	**21**
16-bit up counter 2	2
32-bit down counter	18
9-bit up counter	1
Registers	**2664**
Flip-Flops	2664
Comparators	**11**
32-bit comparator equal	1
32-bit comparator lessequal	9
32-bit comparator not equal	1
Multiplexers	**36**

1-bit 8-to-1 multiplexer	32
16-bit 128-to-1 multiplexer	3
32-bit 4-to-1 multiplexer	1

Table 21 Advanced HDL Synthesis Report

The device utilization summary follows in the table below.

Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	5614	26624	21%
Number of 4 Input LUTs	7380	26624	27%
Logic Distribution			
Number of occupied Slices	7855	13312	35%
Number of Slices containing only related logic	7855	7855	100%
Number of Slices containing unrelated logic	0	7855	0%
Total Number of 4 input LUTs	7465	26624	28%
Number used as logic	7380		
Number used as a Route-thru	85		
Number of bonded IOBs	200	333	60%
IOB Flip Flops	1		
Number of BUFGMUXs	1	8	12%
Number of DCMs	2	4	50%

Table 22 Device Utilization Summary

6.3 Testing MSS module

Having already described the hardware features that are generated in the design process in the FPGA and having successfully generated the .bin programming file, the next step is to control the MSS module function using the software interface which was described in chapter 5.

6.3.1 Testing process

First of all, it will be mentioned what other hardware elements were used in the testing procedure of MSS module.

- ✓ A breakout box is used in order to control the value of each of the 78 CON9 connector's pins.

- ✓ A cable harness is used for the connection between the breakout box and the CON9 connector.
- ✓ The oscilloscope Tektronix TDS 784D is used for the control of MSS timing
- ✓ A loop-back 78-pin connector is used for testing the MSS module in Slave mode.

The use of the breakout box was really significant not only for the control of the MSS timing but also for the control of every single bit on the MSS bus. The following table makes evident the relationship between the CON9 pins and the PIB ports. It shows which pin of CON9 connector corresponds to a specific bit of the PIB ports.

Ports / Bits	0	1	2	3	4	5	6	7
0	**10**	**29**	**49**	**68**	**39**	**20**	**59**	**78**
1	**8**	**27**	**47**	**66**	**37**	**18**	**57**	**76**
2	**6**	**25**	**45**	**64**	**35**	**16**	**55**	**74**
3	**4**	**23**	**43**	**62**	**33**	**14**	**53**	**72**
4	**9**	**28**	**48**	**67**	**38**	**19**	***58***	**77**
5	**7**	**26**	**46**	**65**	**36**	**17**	***56***	**75**
6	**5**	**24**	**44**	**63**	**34**	**15**	***54***	**73**
7	**3**	**22**	**42**	**61**	**32**	**13**	***52***	**71**

GPIO PINS
DUT PINS
MSS PINS
Unused PINS

Table 23 CON9-PIB ports connection

As the MSS bus is speaking only to three PIB ports, the user has to take into consideration only the first two and the fourth column from the table above and what is more, as the MSS bus has length equal to 20 bits, the bits 4,5,6 and 7 of the fourth column (port 4) can be ignored. The pins of CON9 which will be most tested, are obviously the control signals of MSS bus which correspond to the first four bits of port 4.

The table below shows the relationship between MSS bus and CON9 connector's pins.

MSS bus bits	Information	CON9 pins
0	Adr/Data bit 0	10
1	Adr/Data bit 1	8
2	Adr/Data bit 2	6
3	Adr/Data bit 3	4
4	Adr/Data bit 4	9
5	Adr/Data bit 5	7
6	Adr/Data bit 6	5
7	Adr/Data bit 7	3
8	Adr/Data bit 8	29
9	Adr/Data bit 9	27
10	Adr/Data bit 10	25
11	Adr/Data bit 11	23
12	Adr/Data bit 12	28

13	Adr/Data bit 13	26
14	Adr/Data bit 14	24
15	Adr/Data bit 15	22
16	Rd control signal	39
17	Wr control signal	37
18	Ale control signal	35
19	Sel control signal	33

Table 24 CON9-MSS bus connection

The table below shows the relationship between DUT bus and CON9 connector's pins.

MSS bus bits	Information	CON9 pins
0	Adr/Data bit 0	49
1	Adr/Data bit 1	47
2	Adr/Data bit 2	45
3	Adr/Data bit 3	43
4	Adr/Data bit 4	48
5	Adr/Data bit 5	46
6	Adr/Data bit 6	44
7	Adr/Data bit 7	42
8	Adr/Data bit 8	68
9	Adr/Data bit 9	66
10	Adr/Data bit 10	64
11	Adr/Data bit 11	62
12	Adr/Data bit 12	67
13	Adr/Data bit 13	65
14	Adr/Data bit 14	63
15	Adr/Data bit 15	61
16	Rd control signal	59
17	Wr control signal	57
18	Ale control signal	55
19	Sel control signal	53

Table 25 CON9-DUT bus connection

6.3.2 Download the Design

For the downloading of the design into the FPGA the Cesys Software tool **cesys-Monitor** is used**.**

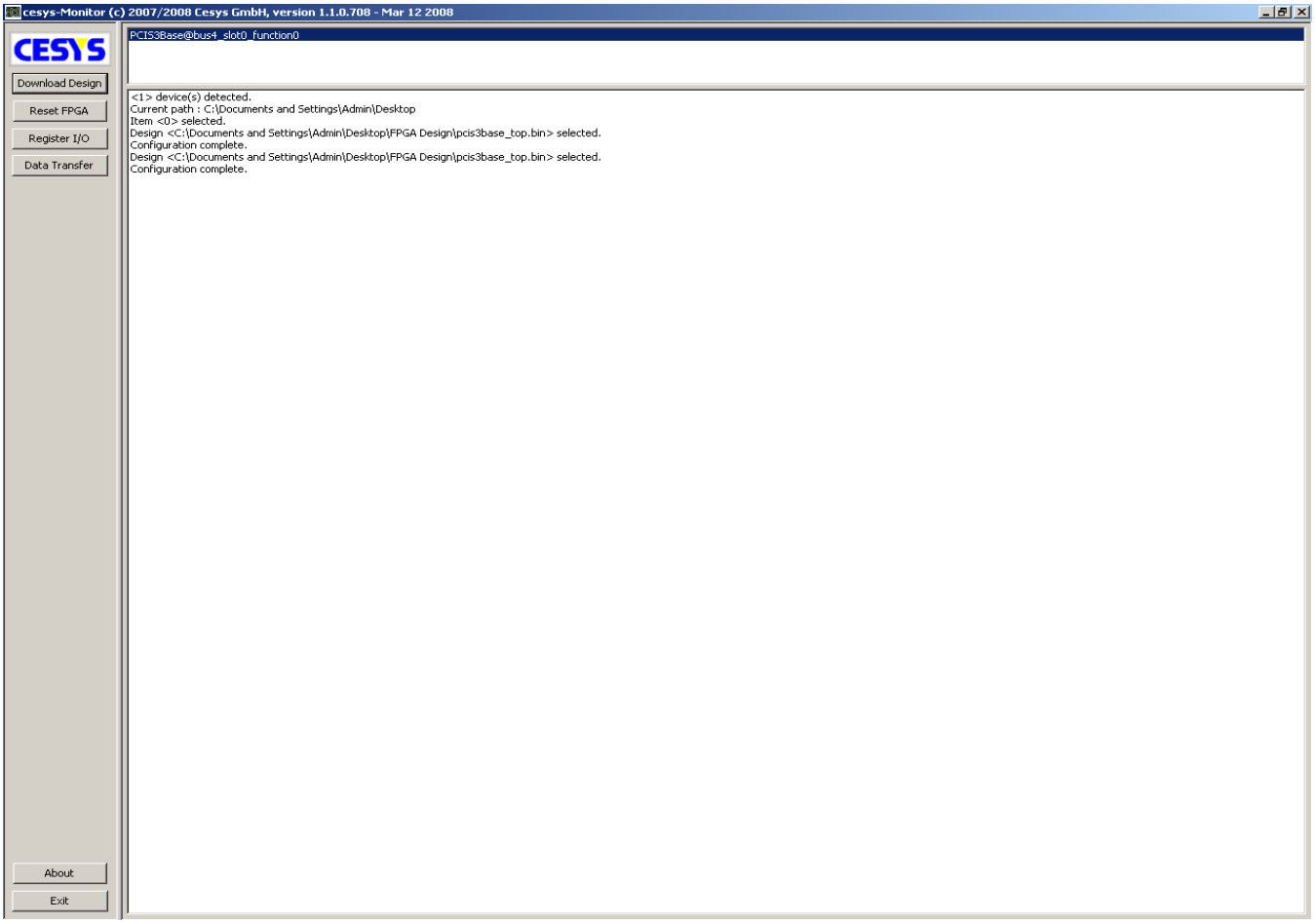


Figure 31 Cesys Monitor

6.3.3 Testing MSS Master

Taking for granted that the MSS module works in master mode (*Master* radiobutton must be checked and then click OK), first of all the user should choose the command text file.

```
conf 027777
wr 0001 0077
wr 0005 0009
wr 0002 000A
wr 0004 0007
wr 0005 0077
rd 0F04
rd 0F02
rd 0F01
wr 0F11 0002
rd 0005
rdv 0011 0001
rdv 0F11 0002
```

Code 15 MSS Command file

Afterwards, it can be defined, whether all commands should be executed instantly using the *exec_all* button or should be executed step by step using the *exec_step* button.

Pressing the *exec_step* button an output log file is received which informs the user what kind of command is executed and its parameters. For instance, when the command **wr 0005 0077** is executed, the output log file (**report.txt**)"MASTER Cmd: wr A: 0005 D: 0077" is receiced . That means that a data package is sent from the MSS-Master towards the DUT register bank and in particular to the address 0005.

Pressing the *exec_all* button an output log file (**report_all.txt**) is received showing all the commands which were executed.

```
MASTER  Cmd: conf  Cnfg: 027777
MASTER  Cmd: wr  A: 0001  D: 0077
MASTER  Cmd: wr  A: 0005  D: 0009
MASTER  Cmd: wr  A: 0002  D: 000A
MASTER  Cmd: wr  A: 0004  D: 0007
MASTER  Cmd: wr  A: 0005  D: 0077
MASTER  Cmd: rd  A: 0004  D: 0007
MASTER  Cmd: rd  A: 0002  D: 000A
MASTER  Cmd: rd  A: 0001  D: 0077
MASTER  Cmd: wr  A: 0F11  D: 0002
MASTER  Cmd: rd  A: 0005  D: 0077
MASTER  Cmd: rdv A: 0011  D: 0002  FALSE
MASTER  Cmd: rdv A: 0F11  D: 0002  OK
```

Code 16 Output log file in Master mode in case the input command file is code 15

As the MSS module works in master mode, in the case of a read command, the register **Data_Ma_in** receives the data package which was sent from the DUT. For instance, at the execution of the command **rd 0004** the GUI image is the following.

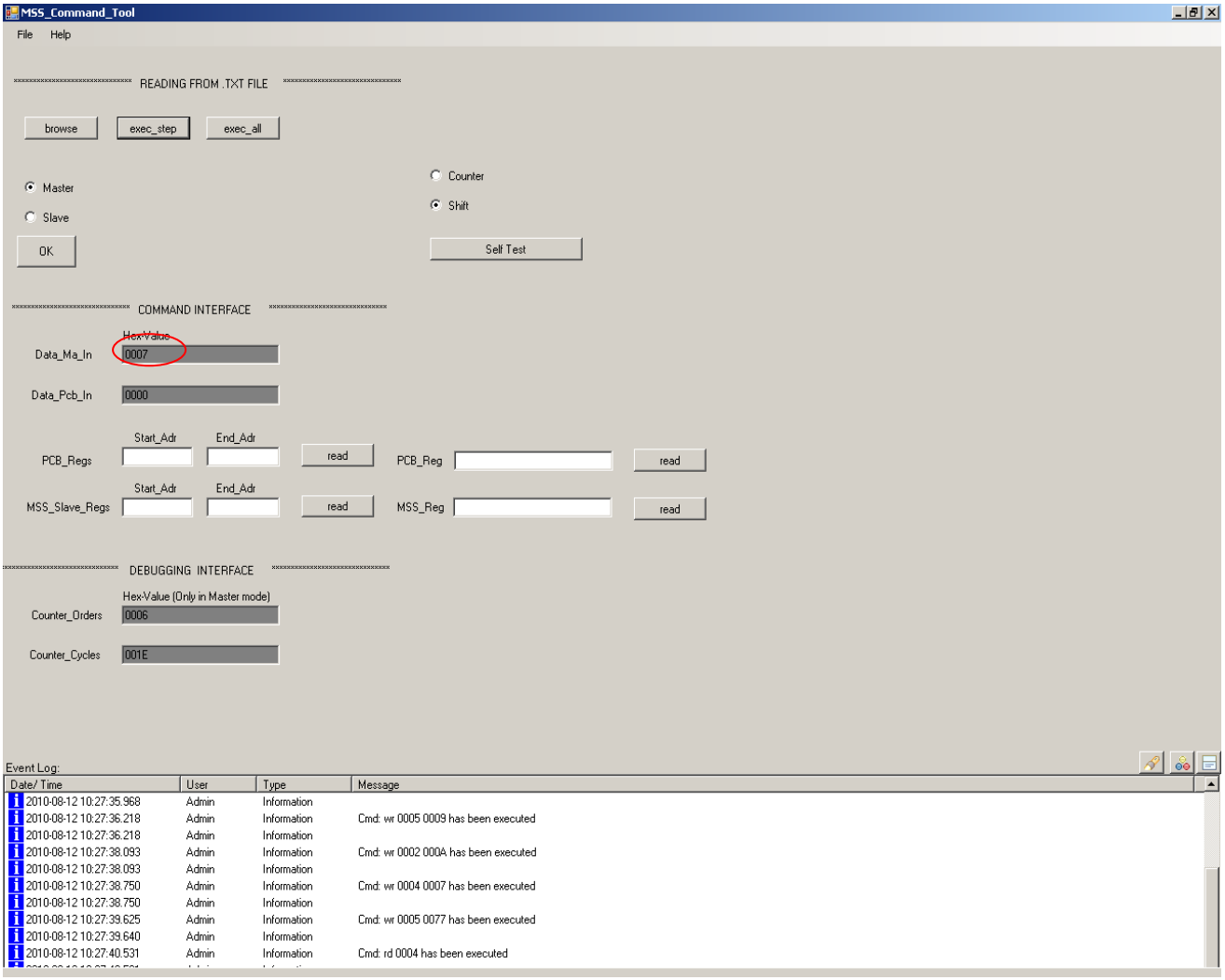


Figure 32 GUI in Master mode

6.3.4 Testing MSS Slave

Taking now for granted that the MSS module works in slave mode (*Slave* radiobutton must be checked and then click OK), and giving as input command file the same file as in Code 15, the user presses at first the *exec_step* button. If the command **wr 0005 0077** is executed, the output log file **(report.txt)** "SLAVE Cmd: wr A: 0005 D: 0077" is received. That means that a data package is sent from the DUT towards the MSS-Slave register bank and in particular to the address 0005.

Pressing now the *exec_all* button another output log file (**report_all.txt)** is received showing all the commands which were executed.

```
SLAVE  Cmd: conf  Cnfg: 027777
SLAVE  Cmd: wr  A: 0001  D: 0077
SLAVE  Cmd: wr  A: 0005  D: 0009
SLAVE  Cmd: wr  A: 0002  D: 000A
SLAVE  Cmd: wr  A: 0004  D: 0007
SLAVE  Cmd: wr  A: 0005  D: 0077
SLAVE  Cmd: rd  A: 0004  D: 0007
SLAVE  Cmd: rd  A: 0002  D: 000A
```

```
SLAVE  Cmd: rd  A: 0001  D: 0077
SLAVE  Cmd: wr  A: 0F11  D: 0002
SLAVE  Cmd: rd  A: 0005  D: 0077
SLAVE  Cmd: rdv A: 0011  D: 0002  FALSE
SLAVE  Cmd: rdv A: 0F11  D: 0002  OK
```

Code 17 Output log file in Slave mode in case the input command file is code 15

As the MSS module works now in slave mode, in the case of a read command, the register **Data_Pcb_in** receives the data package which was sent from the DUT. For instance, at the execution of the command **rd 0002** the GUI image is the following.

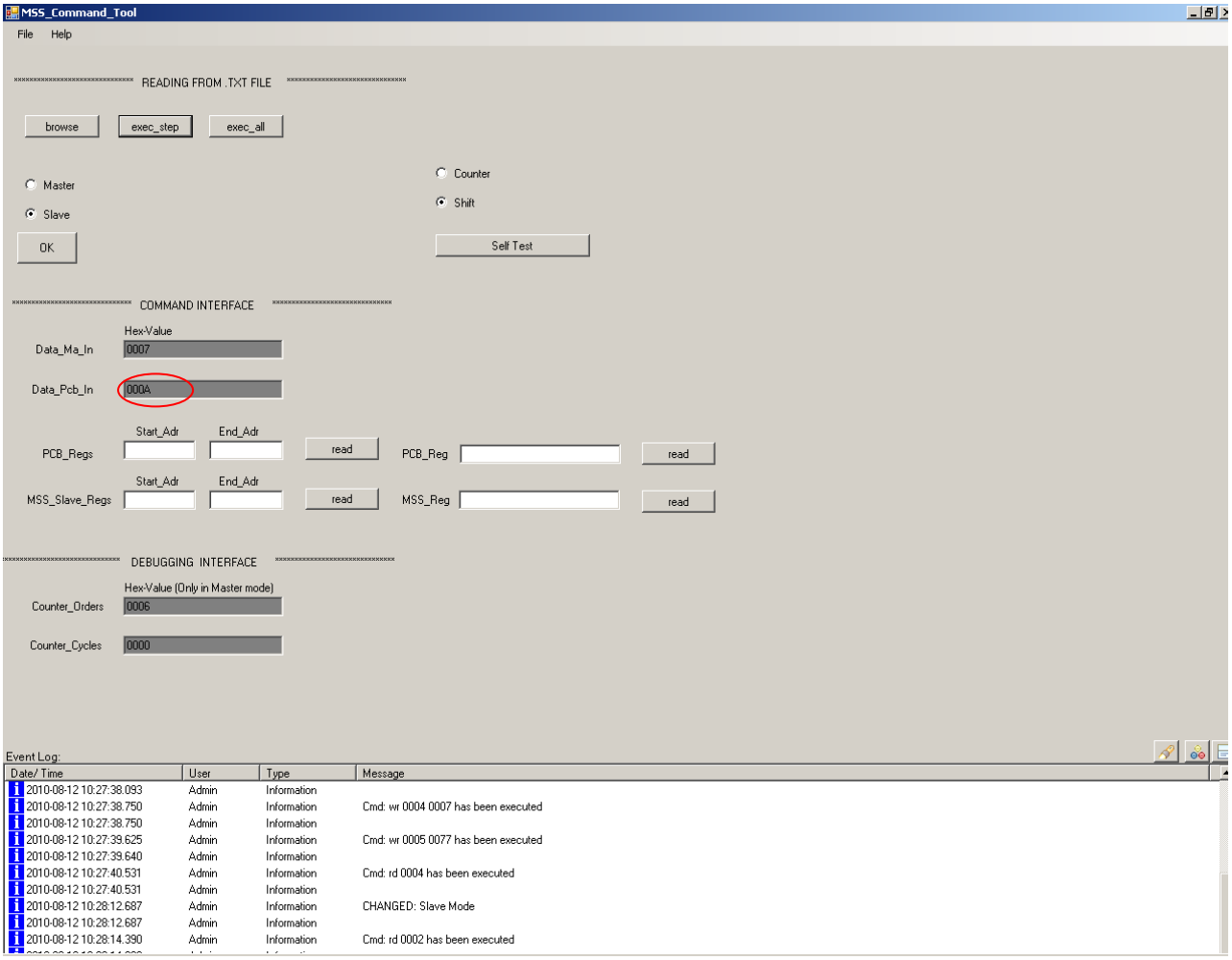


Figure 33 GUI in Slave mode

6.4 MSS Timing

Within this subchapter the MSS timing, which the MSS-Master module generates, will be shown depending on the input commands of the user. As it is already mentioned, the most important role, in matters of the timing specifications, plays the content of the configuration register. The user can adjust the configuration register using the command **config X.** X stands for a 24 bit vector which is divided in 6 groups of 4 bits each of which defines the duration of each stage of the command. The last 4 groups define the duration of each of the four stages of a write command and the last 5 groups define the duration of each of the five stages of a read command.

Below some captures from the oscilloscope are shown, which depict the MSS timing which is generated by the execution of some groups of commands.

6.4.1 Write commands

1st Command group:
conf 024744
wr 0001 0077

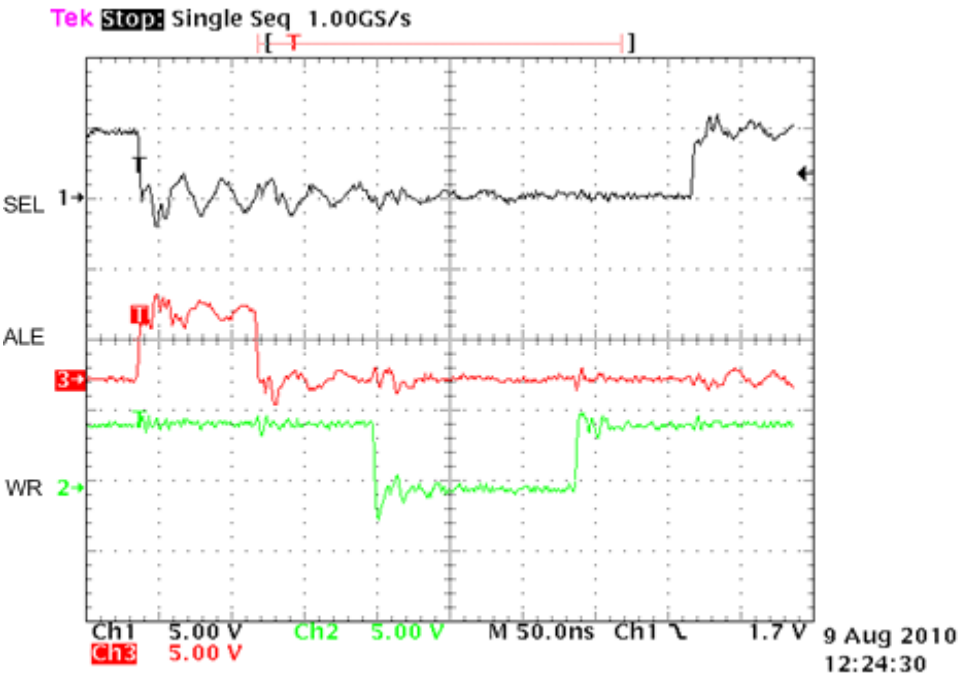


Figure 34 MSS-Timing 1

The duration of the low level of **sel** control signal defines how long the command cycle lasts (how long the microprocessor is selected). Comparing the duration of the low level of **sel** control signal with the sum of the last four hex numbers of configuration register can be understood that the **config** register really defines how long the command lasts.

2nd Command group:
conf 024448
wr 0001 0077

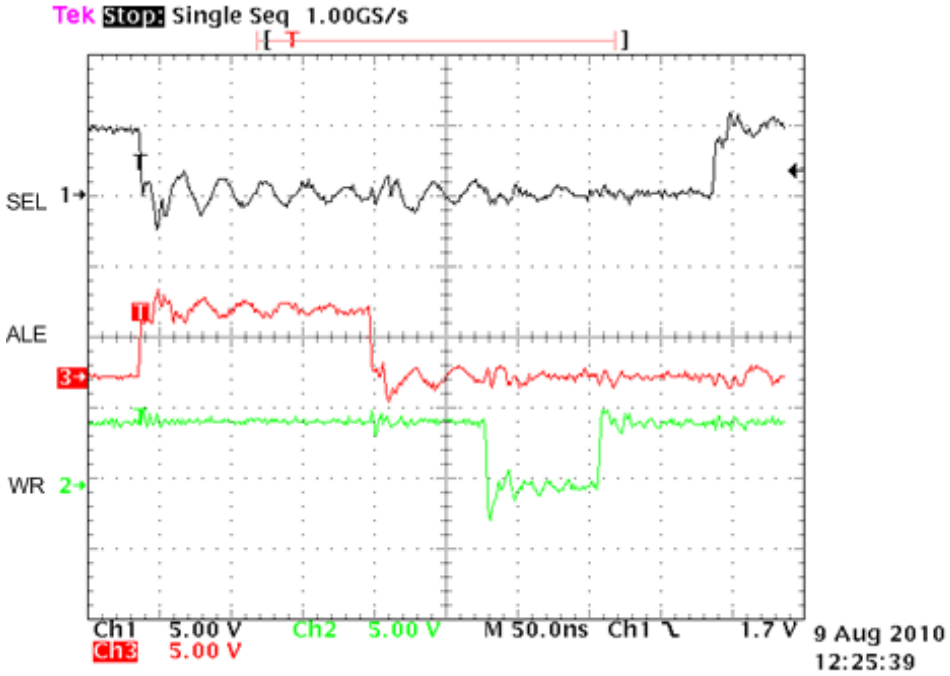


Figure 35 MSS-Timing 2

Comparing the last two figures can be understood that the **ale** control signal stays longer in high level in the last figure. This happens due to the fact that the last hex number of the **config** register in the second command group has a greater value in comparison with the last hex number of the **config** register in the first command group.

3rd Command group:
conf 024848
wr 0001 0077

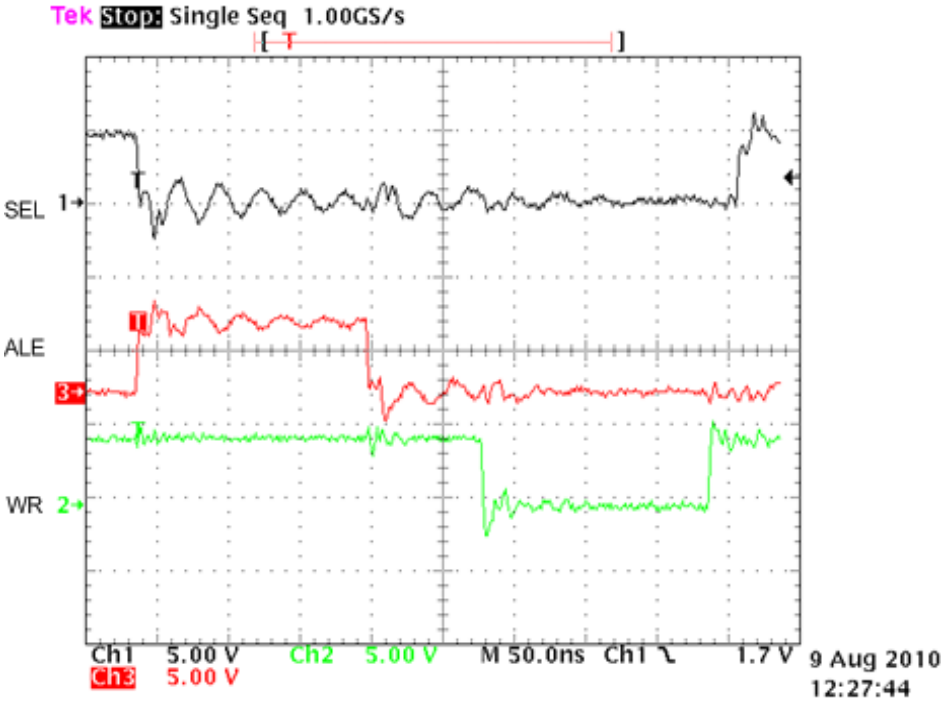


Figure 36 MSS-Timing 3

Comparing the last two figures, the **wr** control signal stays two times longer in low level in the last figure, as the third hex number starting from the end in the last command group is two times greater in comparison with the same hex number in the previous command group.

6.4.2 Read commands

4th Command group:
conf 022422
rd 0001

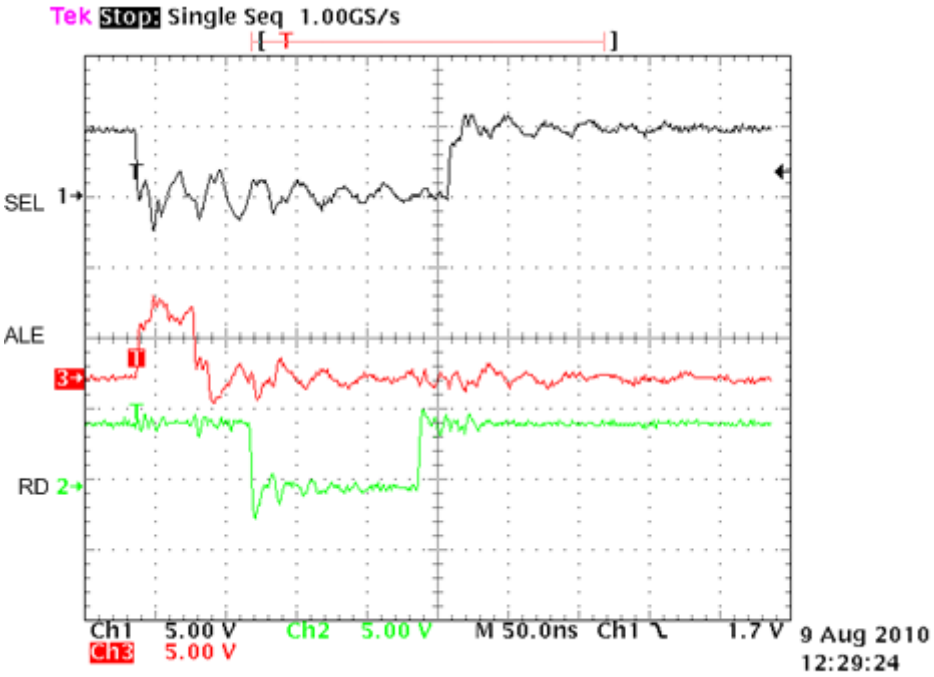


Figure 37 MSS-Timing 4

As the read command is divided in five stages, its duration is defined from the last five hex numbers of **config** register. That means, that in this case its duration must be equal to 12*20 ns (20 ns is the clock cycle) and that is true as the duration of the low level of **sel** control signal is also equal to 240ns.

5th Command group:
conf 026622
rd 0001

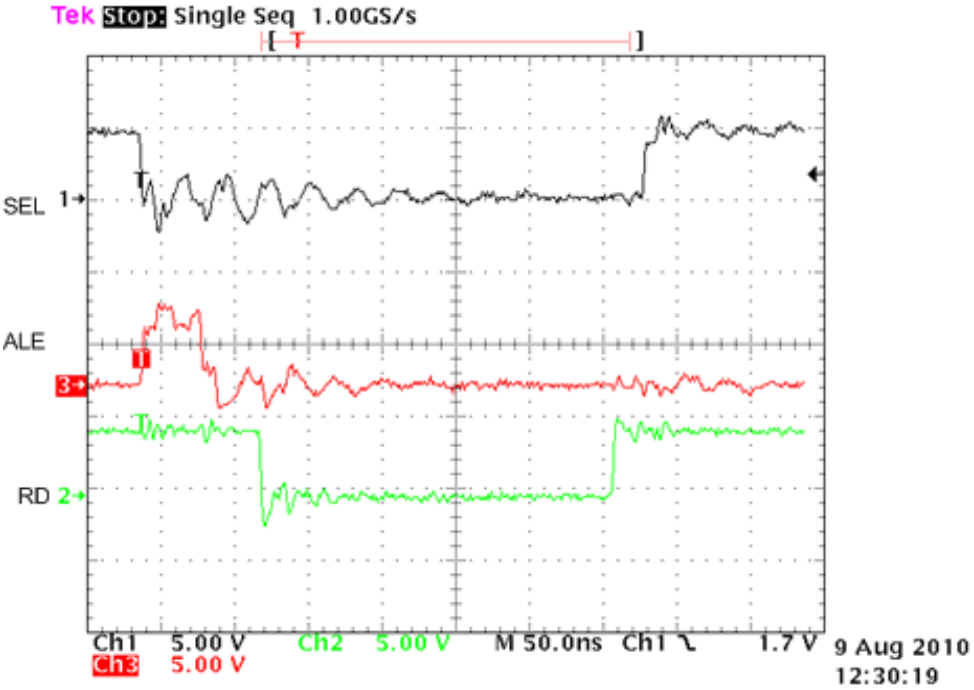


Figure 38 MSS-Timing 5

Comparing now the last two read commands, it can be mentioned that the **rd** control signal stays two times longer in low level in the last figure, as the sum of the third and the fourth hex number starting from the end of **config** register in the last command group is two times greater in comparison with the same sum of hex numbers in the previous command group.

7. Conclusion

To sum up, the achievements and the further research opportunities will be mentioned.

7.1 Achievements

First of all, the MSS module, which was implemented, succeeded to simulate a Microprocessor's timing. Using the interface implemented, it is now possible to test thoroughly slave PCBs. That means, that the slave PCBs can be integrated inside the PDHU (Payload Data Handling Unit) box seamlessly. Last but not least, using the new Software interface the whole testing procedure is shortened, as the testing process can now function automatically.

7.2 Further research and development

The module implemented lays the fundamentals for further research as it gives the opportunity to develop new interfaces such as Channel Link, Space Wire etc.What is more, taking advantage of the Digital Clock Manager (DCM) faster and more exact timing specifications can be implemented.

7.3 Space Wire Interface

SpaceWire is a spacecraft communication network based in part on the IEEE 1355 standard of communications. It is coordinated by the European Space Agency (ESA) in collaboration with international space agencies including NASA, JAXA and RKA. Within a SpaceWire network the nodes are connected through low-cost, low-latency, full-duplex, point-to-point serial links and packet switching wormhole routing routers. SpaceWire covers two (physical and data-link) of the seven layers of the OSI model for communications.

The scope of the Spice Wire Standard is the physical connectors and cables, electrical properties, and logical protocols that comprise the SpaceWire data link. SpaceWire provides a means of sending packets of information from a source node to a specified destination node. SpaceWire does not specify the contents of the packets of information.

The Space Wire Standard covers the following protocol levels:

- Physical level: Defines connectors, cables, cable assemblies and printed circuit board tracks.
- Signal level: Defines signal encoding, voltage levels, noise margins, and data signaling rates.
- Character level: Defines the data and control characters used to manage the flow of data across a link.
- Exchange level: Defines the protocol for link initialization, flow control, link error detection and link error recovery.
- Packet level: Defines how data for transmission over a SpaceWire link is split up into packets.
- Network level: Defines the structure of a SpaceWire network and the way in which packets are transferred from a source node to a destination node across a network. It

also defines how link errors and network level errors are handled.

Furthermore, SpaceWire utilizes asynchronous communication and allows speeds between 2 Mbit/s and 400 Mbit/s.SpaceWire also has very low error rates, deterministic system behavior, and relatively simple digital electronics.. SpaceWire replaced old PECL differential drivers in the physical layer of IEEE 1355 DS-DE by low-voltage differential signaling. However, one of its main features is that it supports automatic failover. That means that it lets data find alternate routes, so a spacecraft can have multiple data buses, and be made fault-tolerant.

The purpose of the SpaceWire standard is:

- to facilitate the construction of high-performance onboard data handling systems,
- to help reduce system integration costs,
- to promote compatibility between data handling equipment and subsystems, and
- to encourage re-use of data handling equipment across several different missions.

Use of the SpaceWire standard ensures that equipment is compatible at both the component and sub-system levels. Processing units, mass-memory units and down-link telemetry systems using SpaceWire interfaces developed for one mission can be readily used on another mission. This:

- reduces the cost of development (Cheaper),
- reduces development timescales (Faster),
- improves reliability (Better),
- increases the amount of scientific work that can be achieved within a limited budget (More).

Payload processing involves several functions:

- Controlling instruments
- Calibrating instruments
- Collecting data from instruments
- Storing the instrument data
- Processing the data
- Compressing the data
- Sending the data to the down-link telemetry transmitter

SpaceWire is able to support many different payload processing architectures using point-to-point links and SpaceWire routing switches. An architecture can be tuned to the requirements of specific missions.

An example architecture is shown in the diagram below, which uses SpaceWire routers to provide the interconnectivity between instruments, memory and processing modules.

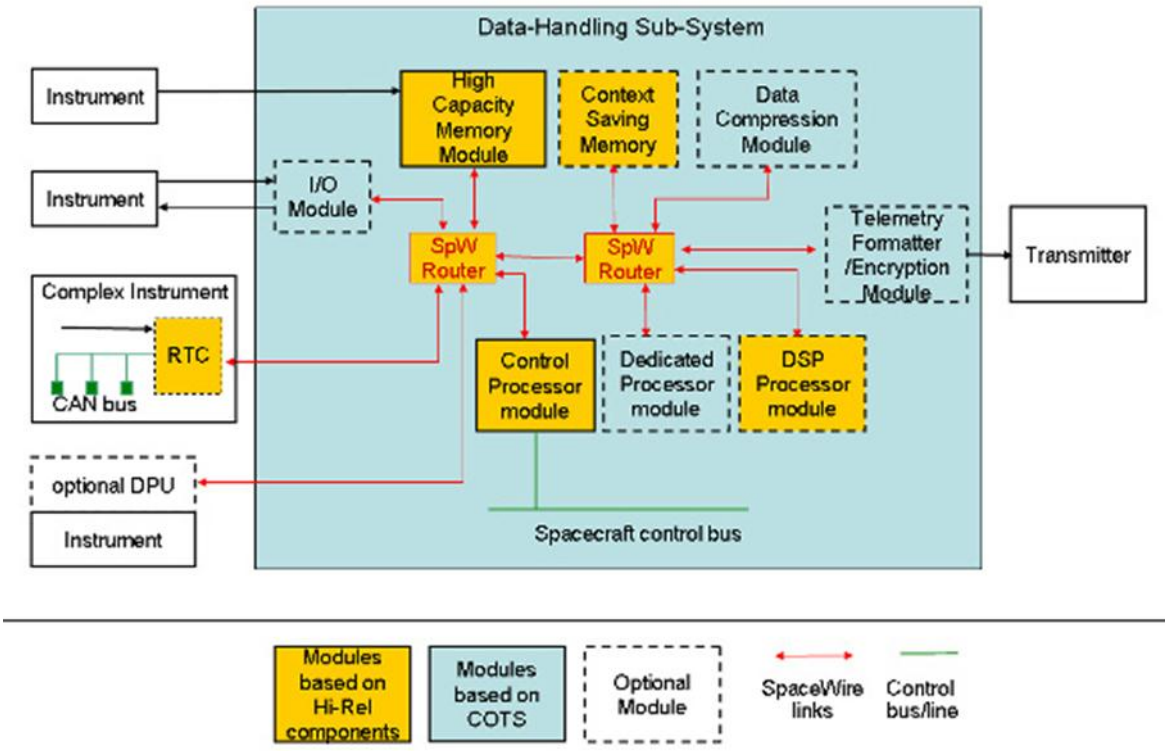


Figure 39 Space-Wire Architecture

The Instrument in the top left-hand corner is a high data-rate instrument. A SpaceWire point-to-point link is used to stream data from this instrument directly into the High Capacity Memory Module. This Memory Module is also connected into the rest of the payload processing architecture by a SpaceWire link to a SpaceWire Router. Data from other instruments can be stored in the High Capacity Memory Module using this link. The data stored in memory from the high data rate instrument can also be accessed using this link, for processing, compression and sending to the down-link telemetry transmitter.

A second Instrument is connection to an input/output (I/O) module. This module is used to connect to instruments that do not have direct SpaceWire connections. The instrument passes data to the I/O module, possibly over a parallel data bus. The I/O module forms this data into SpaceWire packets and sends them to the required destination over the SpaceWire network. This may be the High Capacity Memory Module, a Processor Module, or a Data Compression Module.

The third instrument, the Complex Instrument, has many sub-systems that have to be controlled separately. To do this a low data rate bus, for example the CAN bus, is used, to control and collect data from the various sub-systems. A Remote Terminal Interface (RTI) is used to provide the bridge between SpaceWire and the local bus (e.g. CAN). The Remote Terminal Computer device has been specifically designed to support this function.

The final instrument is an instrument that requires substantial processing so may include its own Data Processing Unit (DPU). The DPU is connected to the SpaceWire network providing processed data to the High Capacity Memory Module for storage before being sent to the down-link telemetry system.

There are four different processing modules shown in the example architecture. The Control Processor module controls the complete set of payload instruments according to commands sent over the Spacecraft Control Bus. The DSP Processor module performs digital signal

processing on the instrument data to extract important information or to help implement computationally intense control loops for the instruments. It may also perform compression of the payload data if the data rates are low. For higher data rate instruments a dedicated Data Compression may be necessary and for instruments that require specific, demanding processing a Dedicated Processor module may be used.

Data from the instruments, High Capacity Memory and processing units can be sent over the SpaceWire network to the Telemetry Formatter/Encryption module under control of the Control Processor module. The Telemetry Formatter/Encryption module sends the data to a ground station via the down-link Transmitter.

A Context Saving Memory is also shown in the architecture diagram. This memory can be used for periodically saving the context of the payload processing system, so that in the event of a failure a previous context can be restored.

Additional SpaceWire links can be added in the network to provide additional bandwidth or to support fault tolerance. The SpaceWire routers may be stand alone units or may be integrated into the memory, processing or other modules. SpaceWire allows standard instruments, memory systems and processing modules to be developed and reused on several missions.

7.4 Channel Link

Channel-Link (C-Link) is a high-speed interface for cost-effectively transferring data at rates from 250 megabits/second to 6.4 gigabits/second over backplanes or cables.

Apart from Spacewire, Channel-Link as well uses LVDS(Low-Voltage Differential Signaling.) and comes in three configurations with three, four, or eight parallel data transfer lanes plus the source-synchronized clock for each configuration. In cable applications, it uses one twisted pair in order to transmit a clock signal, and on the remaining differential pairs it transmits digital data at a bit rate that is seven times the frequency of the clock signal. The backplane applications work the same way except for using differential traces instead of twisted pairs.

The three Channel-Link chipset configurations provide varying user interfaces. For example, the three-lane chipset has 21 single-ended inputs and outputs for the user interface, and the four-lane chipset has 28 single-ended inputs and outputs. The eight-lane chipset has 48 single ended inputs and outputs because it uses one of the 7 serialized bits/lane to DC-balance the other six bits.

Channel-Link is a general-purpose data pipe with no overhead for protocol or encoding.Therefore, there are many system applications for this efficient data transfer technology,such as Camera Link,Multi-faction printers and telecommunication access-aggregator equipment.

We can see below the Channel-Link Chipset Block Diagram :

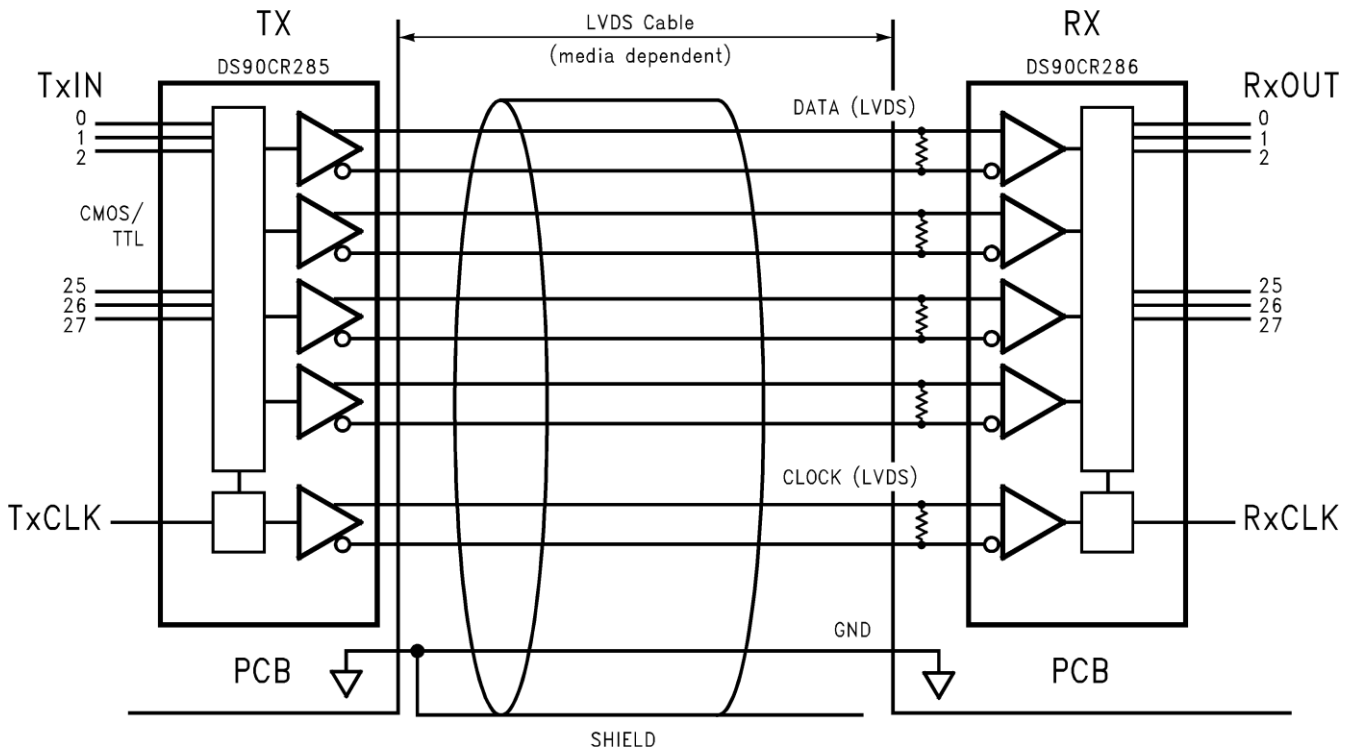


Figure 40 Channel-Link Block Diagram

Bibliography

The Designers Guide to VHDL ,Peter J. Ashenden

Circuit Design with VHDL, Volnei A. Pedroni

http://www.cesys.com/resources/CE031.pdf

http://computer.howstuffworks.com/pci.htm

http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf

http://www.cesys.com/resources/C1050-3506_PIB64IO_UserManual.pdf

http://en.wikipedia.org/wiki/Wishbone_(computer_bus)

Cesys PCIS3BASE PCI Card sourcecode

http://en.wikipedia.org/wiki/Channel-link

http://en.wikipedia.org/wiki/ERC32

http://microelectronics.esa.int/erc32/Hardware%20and%20Documentation%20Status%20of%20the%20ERC32%20Single%20Chip%20i1r1a.pdf

http://spacewire.esa.int/content/Standard/Standard.php