UPTEC F05 072
Examensarbete 20 p
Oktober 2005

A distributed ISA bus network using FPGAs and LVDS links

Johan Johansson

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A distributed ISA bus network using FPGAs and LVDS links

Johan Johansson

This master thesis describes the design of a distributed network with the purpose to evolve into a complete test system to test motherboards and similar units. The network encapsulates signals from an ISA bus and distributes it through a LVDS link. The LVDS network distributing the ISA bus protocol is supposed to run several meters and consist of several slave nodes testing the interfaces of the motherboards. The logic in this design is successfully implemented by the use of Field Programmable Gate Arrays (FPGAs). The advantage of using FPGAs is that they are easily configured and that they support LVDS on chip. LVDS is a differential signalling standard that support high throughput while it consumes low power.
The result of this work is a design that supports the protocols ISA, RS232 and I2C. The nodes in the network also consist of simple digital inputs and outputs. These are directly accessed through the ISA protocol.
The network design is built in a modular manner that makes it very easy to add more registers and protocols. This quality will play an important role when expanding the features of the network. If a protocol has to be added, a module supporting this standard is programmed. Then the module is added to the main logic via internal registers, all accessed via the ISA bus. The strong features in this distributed network design is the flexibility using modules, the support of high speed and the great configurability of the FPGAs.

Handledare: Erik Jansson, Hectronic AB
Ämnesgranskare: Leif Gustafsson, UTH
Examinator: Thomas Nyberg, UTH
ISSN: 1401-5757, UPTEC F05 072
Tryckt av: Ångströmslaboratoriet, Uppsala Universitet

Table of contents

Figures

Tables

Chapter 1 – Introduction

This master thesis describes the implementation of a data network that encapsulates and distributes the ISA bus protocol by the use of a LVDS[I] link. The hardware logic in this network consists of Spartan 3 FPGAs[II].

This work has been performed at Hectronic AB in Uppsala, Sweden. Hectronic AB is the leading Swedish embedded ICT technology supplier and develops small PC-cards. Several of these small PC-cards follow the PC/104 standard and may for example have integrated LCD graphics, serial ports, USB support, disk interface and 10/100Mbit Ethernet. These cards are widely used by the industry, where their small size is a big advantage for built-in applications. Figure 1-1 shows the PC/104 card H6026 Pentium III produced by Hectronic AB.

At different development stages these PC-cards have to be tested. The purpose of the test system is to test a PC-card as thoroughly as possible and generate a diagnostic report. The present test system has shortcomings and has to be improved in several ways. The motivation for this thesis is to improve the present test system.

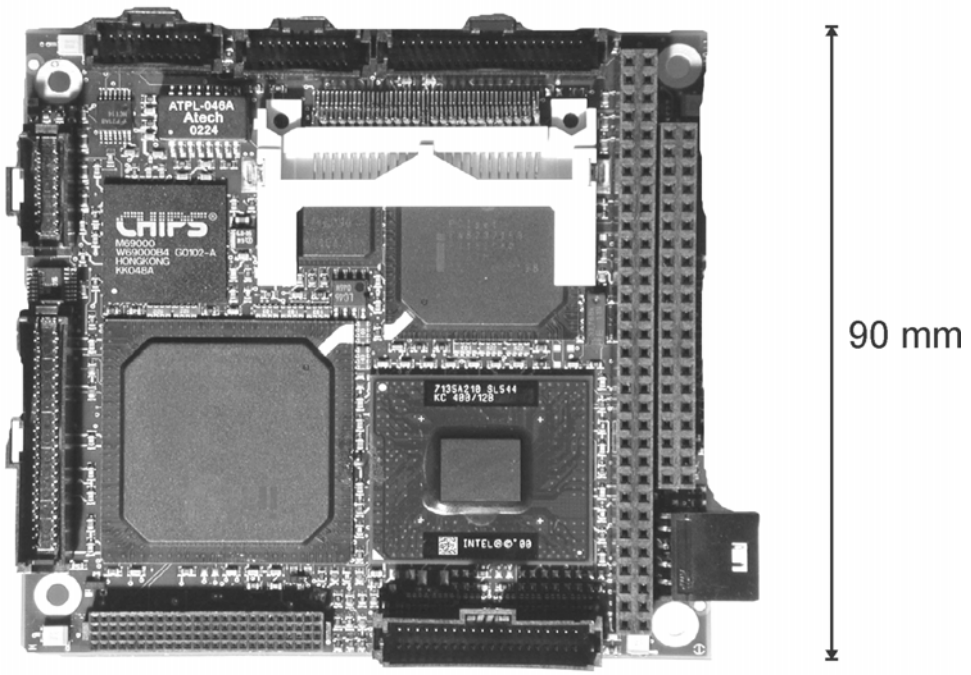


Figure 1-1: PC/104 CPU Board - H6026 Pentium III with the size 90x96 mm.

1.1 The present test system

When the PC-card powers up the BIOS code first runs a series of diagnostic routines called POST[III]. It tests CPU, memory, keyboard, floppy and other basic functions. At the start of each routine the BIOS sends data to port 80[IV] that indicates what routine is being run. These codes are usually also sent via a serial port and if the POST-test stalls, the latest written code to port 80 will show which test went wrong.

[I] Low Voltage Differential Signalling.
[II] Field Progammable Gate Array.
[III] Power On Self Test.
[IV] Port 80 is a hexadecimal data address on the ISA bus.

After the POST-test, an operating system is loaded and started. The operating system will perform a series of peripheral tests of chosen ports and buses on the card. Several different tests may be performed when the operating system has been loaded. The result is reported back to the host computer[I] via the serial port.

1.1.1 Problems with the present test system

One problem with the present test system is that it takes rather long time to load the operating system onto the board so that more advanced tests can be performed. Another delay is when the VGA bus is tested. Here, a LCD display is connected to the bus and the test personnel controls if the LCD display looks ok. If this control would be automatic, even more time could be saved. Also, more specified error reports are desired when an error is found. This is to be able to pinpoint the error source as fast as possible.
All the cases mentioned above contribute to the total test time for every board. If these problems could be solved or minimized, test time would be significantly reduced and the test costs would be lower.

1.2 Requirements for a new test system

A small group of employees were put together to discuss a new test system. The group determined some guideline properties needed in the test system. The directions are presented in this section.

1.2.1 The properties of the test system

The test system should have the following properties:

- **Test of the test objects[II] peripherals:** The buses and ports of the test object should be able to be connected to the test system so that communication tests of these peripherals can be performed.
- **Fast uploading of data to the test object:** To save test time, the uploading of the operating system from the host computer to the test object has to be fast. 1 Mbyte/s and faster was estimated to be sufficient.
- **Flexibility:** The test system should be able to be reconfigured to support other peripheral tests to update possible programming errors in the existing code. The test system should also be able to be expanded to test several test objects all connected to the system at the same time.
- **Distance between two network nodes:** Because several test objects might have to be tested at the same time the network has to support distances up to about 2 meters.
- **Connection wiring:** To connect the network a small bus of a maximum width of five wires are required. This is to make the test system able to be modified into other small future applications.
- **Supply voltage:** The five available wires should carry supply voltage for the distributed system nodes as well. This is not necessary for the test system, but is required in some applications planned for the future.
- **Cost:** The logic chosen should be low cost.

[I] The host computer is the main computer coordinating the test.
[II] The test object is the fabricated circuit board that is to be tested.

1.2.2 Different test situations

There are several test situations to be considered. These are summarised below:

- **Lab test:** During the design and development of a card, different tests of the card should be performed.
- **Function test:** This test is made after the final assembly of the cards. If the card passes this test, it should be fully functional.
- **Temp test:** Here the card is tested at different temperatures. The card functionality is tested while it is exposed to different temperatures in a special designed temperature box. To save testing time several cards need to be tested in the box at the same time. This demands the test system to support several cards to be connected simultaneously.
- **Field test:** If an error occurs after delivery, it is a big advantage if the card could be tested "in the field" by the customer.

1.2.3 Test system architecture

The test system should consist of one host computer running the main test program coordinating the tests. The host computer should be able to upload program code to the test object(s) via the test system. The host computer should also be able to communicate via the test system to the peripherals of the test object.
After performing the test, it would be an advantage if the host computer could report the result to a data server logging all the tests.

1.3 The aim of this project

The aim of this project is mainly to develop a distributed system suitable to handle the communication between the host computer and the test object(s) and its peripherals. The problems with the present system and the requirements of a new system should be considered in the new design.

1.4 Method description

The implementation of the logic is to be done on a suitable CPLD[I]/FPGA. The proper CPLD/FPGA has to be chosen and suitable programming technologies in the device have to be investigated. The programming language used to design the logic is VHDL.
Because a new distributed network has to be designed, suitable system topologies and channel configurations has to be investigated as well.

[I] Complex Programmable Logic Device.

Chapter 2 – Background

This chapter introduces the technologies used in the distributed network design. It also forms a basis for the decisions made in section 3.1. This chapter could be used as a reference when different implementations, using these technologies, are described in other chapters.

2.1 The OSI seven-layer model

OSI[I] is a standard reference model of the communication flow between two end users in a network. The International Organization for Standardization developed OSI in 1984 and it describes a framework for implementing network protocols in seven layers. The OSI seven-layer model is a part of this standard and is a useful reference model for describing and designing computer networks.
The OSI seven-layer model consists of seven layers that data has to go through to travel from one user to another. Control is passed from one layer to the next and each layer uses the functions of the layer below and only exports functionality to the layer above. The data-flow starts from the application layer in one station and proceeds to the bottom layer. There it passes over a channel and travels back up in the layer hierarchy in another station. The seven layers are shown in Figure 2-1 and are briefly described below [1]:

- **Layer 7 – Application:** This layer interacts with the application and supplies network related activities such as file transfers.
- **Layer 6 – Presentation:** This layer is usually part of an operation system and converts incoming and outgoing data from one presentation format to another. For example it could convert a data steam into a popup window displaying some information.
- **Layer 5 – Session:** Establishes, maintains and ends communication with the accessed device.
- **Layer 4 – Transport:** Ensures a complete data transfer. For example by performing error checking and controlling that all the packages has arrived.
- **Layer 3 – Network:** Network data routing and flow control. The way the data is to be sent is determined in this layer.
- **Layer 2 – Data link:** Describes the logical organisation of data bits transmitted, such as framing, addressing and error checking. Error checking may occur in a higher layer as well.
- **Layer 1 – Physical:** Describes the physical hardware properties of the channel such as connectors, voltage levels and timing.

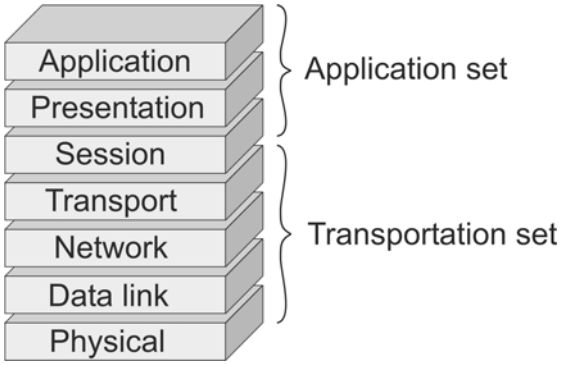


Figure 2-1: The seven-layer OSI reference model.

[I] Open System Interconnection.

2.2 Backplane architecture

A backplane is used to join several peripherals together. The VME, PCI and ISA buses are examples of protocols using backplanes. A backplane set-up could for example be a microprocessor communicating with memory, keyboard, mouse and soundcard devices. The need for backplane performance has always been high. The maximum number of cards connected to one segment[I], data path width, hot-swap[II] capabilities and low cost are concerns to have in mind when designing a high performing backplane. In this section, some important design methodologies are explained [2].

2.2.1 Connectivity

Backplane connectivity refers to how the drivers and receivers are connected. There are two commonly used transmission schemes today, serial and parallel.

2.2.1.1 Parallel design

In this design, the bits are sent from the driver in parallel. All the information sent in the backplane is configured either as multipoint or multidrop [2].
The advantages of a parallel design are summarised below:

- Individual lines can be used as control signals with fast reaction times.
- High data throughput can be achieved with low signal speed.
- No extra logic is needed to serialise and deserialise the data blocks. This means no time delay.

The disadvantages of a parallel design are:

- Many data traces are needed. This means lots of space on the circuit board and high costs.
- The signal skew between the signal lines has to be matched.
- Impractical when going between units at long distances.

2.2.1.2 Serial design

In a serial design the data is sent in a serial bit-stream using one single transmission line. All the communication is made point-to-point [2].
The advantages of a serial design are summarised below:

- No signal skew problem between the signal lines. Only the skew in the individual pairs in a differential transmission line has to be matched.
- The transmission speeds in different data lines are adjustable giving support for longer cable lengths.
- Only a few signal traces are needed which significantly reduces the space occupied on the circuit board.

The disadvantages of a serial design are:

- A delay is introduced to serialise and deserialise data blocks.
- Serial devices are usually more expensive than parallel drivers.
- Higher speed is often needed to compensate for fewer lines. This demands better impedance matching and trace layout.

[I] The segment is the continuous traces on the circuit board that joins the peripherals.
[II] Hot-swap is the ability to remove and plug-in a card on the backplane while having the components turned on.

2.2.2 Timing architecture

The timing architecture refers to how the bits in a data block are synchronised. There are many ways to do this and timing architecture plays a big role in the resulting data-rate.

2.2.2.1 Synchronous clock

This design is a synchronous timing architecture and has only one clock source and the data is synchronised with this clock. The routing of the clock is done so that all the peripheral cards receive the clock at the same time as shown in Figure 2-2. Because the data lines are driven by a source in every peripheral card, the data paths can not be adjusted to solve for data delay. This delay issue makes this architecture not suitable for high data-rates.
Careful layout to minimise the clock skew is critical for this type of design. When this is done, the platform will be robust and effective.

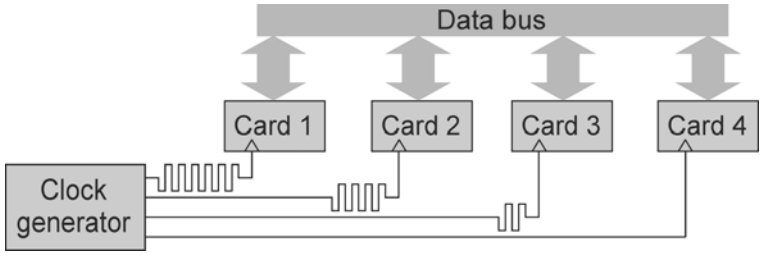


Figure 2-2: Synchronous timing architecture using only one clock generator.

2.2.2.2 Source synchronous clock

In this design, every source generates a clock that travels in parallel with the data to the receiver. This is shown in Figure 2-3. Note that only the clock signal from card 1 is shown even though all other cards has clock lines to all other receiving cards. The clock and the data line have to be the same length and have the same loading. This is to ensure that the data and the clock signals will arrive at the receiver with no skew between each other. Because the data and clock signals are matched to minimise the skew, this design will support much higher data speeds compared to the synchronous clock design. The design also has potential for bigger networks because the skew between clock and data does not increase as much as for the synchronous clock design. The drawback is that many clock signals are needed – one set for every pair of data-linked devices. This will consume much space on the circuit board as the number of cards increase.

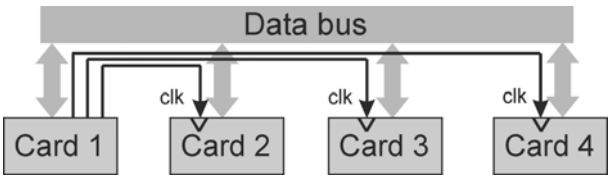


Figure 2-3: Source synchronous clock design (showing clock signals from card 1 only).

2.2.2.3 Asynchronous system

An asynchronous system does not use a clock. The communication between devices is made by the use of control signals also called “handshaking”. For example, the transmitter uses one signal to tell the receiver that data is ready on the bus. When the receiver has sampled the data it uses one acknowledge signal to tell the receiver that data was received.
Often asynchronous designs are very solution specific and are usually designed uniquely for each implementation. One advantage of this design is that it does not have clock timing problems. It is also more power efficient because it does not have a constantly running clock signal. One drawback is that it is only efficient over short distances.

2.2.2.4 Embedded clock

Embedded clock or CDR[I] is a design method that has become more popular in recent years. In this design you send only one bit stream and the receiver unfold the data bits without the use of a parallel clock signal sent with the data.
The method needs the data to be encoded in a special way so the clock is included in the bit stream. There are several ways to do this. Some of the codes used are briefly explained below [3]:

- **Manchester encoding:** This encoding is commonly used and one application is for example the data coding in Ethernet. The method encapsulates the clock in the data by sending the zero-bit as a positive signal edge and a one-bit as a negative signal edge. In this way the receiver can easily figure out the centre of each bit. It is at every edge of the received signal.

- **8B/10B:** Gigabit Ethernet and fibre channel use this coding technique. It is a coding technique, which for every 8 bits of data sends a 10-bit code. This code has the property consisting of either 4 zeros and 6 ones, 5 zeros and 5 ones or 6 zeros and 4 ones. This will give the code the ability to send equally many zeros as ones to generate a DC-balanced signal. The clock can be deduced from this code and it guarantees that only a maximum of five zeros or ones are sent in a row. Usually this code is implemented in hardware using look-up-tables (LUTs).

Extraction of the clock from the Manchester code embedded clock signal is done by the use of a DPLL[II]. Because the signal contains a high and constant rate of bit transitions, the DPLL can lock on to this frequency and generate the clock signal.
To extract the clock from 8B/10B signal a DPLL is also used. By tracking the changes in the bit pattern a clock signal from the DPLL can be generated in the same way as for Manchester encoding.
To use an embedded clock design a constant flow of bit-patterns to lock on, has to be sent. If the DPLL has to reacquire lock over and over again, this will significantly slow the bus down. Sending bit patterns that the DPLL could lock on to, even if there is no actual information contained in them, solves the problem.

2.2.2.5 Signal path delay

A signal propagates in a twisted pair cable with a speed of about $v = 0.6 \cdot c = 0.6 \cdot 3 \cdot 10^8 = 1.8 \cdot 10^8 \ m/s$ [4]. For a clock signal with a frequency f = 100 MHz the wavelength would be $\lambda = \frac{v}{f} = \frac{1.8 \cdot 10^8}{100 \cdot 10^6} = 1.8 \ m$. If we estimate that, a data signal is allowed to have a maximal skew of 1/20 of a wavelength to be received with high accuracy. Then the maximum length difference between two twisted pairs could be $\frac{1.8 \ m}{20} = 0.09 \ m = 90 \ mm$.
It thus follow that the maximum length difference allowed for a 200 MHz clock transmitted in a twisted pair would be about 45 mm.
These calculations show the maximal skew allowed in the circuit design. All cables, connectors and traces have to be included in the total channel length.

[I] Clock Data Recovery.
[II] Digital Phase Locked Loop.

2.2.3 Data distribution topologies

Data can be distributed in many ways. The three most common topologies are summarised below:

- **Point-to-point:** This topology has the highest data speed and is the most common and basic bus topology. The transmission of data is over one channel between two nodes. If the nodes consist of transceivers, the configuration is called half-duplex point-to-point. If the data is transmitted only in one direction, from the transmitter to the receiver, the configuration is called simplex point-to-point.
- **Multidrop:** In this topology there is one transmitter and several receivers. The communication is made over one single channel.
- **Multipoint:** This topology consists of several transceivers sharing one channel. Each transceiver is able to communicate to either of the other transceivers. The possibility that several of the transceivers transmit data at the same time has to be taken care of. Only one transmitter is allowed to transmit at a time.

More information about the differential configuration of these topologies is found in section 2.5 describing LVDS.

2.2.4 Single-ended versus differential signalling

A single-ended channel consists of only one wire or trace that carries the signal. The signal level is transmitted and received using one common ground. The advantages of single-ended signalling are easy implementation, low cost and few components. The disadvantage is its sensitivity to noise and that it can not travel far due to degradation.
A differential channel uses two wires or traces – one signal and its complement. It is the difference between the signal levels that carry the information. The advantages of differential signalling are that it has high noise rejection and supports long cable lines. The disadvantages are that it is harder to implement and the components needed cost more than for single-ended designs. More information can be found in section 2.5 describing LVDS.
All three topologies point-to-point, multidrop and multipoint can be implemented with either differential or single-ended signalling.

2.2.5 Power over Ethernet

Power over Ethernet (PoE) is a technology that makes it possible to supply power to a device using the same wires as for transmitting the data. The data is sent using a differential standard and two or four differential pairs. Power over Ethernet provides as much as 13 watts (using 48 volts) to the devices [5]. Adding a DC voltage between two differential pairs transmits the power. At the receiver the DC voltage is used for power supply and the low differential signals are used to transmit the data.

2.3 FPGA technology

To communicate with the host computer and to distribute the data through the test system, the Xilinx FPGA Spartan 3 was chosen. The code for the Spartan 3 chip was developed and compiled using the Xilinx ISE WebPACK 7.1i software. The design code is usually either programmed using VHDL directly, pregenerated by IP-cores[I] or with the schematic editor in the ISE software. The architecture of the FPGA, the design flow using the software, the VHDL language and some high-speed applications are explained in this chapter.

2.3.1 FPGA introduction

FPGA stands for Field Programmable Gate Array and is a programmable chip that has many advantages over ordinary logic [6]. The chip is usually equipped with hundreds of thousands of logic gates that can be connected to each other. By programming the connection between the logic gates very complex logic can be created. A complete high performance CPU may for example be implemented in a small part of a chip.

The FPGA chip gives the opportunity to program logic that runs in parallel. In an ordinary microprocessor, the CPU processes the programmed instructions in the memory sequentially. This is a disadvantage if the problem-solving algorithm is able to run in parallel. In a FPGA chip thousands of logic structures may run completely separated and their processed data may be joined together in a final stage. By programming the logic in parallel, the data processing performance can be significantly increased.

The logic in the FPGA is also highly flexible and does not occupy more space than needed. The static structure of an ordinary 32bit ALU forces it to operate using 32 bits data registers even if only a few bits are actually carrying information. The flexible structure of an FPGA makes it possible for the logic not to be bigger than necessary. For example, if the logic operates on 5 bits of data only a 5-bit register is used.

The FPGA chip has many built-in features such as clock dividers, phase shifters, multipliers and support of several different I/O-standards.

The parallelism and the flexible logic bring the FPGA chip tremendous speed and applicability. In image-processing or data encryption applications, the speed may be 100 times faster for an FPGA than for a CPU.

2.3.2 The Spartan 3 FPGA

The Xilinx Spartan 3 XC3S200 FPGA chip [7] is chosen for this project because it has many useful features and is low cost. Some of the Spartan 3 XC3S200 features are presented below.

- **Distributed logic:** The FPGA holds 4,320 logic cells and 200,000 system gates.
- **Embedded multipliers:** The FPGA has 12 embedded multipliers capable of multiplying 18-bit wide registers.
- **Block RAM (BRAM):** The FPGA has a total of 216 Kbits of Block RAM.
- **Digital clock managers (DCMs):** Four DCMs are distributed in the chip to support multiple system clocks. They also support clock skew elimination, high-resolution phase shifting and frequency synthesis.
- **I/O banks:** Eight I/O banks support 24 I/O standards including LVDS.

[I] Intellectual Property Cores.

2.3.2.1 Internal logic architecture

Figure 2-4 below shows the basic internal structure of the Spartan 3 FPGA.
The Configurable Logic Blocks (CLBs) make up the main logic resource for the FPGA. Every CLB consists of four Slices for implementing synchronous as well as combinatorial circuits. The Slice consists of the basic logical building blocks such as arithmetic gates, storage elements, carry logic and lockup tables (LUTs).
The Input/Output Block (IOB) is the link between the internal logic in the FPGA and the I/O pin. Each IOB is bi-directionally programmable and supports several I/O standards.
By combining the logic elements within the slices in the CLB and by combining the CLBs with BRAM, IOBs and multipliers, very complex logic may be constructed.

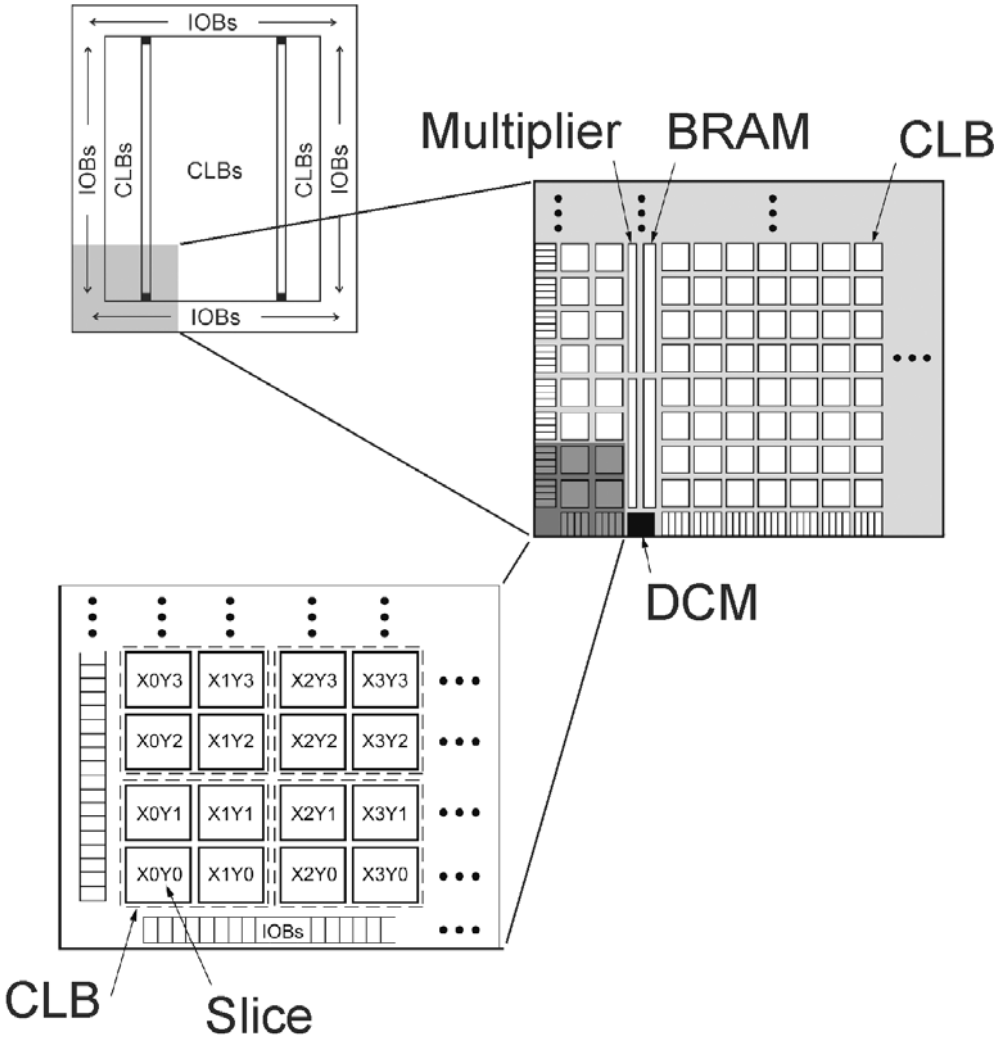


Figure 2-4: The internal logical structure in the Spartan 3 FPGA.

2.3.2.2 The global clock network

Eight global clock lines called GCLK0 - GCLK7 are distributed in the device. This network connects clock signals from the input pad to the internal logic. The DCMs are also connected with this network and are able to generate new phase shifted or multiplied clock signals driving different internal logic.

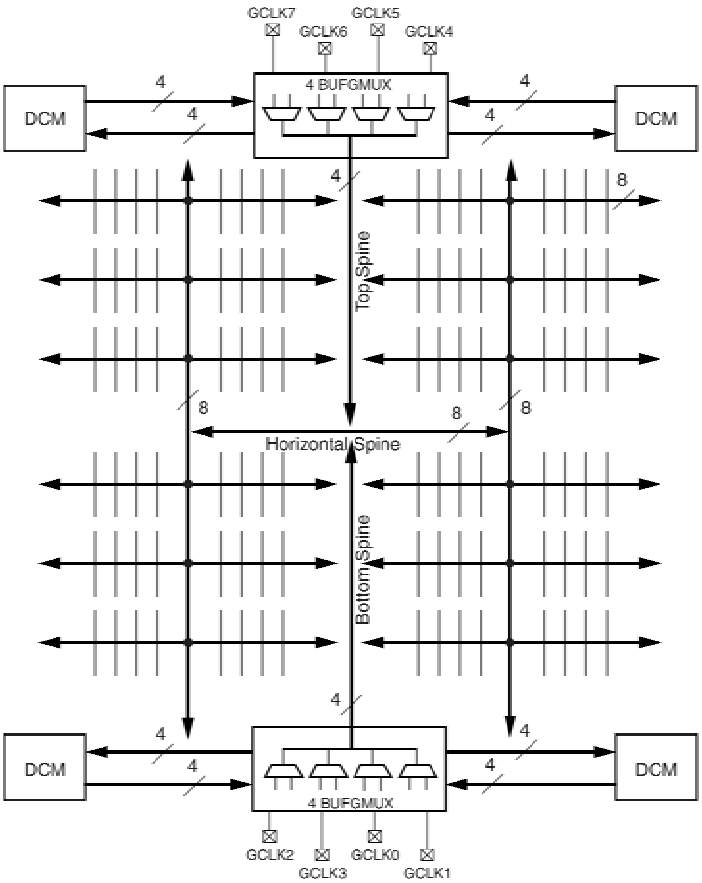


Figure 2-5: The distributed clock network in the Spartan 3 FPGA.

2.3.2.3 The Digital Clock Manager (DCM)

The DCM is able to generate a wide range of clock frequencies and is able to phase-shift the output signal with respect to the input signal. As shown in Figure 2-5 [8] only four clock signals may be distributed from the DCM using the high quality global clock network.

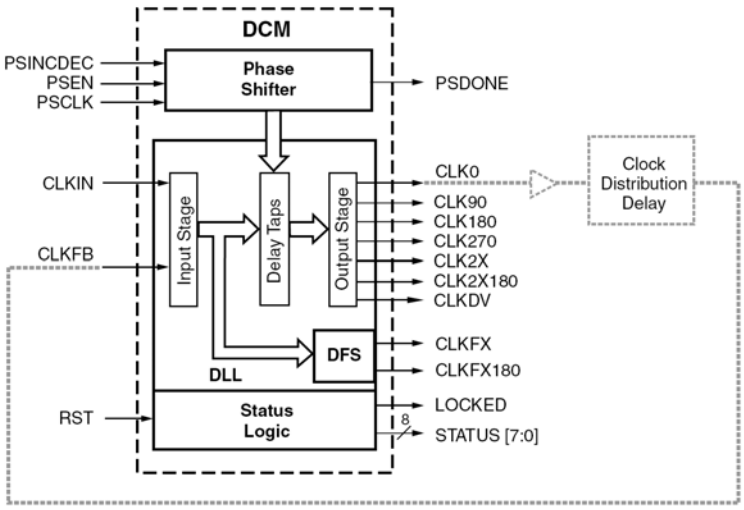


Figure 2-6: The internal logical structure of the DCM in the Spartan 3 FPGA.

In Figure 2-6 the DCM logical structure is shown. Some of the signals are explained below:

- **RST, Locked and CLKFB signals:** RST is the reset signal and forces the DCM to reacquire the lock on the input signal. The locked signal is asserted when the DCM has locked on the input signal and the output signals are accurate. The CLKFB signal is for feedback so that the output signals gets the chosen phase shift.
- **270, 180, 90 and 0 degrees phase shift:** The DCM can be configured to produce four simultaneous output clocks phase shifted 270, 180, 90 and 0 degrees with respect to the input clock. These signals may be used in data transfers using DDR[I] with a separate clock line (for more information see the DDR section below). The drawback of this configuration is that the clock rate cannot be multiplied. If the input clock is 50 MHz the maximum data rate is 100 Mbps using DDR.
- **CLKFX and CLKFX180 with arbitrary phase shift:** If CLKFX and CLKFX180 are used; an input clock signal faster than 48 MHz may generate a clock signal and its inverse of a maximum frequency of 280 MHz. An arbitrary phase shift can be applied as well.
- **CLK2X and CLK2X180 with arbitrary phase shift:** Works as CLKFX and CLKFX180 above but are only available up to a maximum frequency of 210 MHz.
- **CLKDV with arbitrary phase shift:** CLKDV is used to divide the clock signal. The division value can be chosen in several stages from 1.5 to 16. An arbitrary phase shift can be applied as well.

DCM lock time

The time it takes for the DCM to lock on to a clock signal and generate the output clocks is essential to know for some special designs. If the master node in the distributed network has to reacquire the lock on the clock signal sent from the slave at each data transmission, this time will contribute to the resulting data transfer rate. In some applications, the data is sent in parallel with the clock on two different wires. If the clock frequency is higher than about 100 MHz, the clock might disturb other component on the circuit board [9]. To solve this problem a clock at a lower frequency is transmitted and then multiplied in a DCM to achieve the wanted clock frequency. The data is then sampled at the new higher clock rate.
The Figure 2-7 shows the functional simulation of a DCM with an input frequency of 50 MHz and two outputs of 200 MHz where one of them is shifted 180 degrees. The Figure 2-8 shows the post-place and route simulation of the same design. If one considers only the functional simulation, the generated clk0 and clk4x signals are in phase with the input signal clkin. If this would be accurate for the final hardware communication, the method of sending a slower sampling clock would work fine. If one consider the post-place and route simulation this would not work because the sent clock clkin and the generated clock signals are not in phase.
According to [10] the maximum lock time at an input clock frequency of 50 MHz is about 1 ms.

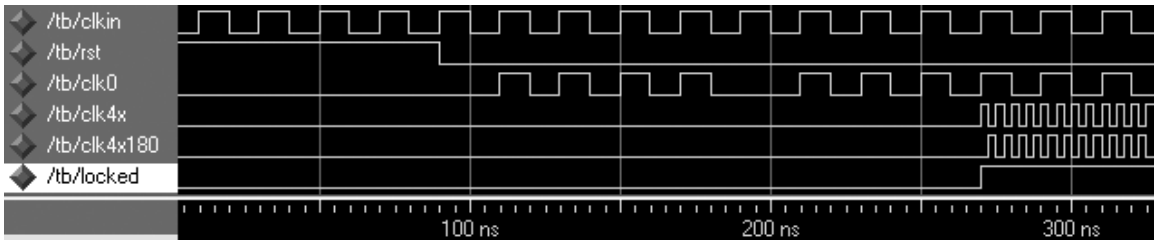


Figure 2-7: The functional simulation of a DCM. 50 MHz input frequency and 200 MHz output frequency.

[I] Double Data Rate.

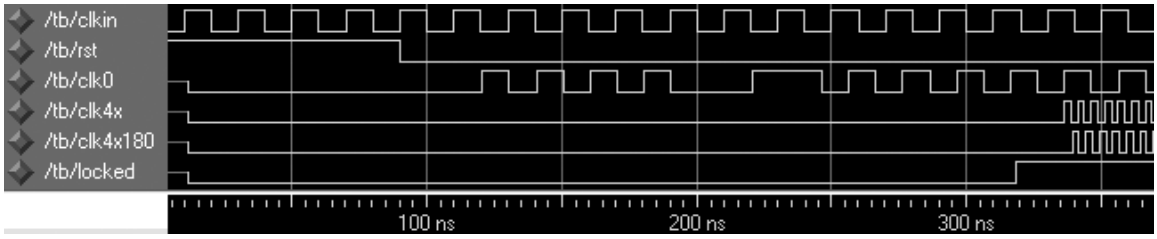


Figure 2-8: The post-place and route simulation of a DCM. 50 MHz input frequency and 200 MHz output frequency.

2.3.2.4 High speed data communication

The ability to communicate with high speed is essential in many distributed test networks. If operating systems and test programs has to be loaded from the host computer to the test objects, much testing time will be gained in having a fast network. FPGAs have many ways to generate high-speed data and clock output signals.

Double Data Rate (DDR)

DDR is a method to send data at twice the rate of the clock. The method uses both the positive and the negative flank of the clock. A DDR MUX[I] according to the Figure 2-9 switches the data from two flip-flops. The flip-flops triggers on clock signals phase shifted 180 degrees from each other. For the best result, a DCM is used to generate the two clock signals. An inverter is sometimes also used to generate the 180 degrees phase shift of the second clock signal.

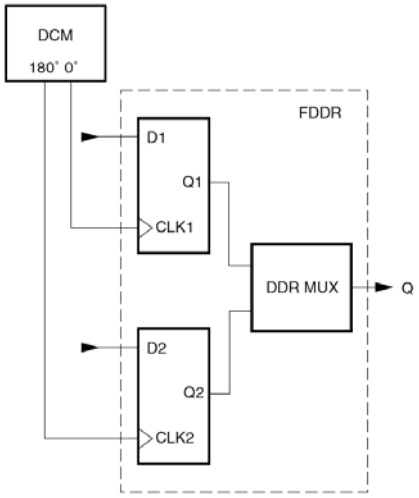


Figure 2-9: The DDR component consisting of two flip-flops and one DDR MUX.

2.3.2.5 Serialising data at high frequencies

The logic serialising the data before the DDR flip-flops has to be very well structured and planned. The Spartan 3 device is only of speed grade -4 and this creates problems when the logic has to be run at high frequencies. There are two commonly used methods to prepare data. One is using two shift registers and the other one uses two multiplexers. Both techniques are explained below.

[I] Double Data Rate Multiplexer.

Preparing data using multiplexers

The design in Figure 2-10 shows how data is serialised and sent along with clock and framing signals [11]. The framing signal is in this example used to signal to the receiver when the data package starts and stops. The output clock is sent at four times the internal clock frequency. Because the use of DDR the data bits will be sent two times the clock output frequency and eight times the internal clock frequency.

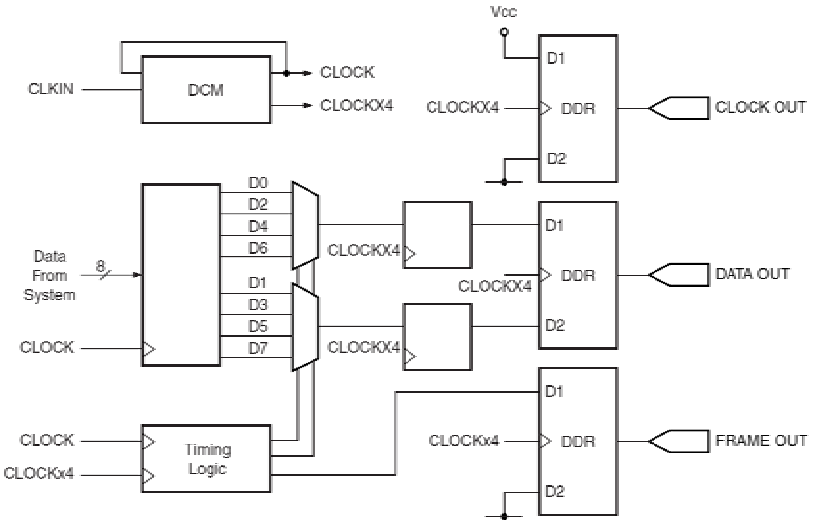


Figure 2-10: This schematic shows two muxes serialising the data to the DDR component.

The logic above transmits 8 bits at a time and for this purpose the MUX arrangement works fine. If more bits are to be sent at a time, the maximum frequency allowed in the design will be lower. This is because bigger MUXES have to be used.

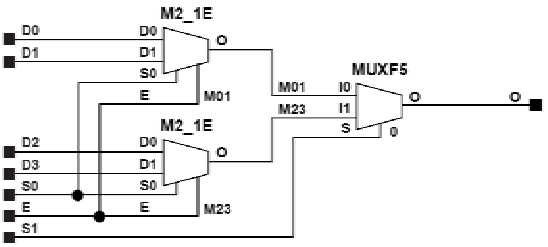


Figure 2-11: Library Component M4_1E (4:1 MUX) implemented in a Spartan 3 FPGA.

Figure 2-11 shows a 4:1 MUX implemented in a Spartan 3 FPGA [12]. The 4:1 MUX is built of three 2:1 MUXES and each data signal has to pass two MUXES. If for example 32 bits of data are to be sent, two 16:1 MUXES are used. This means that the data signal has to pass four MUXES.
The conclusion is that it is not appropriate to serialise data using MUXES if many bits at a time at high bit rates are to be transmitted.

Preparing data using shift registers

The design in Figure 2-12 shows how data is serialised using two shift registers [13]. The shift registers (named PISO) take turn in presenting the data to the DDR module named DDRFD. The data and the clock are in this design sent in parallel using two LVDS pairs.

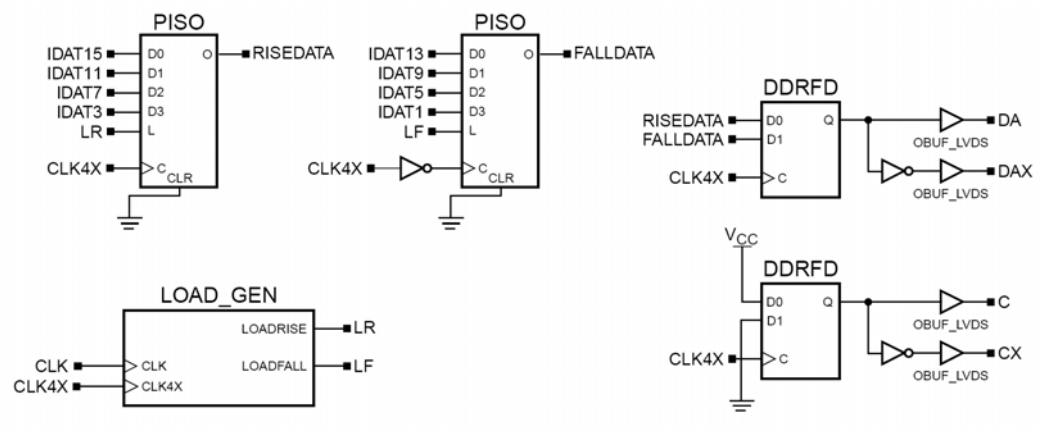


Figure 2-12: This schematic shows two shift registers serialising the data to the DDR component.

At high frequencies the shift registers handles large data blocks more efficient than the MUXES in Figure 2-10 above. The shift register is implemented using flip-flops as shown in Figure 2-13.

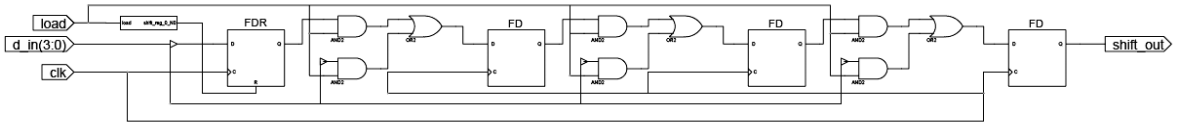


Figure 2-13: A 4-bit shift register implemented in a Spartan 3 FPGA.

Because of the pipeline structure of the logic, the maximum frequency limit is not dependent of the number of bits in the input block.
The conclusion is that shift registers are the right design implementation to serialise big blocks of data at high frequencies.

2.3.3 VHDL

VHDL is a very powerful hardware description language. It is used to specify, verify and construct electronic logic circuits. VHDL was demanded by the US Department of Defence in the beginning of the 1980s to describe the behaviour of the ASIC chips in their equipment. Later that decade the language was used to describe circuit models in simulation tools. A few years later the VHDL-standard was used in electronic constructions.
The advantages of VHDL over traditional schematic circuit designs are shorter development time and easier maintenance [14]. Because VHDL is a standardised language, it is easy to transfer the code between different development tools. Unfortunately, VHDL is not standardised for construction (synthesis). This means that it is harder to transfer code for construction between different development tools. If the construction code is supposed to be transferred, the code has to be written in a common agreed manner accepted by the different tools. Luckily, this is easily done for the most frequent designs.
Another problem with transferring the code is that many components only exist in the used device. This code may not be transferred to a device not supporting the component. Some example of such components in the Spartan 3 FPGA is the DCM, the DDR-block and the LVDS driver.

2.3.4 Schematic capture

Many development tools include a schematic editor. This editor is a very useful tool when used together with modules written in VHDL. Even though a complete design may be constructed only using the schematic editor, it is often not recommended. Some designs may be very hard to implement using the schematic editor but very easily implemented using VHDL.

A good reason to use the schematic editor is the good overview of the design. Some components like the DCM can be implemented here because they do not benefit as much from the VHDL transfer ability. Small parts of a design, not using any device specific components benefit, a lot on the other hand, by being written in VHDL. The different modules are easily combined in a schematic editor and you get a good overview of the complete design. If the modules are joined only in VHDL, the whole design is often harder to grasp.

2.3.5 Xilinx ISE 7.1i – design flow

Xilinx ISE 7.1i is an all in one package that integrates the tools needed through all the common design steps [15]. Third party tools can also be integrated for more advance simulation and synthesis. In this section, the design flow shown in Figure 2-14 below will be explained.

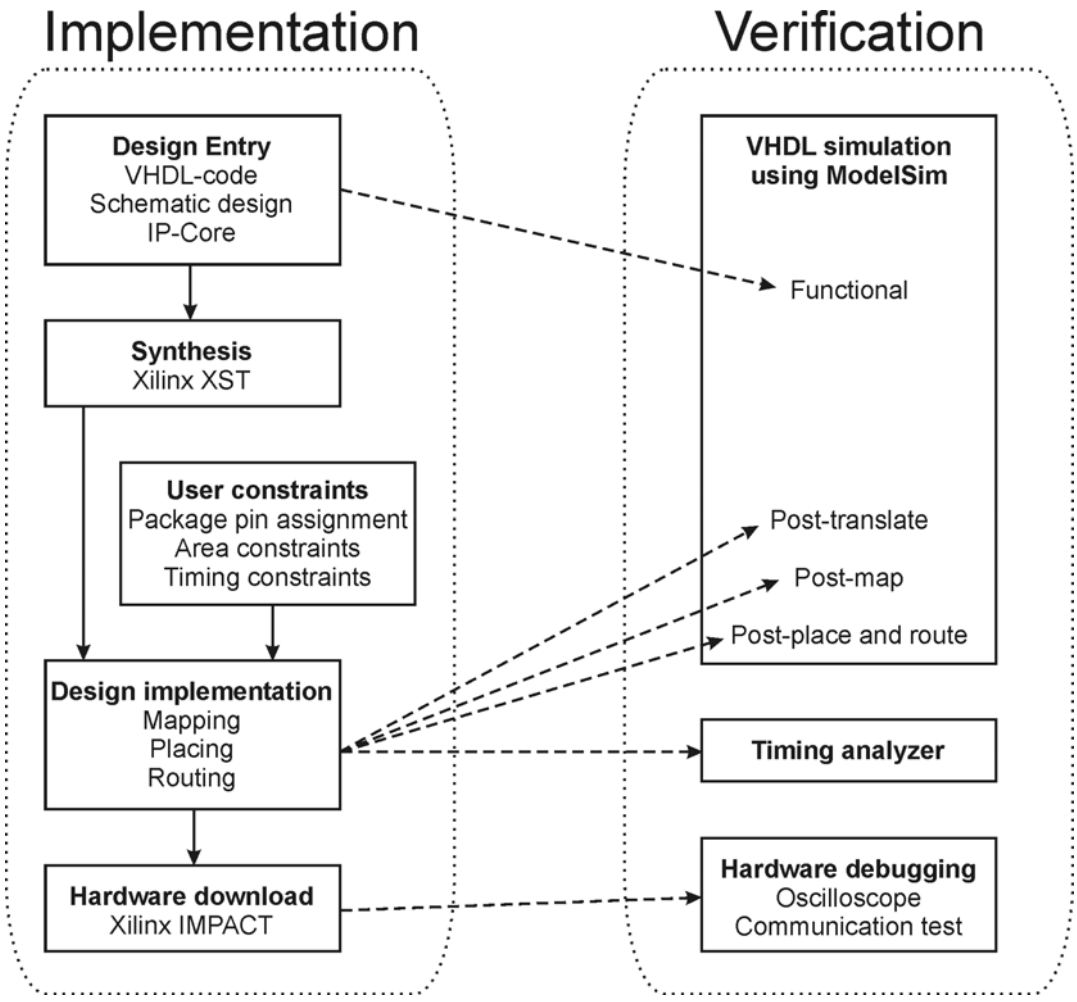


Figure 2-14: The diagram shows the Xilinx ISE 7.1i design flow.

2.3.5.1 Design Entry

At this stage, the designer describes the design. This is done usually in the HDL editor and the schematic editor. Already invented designs called Intellectual Property (IP) Cores are also put into the design here. The VHDL code is written in the HDL-editor or in an ordinary text editor. VHDL code is also generated from the schematic design in the schematic editor.

2.3.5.2 Designs synthesis

The synthesis tool uses the described design in the VHDL-files to decide which components to use and to generate the net list. The net list describes how the components are interconnected. It is important that the synthesis tool generates a good design. Xilinx ISE provides its own synthesis engine (XST). For an even better result a third party synthesis engine is also possible to integrate in ISE at this stage.

2.3.5.3 Design implementation

- **Mapping:** Uses the logic description from the synthesis and maps the logic to the logic cells, I/O cells and other components in the FPGA.
- **Placement:** Decides where the components from the mapping should be placed in the FPGA.
- **Routing:** Decides how the placed components should be interconnected.

2.3.5.4 Design verification

After place and route, timing analysis can be made in the timing analyser tool. Here you can see if your timing constraints have been met. The timing analyser also displays pad-to-pad delays.
The design can also be simulated in different stages. After the design entry stage a behavioural simulation can be done. Here you see how the code would work if no wire and components delay existed.
After place and route the most complete simulation can be done. Here the construction is simulated with component and wire delays. Because of the wire delays, the final layout strategy is very important. That is why modern FPGA designs are constrained in the place and route level rather than in the logic block level.
A behavioural model simulator is already included in the ISE package. For more advanced simulations a third party simulation tool has to be used. In this project the Mentor Graphics ModelSim XE III/Starter 6.0a is used.

2.3.5.5 Hardware implementation

After place and route a programming file with the configuration bits for the FPGA is generated. The Xilinx IMPACT tool is used to transmit the bits to the FPGA. Because the FPGA does not retain data after power shut down, a Flash EPROM can be used. In the Spartan 3 Starter kit development card and on the Hectronic H4070 heat test card, both the Flash EPROM and the FPGA is programmed via a JTAG-chain [16]. The data from the IMPACT tool is sent to the JTAG-chain via the parallel port of the PC.

2.4 The ISA bus

2.4.1 Introduction

The ISA bus is a very old computer bus that has been very common in the PC world. IBM developed it in 1981 and in 1984 it was expanded into a 16-bit bus. Some advantages using the bus are that it is very easy to implement and that there are many peripherals supporting it. The disadvantage is that the bus is slow. Even though the bus is 16 data bits wide, it only has a maximum speed of 15.9 Mbytes/sec in 16-bit mode and 7.9 Mbytes/sec in 8-bit mode. But if the application does not demand speed, the ISA bus might be the best choice.
The ISA bus is still used in many applications such as in digital and analogue I/O cards and in the famous PC/104 cards.

2.4.2 ISA bus device communication

The ISA bus is asynchronous and can address I/O and memory devices. The bus also supports interrupts and DMA.
In this section the communication between the microprocessor and the I/O device is briefly explained [17].
Usually a microprocessor is the master in the bus structure. The I/O devices are slaves. In an ordinary I/O device access the master directs the communication. The slave has the ability to shorten or lengthen the bus cycle. The following signal lines are used in this project:

- **BCLK:** This is the bus clock usually at a frequency of 8.3 MHz. In other words, there are 125 ns between two successive rising edges on the clock signal.
- **SBHE#:** When SBHE# is asserted the microprocessor is doing a 16-bit access. Otherwise, it is doing an 8-bit access.
- **SA(19:0):** 20 address bits are used to address a memory device. Only 10 bits are used to address an I/O device.
- **BALE:** When this signal goes low it tells the device that the address bits SA(19:0) are latched and can be read.
- **IO16#:** When this signal is asserted the device tells the microprocessor that it is capable of handling a 16-bit access.
- **IORC#:** When asserted it signals to the device that a read access is being made.
- **IOWC#:** When asserted it signals to the device that a write access is being made.
- **NOWS#:** This signal tells the microprocessor that no wait state is needed.
- **CHRDY:** This signal tells the microprocessor to insert wait states.
- **SD(15:0):** These are the data bits. Only the lower 8 bits are used in an 8-bit access.

The 8-bit access is explained in the sequential list below and the numbers in Figure 2-15 corresponds to the numbers in the list. A more detailed and complete description of the access can be found elsewhere [17].

1. The microprocessor starts by putting the address to be read from or written to on the address lines SA(19:0). The SBHE# state is also asserted at this time.
2. The BALE signal is deasserted by the microprocessor.
3. The microprocessor asserts the IORC# for a read instruction or the IOWC# for a write instruction.
4. The IO16# is sampled deasserted.
5. If NOWS# and CHRDY is sampled asserted by the microprocessor the bus cycle end on the next rising edge of BCLK. At this time, the device can read the data on the bus if the access is a write cycle.
6. If CHRDY is sampled deasserted by the microprocessor, it will insert wait states to postpone the ending of the bus cycle till the CHRDY is sampled asserted.
7. If CHRDY is not sampled deasserted and NOWS is not sampled asserted the bus cycle will be terminated after four wait states.
8. If the access is a read cycle, the microprocessor will read the data put on the bus by the device. This is done at the rising edge of BCLK at the end of the bus cycle.
9. The microprocessor asserts BALE and a new bus cycle can be started at the next rising edge of BCLK.

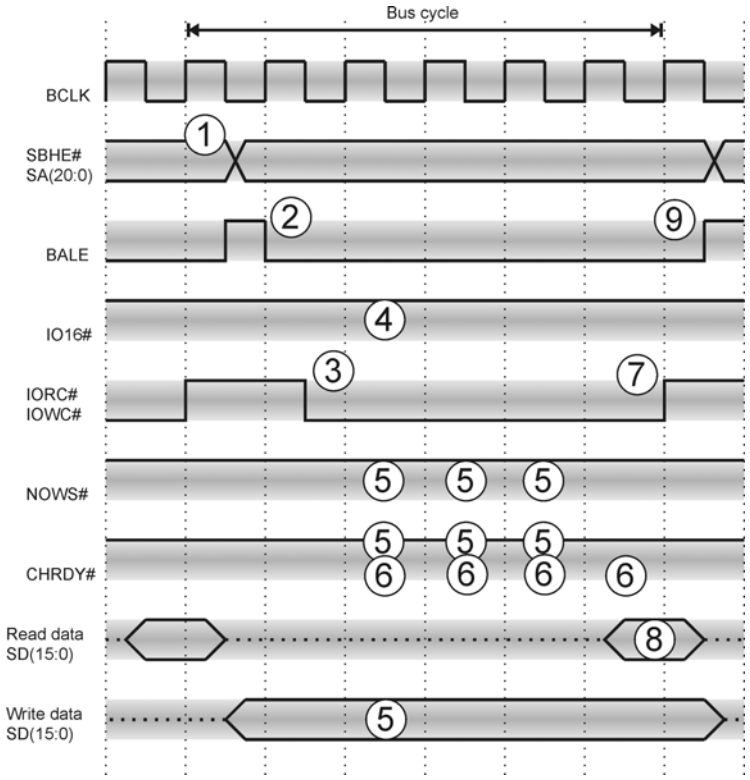


Figure 2-15: ISA bus access to a standard 8-bit I/O device.

The 16-bit access is made as shown in Figure 2-16. The difference from the 8-bit access is that all of the 16 data bits are used and the bus cycle is normally terminated earlier (if CHRDY is not sampled deasserted). NOWS# is never used in a 16-bit access and the IO16# signal is sampled asserted instead of deasserted [17].

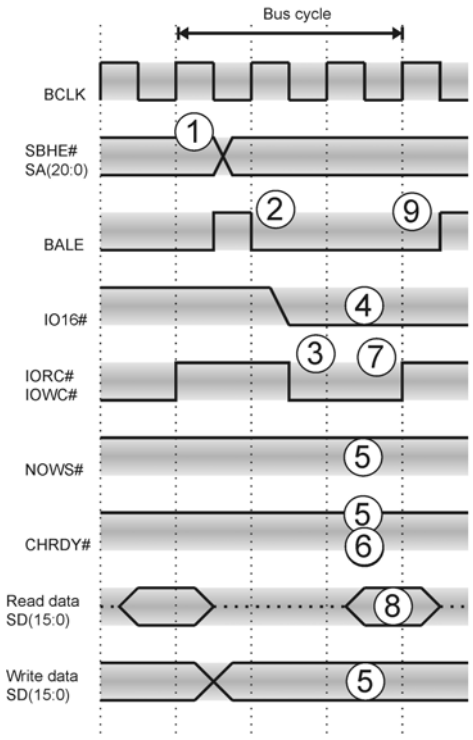


Figure 2-16: ISA bus access to a standard 16-bit I/O device.

2.4.3 ISA bus interrupt signalling

An interrupt on the ISA bus is when a device needs service from the microprocessor. To generate an interrupt request the device generates a positive flank on its interrupt line. To generate an interrupt the device could for example let its interrupt line be pulled high in normal state. When the device needs service from the microprocessor, it drives its interrupt line low. Shortly after that the device releases it and the signal will be high again because of its pull-up. The microprocessor will notice the positive flank and service the device corresponding to that interrupt.
There are 11 interrupt lines on the ISA bus and sometimes some interrupt lines are shared among several devices [17].

2.5 LVDS

2.5.1 Introduction

LVDS is a differential data transmission standard that started to become popular in the industry in the end of the 1990s. It supports high-speed communication from 100 Mbps to more than 1 Gbps while it has very low power consumption. It also has the benefits of low noise generation and high noise rejection [18].
The reason why LVDS was invented is that better performance was needed. Other differential standards consumed much more power and are not capable of the high data rates of LVDS. For example, RS-422 has a power consumption of about 90 mW while LVDS only consumes about 1.2 mW. Other technologies such as RS485, ECL, and PECL also consume significantly more power. The voltage swing of LVDS is also very small, which could be a draw back because the logic states are harder to determine.
Considering the transfer speed, LVDS is capable of data rates of about 500 Mbps while for example the RS-485 maximum data rate is 10 Mbps or less. The reason why the standards ECL and PECL did not get as big acceptance was their incompatibility with standard logic levels and high chip power dissipation [18].

2.5.2 LVDS link configurations

The configuration shown in Figure 2-17 is called point-to-point [19]. This is the most common and basic LVDS link configuration. It transmits the differential data signal in one direction and is the best suitable configuration for high data rates.

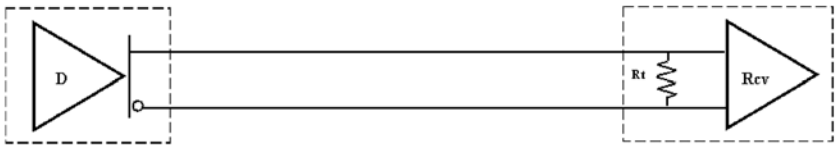


Figure 2-17: LVDS point-to-point configuration.

A 100 Ω termination (R_t) is needed at the receiver to avoid signal reflection. This value should correspond to the internal resistance of the cable bus. In this and in the following configurations the two wires are assumed to have an impedance of 50 Ω.
The configuration in Figure 2-18 is called multidrop and only one termination (R_t) is needed at the end of the bus. It is important that the receivers without termination are connected close to the bus to avoid reflections. The number of receivers can be up to about 20, depending on the transmitter driver capacity and the line quality.

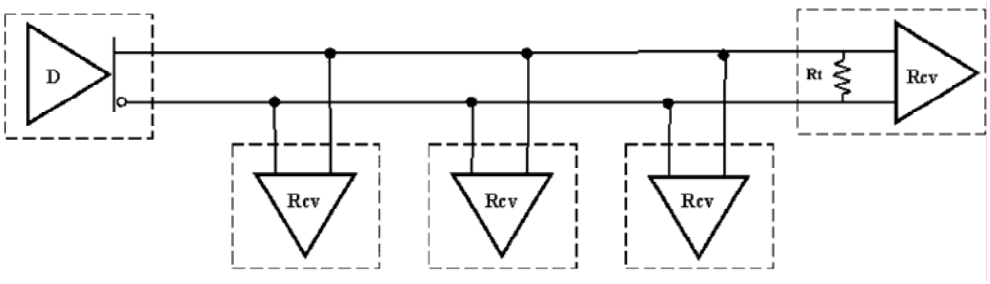


Figure 2-18: LVDS multidrop configuration.

The multipoint configuration shown in Figure 2-19 consists of several transceivers. Each transceiver is able to communicate to either of the other transceivers. The fact that it is two terminating resistors in this construction the transmitter has to be able to drive more current. There are LVDS standards that take this into account. See the sections about BLVDS and M-LVDS below.

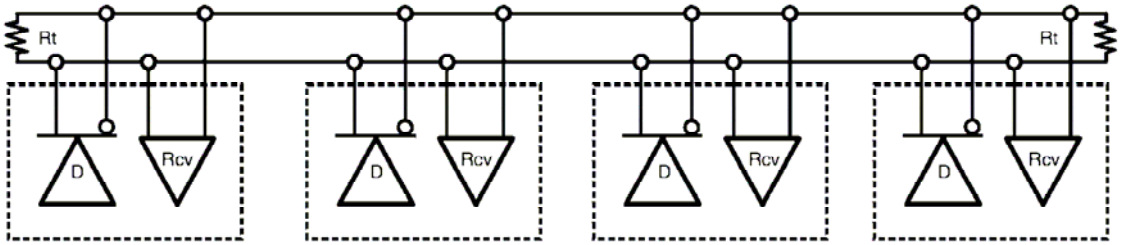


Figure 2-19: LVDS multipoint configuration.

2.5.3 LVDS standards

2.5.3.1 LVDS

National Semiconductor introduced a standard for low voltage differential signalling in 1994. The general LVDS standard is now defined by the industry standard ANSI/TIA/EIA-644. The standard defines the electrical specifications including voltage levels, transmitter driver and the receiver input characteristics [20]. The LVDS voltage levels are shown in Figure 2-20. The dotted line is the voltage level in one of the two wires in the data bus. The continuous line is the voltage level in the other wire. The voltage swing is typically 350 mV and the offset above ground is typically 1.25 V. The Spartan 3 FPGA uses these voltage levels in the BLVDS_25 I/O standard as well.

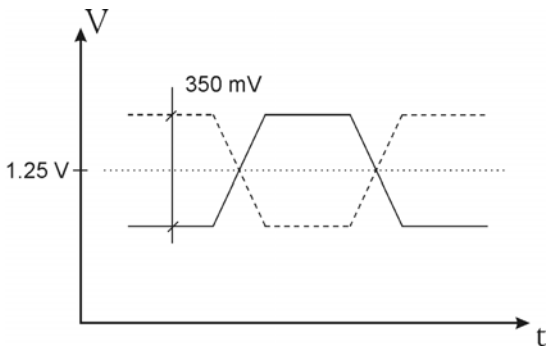


Figure 2-20: The LVDS voltage levels typically used.

2.5.3.2 BLVDS

The BLVDS (Bus LVDS) standard was invented by National Semiconductor to solve driving problems in bus configurations using many receivers and several terminating resistors. In the multipoint configuration two terminating resistors are used. If many transceivers are connected to the bus, the bus impedance will be lower. If the bus impedance is lower, a lower termination resistor is needed. This will demand a higher current from the transmitting driver. BLVDS is defined to support up to 10 – 17 mA driving current to solve this problem. Even more improvements are defined in this standard and the resulting data transfer rate is in the range 200 to 400 Mbps [18].

2.5.4 Data rates supported by LVDS in different applications

The achievable bus data rate depends on many things. First of all correct termination has to be used. The driver also has to be able to support enough current. If this is not the case, the terminating resistance value might have to be increased as a compromise. The number of receivers and the cable length is also vital to the resulting data rate.

For long transfers and if a CAT5 type cable is used in point-to-point configuration 100 Mbps is achievable in a 20 m cable. If a 50 m and a 100 m cable is used a data rate of 50Mbps and 10 Mbps respectively is achievable [21].

For short transfers on one singe PCB board the data rate may be up to 622 Mbps using a Virtex-E FPGA with speed grade –7 [13].

For transfers over a 1.5 m PCB-trace and using the same FPGA a data rate of 311 Mbps is reliable [22].

Chapter 3 – The distributed ISA bus network design

The distributed ISA bus network designed in this master thesis is described in this chapter. All the VHDL codes and schematics were programmed by me except for three components of the RS232 controller[I]. No VHDL code presented in this project has been automatically generated. The described design has been implemented using the Xilinx Spartan 3 starter kit board and the H4070 heat test board designed by Hectronic AB. First, in the section 3.1 below, the choices of the used technologies are discussed.

3.1 Preliminary work – designing the distributed network

3.1.1 Main system interface

The interface from the host computer to the test system has to be fast enough to upload programs to the test object. Because the test system has the limit of only five wires the speed of the interface does not have to be significantly faster than the maximum speed of the test system bus. If the test system bus has two differential channels, the bit rate would be about 100 to 800 Mbps.
The ISA bus was chosen because it supports enough speed for the applications (about 40 Mbps) and because it is simple to implement in programmable logic. Many I/O devices support this bus and so does the Hectronic PC/104 boards.

3.1.2 Backplane architecture

3.1.2.1 Parallel versus serial signalling

Because the thin bus structure consists of only five wires, the parallel communication would not meet the data-rate needed. Serial signalling will therefore be implemented.

3.1.2.2 Single-ended versus differential signalling

As mentioned above, the specified bus width is only five wires. This means that a single-ended channel does not support the data-rate needed. A differential channel is harder to implement but the high-speed support and the good noise rejection ratio makes this the best choice.
This means that a maximum of two differential channels are available; which corresponds to a transfer rate of about 100-800 Mbaud[II] depending on the timing architecture.

3.1.2.3 Data distribution topologies

Because the test system has to be able to access several network nodes and the data-bus consists of only five wires, the multipoint topology is the most suitable one. The point-to-point topology is not applicable because a maximum of two differential channels are available and the system has to support several transceivers.

[I] The RS232 controller is designed in the UART-RE232.sch schematic on page 118. The only components not programmed by me is the bbfifo_16x8, kcuart_rx and kcuart_tx.
[II] Baud is the rate of logical changes per second in a transmission channel. Start and stop bits are included in the baud rate even though they do not carry any data. The data rate measured in bps includes only actual data bits.

3.1.2.4 Timing architecture

The asynchronous timing architecture does not allow sufficient data rate because a wide data bus can not be used.
The synchronous architecture with one clock generator, in section 2.2.2.1, does not support high data rates at long distance because the big clock to data skew. The maximum allowed path difference was calculated in section 2.2.2.5 and limits the distance between two cards to 90 mm at 100 MHz. Because two cards in the network could be a few meters apart, the synchronous architecture is not suitable.
Source synchronous architecture is a fast design because data and clock signals are sent in parallel (low skew). The design in section 2.2.2.2 uses several clock lines. The wire limit only allows 2 differential channels and therefore this architecture has to be modified. The solution is to have both a data and a clock bus. The transmitters send the data and the clock signals on two differential pairs in parallel as shown in Figure 3-1.

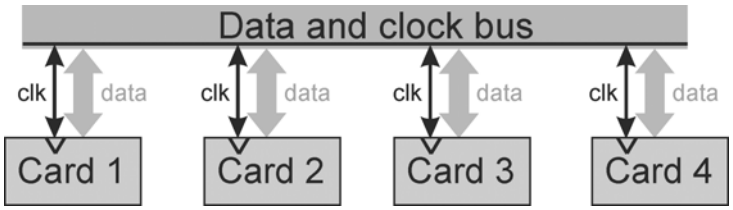


Figure 3-1: Modified source synchronous architecture. Both data and clock are sent in parallel on the bus.

Because two differential pairs are needed in this design, power can be supplied by the use of Power over Ethernet technique or similar.

3.1.2.5 Embedded clock - timing embedded in the data

To have the timing embedded in the data seems to be preferential. Only one differential pair has to be used and the power supply could be transmitted in two of the remaining wires. The multipoint topology could still be used and all the data transmissions could take place in only two wires – one differential pair.
The big problem with this design is to overcome the difficulty of extracting the data clock and still be able to receive the data at a high bit-rate. It is also a matter of cost. Devices handling high-speed data with an embedded clock are rather expensive. Maybe it will not be possible to implement this design in a low-cost FPGA. Much thought has been put into this problem but no good solution has been found. But this design can not be ruled out completely and should be investigated further.

3.1.3 The chosen logic in the test system

The programmable logic in the test system has to support a fast differential standard. It also has to be able to support the ISA bus standard and other bus architectures. A FPGA/CPLD is suitable for this task. The advantage with a CPLD is that it is cheaper than a FPGA and it maintains its configuration without the need of constant power. The disadvantage is that it contains less complex logic.
The FPGA on the other hand has support for several I/O-standards, phase looked loops (PLLs) and clock multipliers. The major disadvantage is that it has to have an external Flash memory to maintain the programming code.
The Xilinx Spartan 3 FPGA was chosen because it supports the BLVDS standard, is a low-cost FPGA, and has digital clock managers.

3.1.4 ISA backplane design

Because the host computer will access the test system via an ISA bus, a design of an ISA backplane is a good approach. This means that a transparent ISA bus is distributed by the test network. All of the nodes in the network will act as an ISA bus and these can be connected to several peripheral devices at

some distance apart. The devices should not notice any difference if they are connected to the real data bus or one node in the distributed system. Two different ISA backplane design techniques were thought of. These are explained below:

3.1.4.1 Continuous ISA bus distribution

In a transparent distributed backplane, the idea is that the distributed bus should act just as if it was the system bus in the computer. This can be done by having the distributed network to continuously sample the original computer bus. All of the distributed nodes are continuously updated. It works the same way the other way around. The node at the device samples the signal-bits from the device and sends update information to the master node connected to the host computer.
This solution is simple and would work fine for most devices. But it has one major problem. There will always be a delay in the system. In the ISA bus architecture there is a signal used by the device to postpone the ending of the bus cycle. When the device has decoded the address and need some extra time to finish the access, it deasserts this signal (CHRDY). For a normal 16-bit access, the device has 125 ns to deassert this signal [17]. But for a distributed system the address will reach the device delayed. Then the response from the device will be transmitted back to the master node resulting in one more delay. If the device signals for a wait state after about 80 ns this would work fine in a normal ISA system. In the distributed system it might not work because of these delays.
The same problem is with the IO16# signal. This signal tells the system if the device is 16-bit or 8-bit compatible.
Giving all the devices one extra wait state can solve the CHRDY signal delay problem above. To solve the IO16# signal delay problem one has to allow only 8-bit or 16-bit devices. If only 8-bit devices are allowed, there will be slower transfers. If only 16-bit devices are allowed, this will significantly limit the use of the bus because 8-bit I/O-devices are very common.
The delay problem makes the use of this continuously distributed ISA bus method too limited. The solution in next section is therefore chosen for this project.

3.1.4.2 ISA slave in the master node and ISA bus masters in the slave nodes

To solve the problem with the delay and to support both 8-bit and 16-bit devices this design was chosen. The test network acts like a device (ISA slave) on the computer bus and the nodes in the distributed network acts like a bus manager (ISA master). The distributed ISA bus architecture is shown in Figure 3-2.

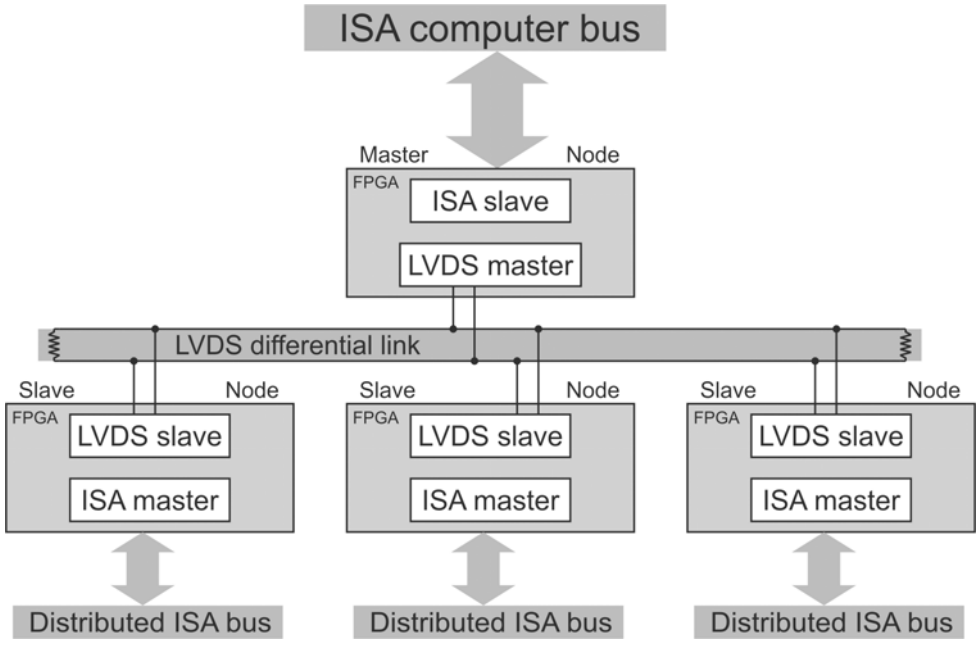


Figure 3-2: Distributed ISA bus network connected using a LVDS link.

The command is put on the ISA computer by the host computer test program. If it is a write command, the data and address bits are sampled by the ISA slave and sent to the LVDS link by the LVDS master in the master node. All the slaves receive the LVDS data block and the device that has the matching address will send the address, command and data bits to its ISA master.
The communication flow described above is slightly more complicated in the final system and is explained in detail in the sections below. For information about the communication flow, please consider the chapter about the ISA bus in section 2.4 and the LVDS master/slave protocol in section 3.3.1.

3.1.5 Conclusion

The decisions made in this chapter have resulted in a design that fulfils all the wanted properties in a new test system. In short, the design will distribute an ISA bus via LVDS and the design will be implemented by the use of Spartan 3 FPGAs. The only thing that might improve the design is the use of an embedded clock. But this option needs to be investigated further.
It should be noted that the chosen design is very flexible. If a better solution to transmit the LVDS data is found, only small changes are needed in this final design. A complete design description will be found in the next section.

3.2 Introduction to the distributed ISA bus network design

The distributed ISA bus network design distributes data from the host computers ISA bus to the slave nodes. The master node samples the computers ISA bus and distributes the data, command and address to the addressed slave node, via the LVDS link. The master node and the slave nodes consist of a central FPGA that may include several functions. In this project, the master node has an internal register manager and an ISA slave, a RS232 and a LVDS master controller. The slave node has an internal register manager and an ISA master, a RS232, an I^2C slave and a LVDS slave controller.
Both the master node and the slave node are very flexible designs. All of the communication controllers are programmed as modules (black boxes) and are controlled by a central state machine. The state machine receives commands from either the ISA slave module (master node) or LVDS slave (slave node). The state machine then coordinates the data flow between the modules. The RS232 and the I^2C controllers are completely managed by the internal registers in the FPGA. The internal register manager module handles all internal registers.
The master node, the slave node and all the modules are explained in their respective sections found in chapters further down.

3.2.1 Communication flow

The design shown in Figure 3-3 consists of one master node and two slave nodes. Several slave nodes can be connected to the distributed system and the number of slave nodes is only limited by the LVDS link capacity. The host computer can read and write data to all shown controllers in Figure 3-3.

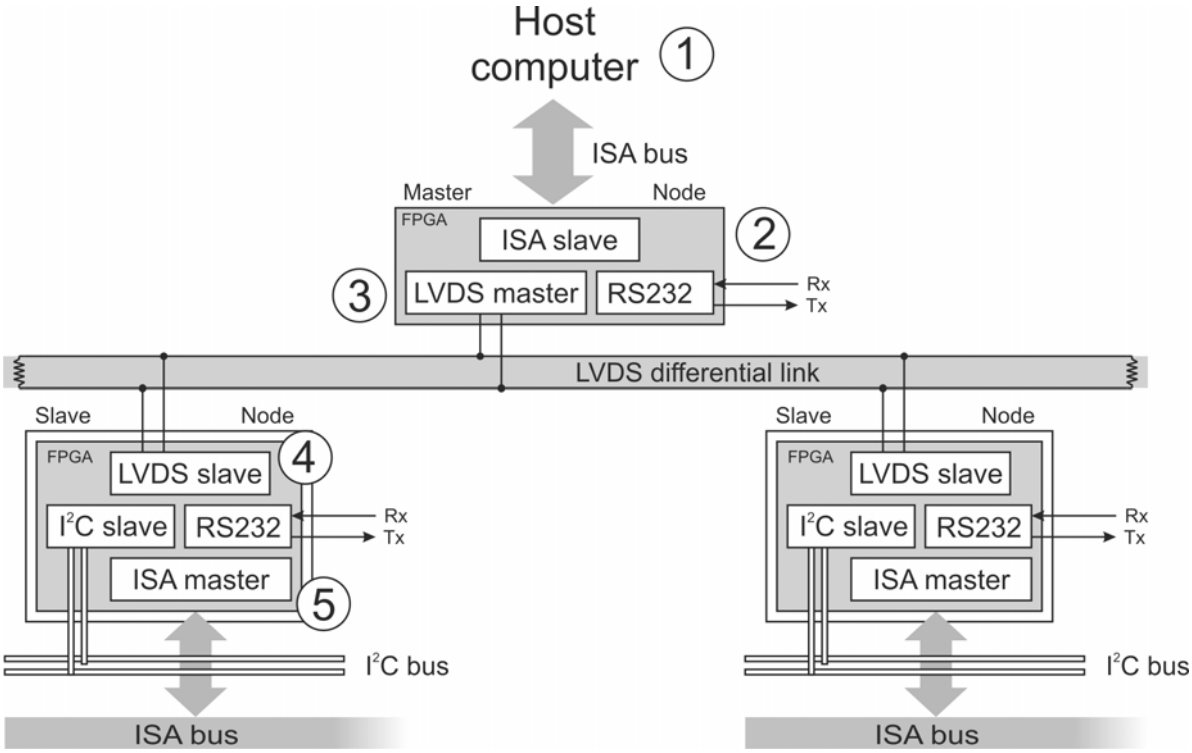


Figure 3-3: The distributed ISA bus network design.

The communication flow is explained in the numbered list below and the numbers in the list correspond to the numbers in Figure 3-3.

3.2.1.1 Writing data to a slave node in the distributed system

1. The main program on the host computer writes data to an address in the predefined address interval of the distributed system.
2. The ISA slave module in the master node FPGA checks the address on the ISA bus. If the address is in the right address range the ISA slave module halts the computers ISA bus by deasserting the CHRDY command line. If the address corresponds to an internal register or to the RS232 module, it is sent to the internal register manager. If the address is in the range of the slave nodes, it is sent to the LVDS master module. When the internal register manager or the LVDS master module signals that the transmission has ended, the ISA slave module asserts the CHRDY signal and the ISA bus cycle on the host computer is allowed to end.
3. The LVDS master module transmits the data, the address and the error checking code on the LVDS link. It then waits for a reply from the addressed slave node. If no reply is received in a specified time interval or if the received data block is corrupt, the data is transmitted once more. If an accurate acknowledge message is received or if a transmission error has occurred twice the LVDS master signals that the transmission is completed.
4. The LVDS slave in the slave node checks if the data block received corresponds to its address range. If the address corresponds to an internal register or to the RS232 or the I^2C controller, it is sent to the internal register manager. If it corresponds to the ISA bus, it is sent to the ISA master module. When the addressed module signals that the transmission is completed, a response message is sent to the LVDS master in the master node.
5. The ISA master module sends the data and address to its distributed ISA bus and signals to the main logic in the slave node when the ISA bus cycle has ended.

3.2.1.2 Reading data from a slave node in the distributed system

The procedure to read data from a slave node works in the same way as writing but the acknowledge message sent back from the slave node now contains the read data as well.

3.2.2 The design described using the OSI model

When designing the modules in this project the OSI model has been considered as a reference. All of the design is contained in the four lower levels of the model. The figures below show the data flow through the layers when different accesses are being made. The modules used in the access are placed in their specific layers and the arrows show the direction of the data flow. Figure 3-4 shows the data flow when a distributed ISA bus is accessed. Figure 3-5 shows an access to the internal register manager and the RS232 controller in the master node. Figure 3-6 shows the access to the register manager and the I^2C and RS232 controllers in the slave node. Information about the OSI model can be found in section 2.1.

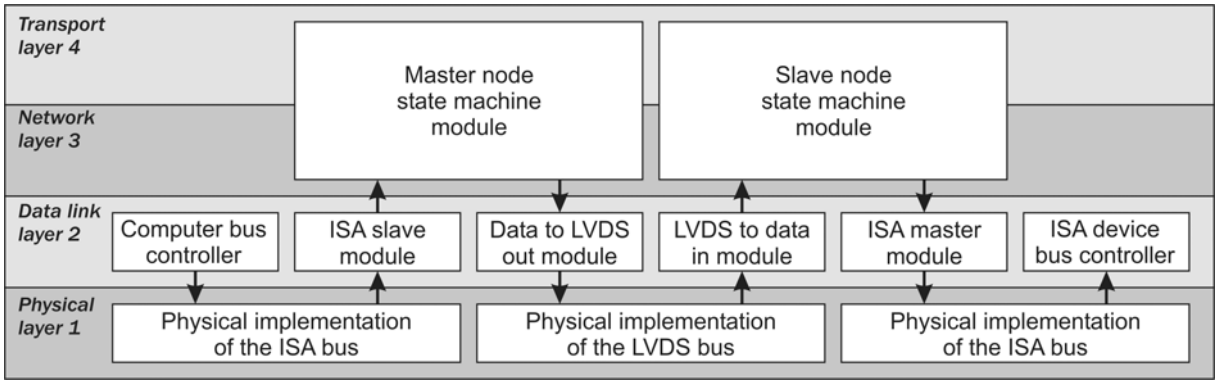


Figure 3-4: The data flow during a distributed ISA bus access, presented using the OSI model.

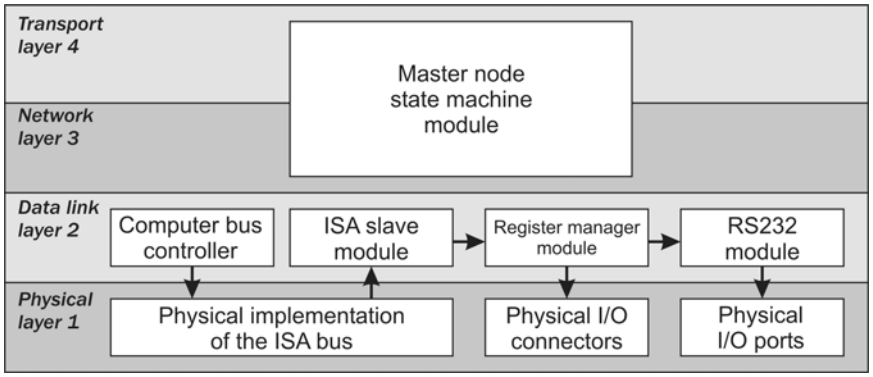


Figure 3-5: The data flow during an access to an internal register, I^2C or a RS232 module in the master node. The flow is presented using the OSI model.

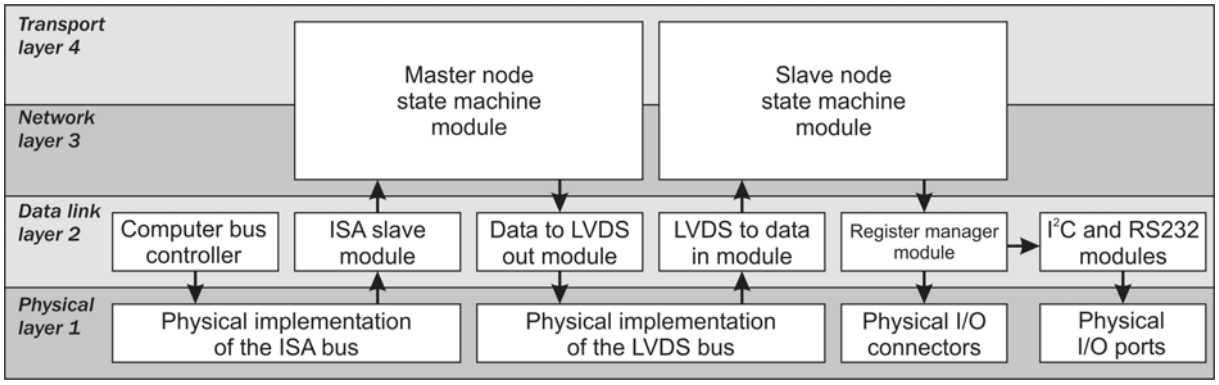


Figure 3-6: The data flow during an access to an internal register, I^2C or a RS232 module in the slave node. The flow is presented using the OSI model.

3.3 Internal system communication

3.3.1 Serial data block between nodes

The data on the LVDS link is sent as a serial data block. The data block is 40 bits long plus one start bit. The data is sent with the most significant bit first. That means that bit 39 is sent first, directly after the start bit. The data block is shown in Figure 3-7 and is the same for all communications on the LVDS link. This is for simplicity. The data block could be shorter when sent from the slave node, but because this test system is only in the development stage, it can be optimised later.

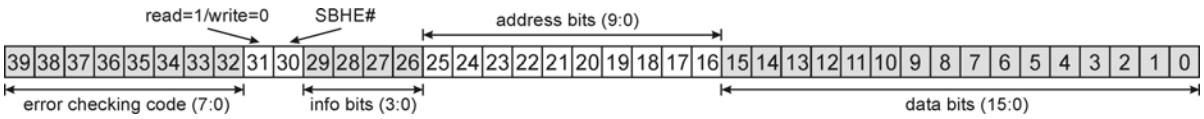


Figure 3-7: The 40-bit LVDS data block (39:0).

The functions of the bits in the data block are explained in the list below:

- **Error checking code (7:0):** This is the error checking code for the data block. It is generated from the remaining 32 bits in the data block. The code is generated by the use of XOR gates as explained in section 3.4.2.1 on page 42.
- **Read=1/write=0:** This bit specifies if it is a read (logic 1) or a write (logic 0) access.
- **SBHE#:** This signal specifies if it is two bytes (logic 0) or one byte (logic 1) in the data bits (15:0).
- **Info bits (3:0):** These bits are used to send certain messages between the nodes. Three of these bits are not used but are spared for future designs. Info bit 2 is used to tell the slave that the message is an interrupt poll request. These bits could in a future design, be used to reset the slave nodes or to request a resend of the data block, for example.
- **Address bits (9:0):** These bits contain the address of the I/O device or register accessed.
- **Data bits (15:0):** These bits contain the data to be written or the data that has been read. If SBHE# is asserted all bits are valid and if the bit is deasserted only the data bits (8:0) are valid.

3.3.2 Asynchronous communication between the modules and the state machine

The state machine and the modules in the nodes are communicating via asynchronous signals. The signalling can be done in many ways, but in this design only two control signals are used.

Figure 3-8: Example of the internal asynchronous communication between components in the FPGA.

The connections are shown in Figure 3-8 and the communication is explained in the list below. Note that in this example the state machine masters the communication:

1. First, the state machine latches the data on the output.
2. Then the state machine asserts the command signal.
3. The module receives the command signal and performs the requested task. For example, the command could signal a write access, and the data would then be written to a register in the module.
4. When the task is performed, the module latches the data on its output (if needed).
5. Then the module asserts the transmission_ok signal and waits for the command signal to be deasserted.
6. The state machine notice the transmission_ok signal and deasserts the command signal.
7. The module deasserts the transmission_ok signal and the communication cycle ends in the state it started from.

3.4 The modules

3.4.1 The internal register managers

There are one register manager in the slave node and one in the master node. These can be addressed from the host computer and may be used to configure the master/slave node or to communicate with implemented controller modules.
The internal register manager module is very easy to expand into more registers. If more modules such as JTAG and CAN controllers are to be implemented, registers supplied by the register manager can easily control them. All the communication to and from these controllers will be made through the register manager.
When needed, a register is accompanied by an extra signal. This signal tells when the register has been written to or read from. For example, when the receiver buffer register in the RS232 module is read from, the internal register manager has to signal to the module. The RS232 module receives this signal and the read character is removed from the FIFO. In the same way, a signal is needed to inform the FIFO that it is written to, and therefore has to load the FIFO with this new value. Note that the module receives an acknowledge command signals asynchronously. But the modules controlled by the register manager has to use the same clock if the read and write signals has to accompany the registers.
The main signals discussed above are shown in the Figure 3-9. The figure only shows some of the input and output signals used by the register manager.

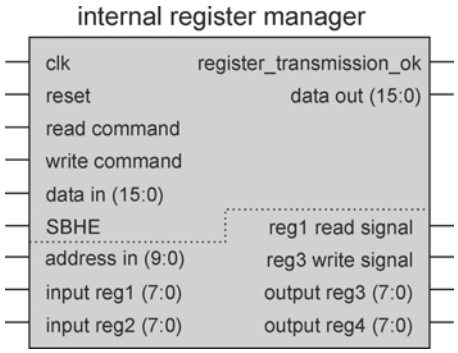


Figure 3-9: The main input and output signals of the internal register manager module.

The communication flow in the internal register manager is described below:

1. First, the state machine controlling the register manager puts stable signals on the SBHE#, data and address inputs. The register_transmission_ok signal has to be deasserted before the state machine may assert any of the command signals.
2. Then the read or write command is asserted. If the SBHE signal is asserted it is a 16-bit access, else only one byte is read or written.
3. If the access is a read command, the value of the addressed input registers is put on the data out bus. If a read signal is needed to be sent along with the input register, this one is asserted now as well. If the access is a write command, the value of the data input bus is written to the addressed output register. If a write signal is to be sent along with the output register, it is asserted now just as for the read access. If it is a 16-bit read access and the addressed register is even, the addressed register and the register of the address +1 will be set as two bytes on the 16-bit data bus. In the same way if it is a 16-bit write access and the addressed register is even, the addressed register and the register of the address +1 will be written to. Now when the transfer is complete the internal register manager will signal to the state machine by asserting the register_transmission_ok signal.

3.4.1.1 The registers in the distributed system

The register manager in this design is differently configured for the master node and the slave nodes. The address space used is h3E8 – h3EF. These registers are addressing either the master node or the slave node according to the “slave bit” set in the master bus controller register. The register functions are explained below in Table 3-1.

Slave bit	Register address	Read/ Write	Register name
0	h3E8	R/W	Master: Receiver buffer reg. / Transmitter holding register
X	h3E9	R/W	Master bus controller register
0	h3EA	R/W	Master: Divisor, low byte
0	h3EB	R/W	Master: Divisor, high byte
0	h3EC	R	Master: FIFO status register
0	h3ED	R/W	Master: Scratch register
0	h3EE	R/W	Master node configuration data, low byte
0	h3EF	R/W	Master node configuration data, high byte
1	h3E8	R/W	Slave: Receiver buffer reg. / Transmitter holding register
1	h3EA	R/W	Slave: Divisor, low byte
1	h3EB	R/W	Slave: Divisor, high byte
1	h3EC	R	Slave: FIFO status register
1	h3ED	R/W	Slave: Scratch register
1	h3EE	R/W	Slave: I2C bus register select
1	h3EF	R/W	Slave: I2C bus data register

Table 3-1: Register addresses in the distributed system.

Receiver buffer register

This register contains the received character in the RS232 module.

Transmitter holding register

This register contains the character to be sent from the RS232 module.

Master bus controller register

This register controls if the registers in the slave node or the master node should be accessed. If the least significant bit called the "Slave bit" is set, the registers h3E8, h3EA-h3EF is connected to the slave node, else these will be connected to the master node. The bit-functions are explained in Table 3-2.

Bit	Function
7	'0'
6	'0'
5	'0'
4	'0'
3	'0'
2	'0'
1	'0'
0	"Slave bit"

Table 3-2: Bit functions of the master bus controller register.

Divisor, low byte and high byte

These two registers forms the 16-bit divisor controlling the baud-rate of the RS232 module. The register manager clock domain is connected to the baud-rate generator. The 16-bit divisor is calculated as:

$$Divisor\ number = \frac{baud\ rate\ generator\ \ frequency}{(the\ wanted\ baud\ rate)\cdot 16}$$

FIFO status register

From this register the status of the RS232 FIFOs can be read. The bit-functions are explained in Table 3-3.

Bit	Function
7	'0'
6	Rx FIFO is full
5	Rx FIFO is half full
4	Rx FIFO data is present
3	'0'
2	Tx FIFO is full
1	Tx FIFO is half full
0	Tx FIFO data is present

Table 3-3: Bit functions of the FIFO status register.

Scratch register

This register can be used as a scratch register. It has no other purpose.

Master node configuration data, low byte and high byte

These registers contains 16 bits to configure the master node in future implementations.

I^2C bus register select and the I^2C bus data register

This register acts as an address and decides which of the I^2C module registers to be accessed by the I^2C bus data register. The available I^2C registers are listed in Table 3-4.

I^2C bus register select value	R/W	Activated I^2C register
h00	W	Register 0
h01	W	Register 1
h02	W	Register 2
h03	W	Register 3
h04	W	Register 4
h05	R	Register 5
h06	R	Register 6
h07	R	Register 7
h08	R	Register 8
h09	R	Register 9

Table 3-4: Registers in the I2C module.

3.4.2 LVDS in and out modules

The LVDS in and LVDS out modules are controlled by the main state machine in the slave or the master node. The state machine decides when to receive and when to transmit data. This means that the LVDS modules can act both as a LVDS master or a LVDS slave depending on how the state machine is configured. The LVDS out module transmit the data 90 degrees ahead of the clock signal so that the receiver can sample the data bits using the clock signal. The simulation of the LVDS out module is shown in the Figure 3-10.

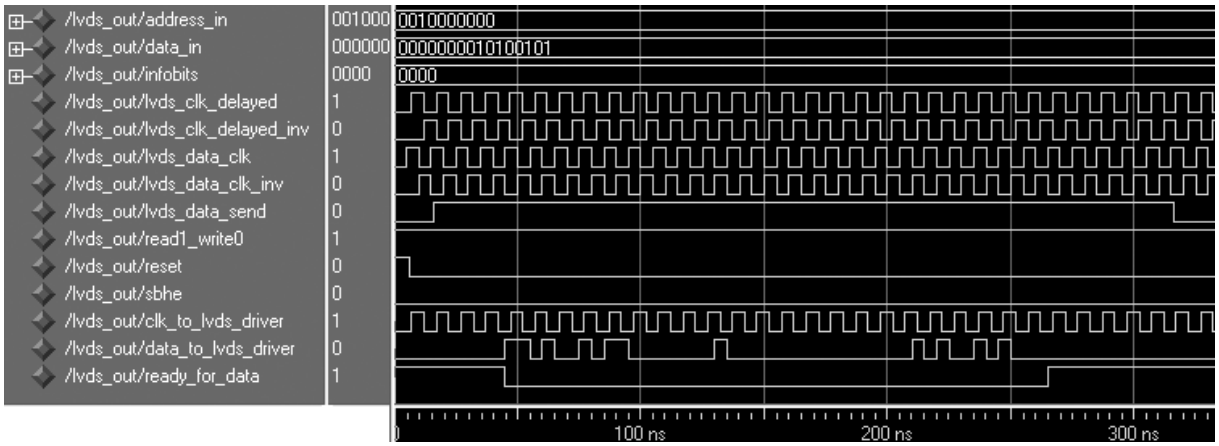


Figure 3-10: Simulation of the LVDS data and clock signals generated from the LVDS out module.

The data flow in the two modules are shown in Figure 3-11 and the communication between the state machine, the LVDS link and the LVDS modules are explained below:

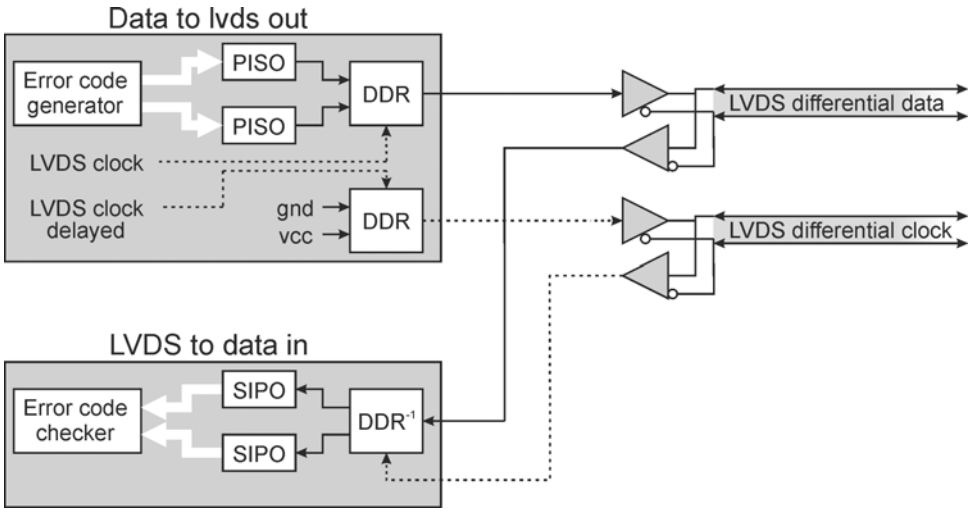


Figure 3-11: The LVDS in and LVDS out modules.

The LVDS out module transmits data according to the list below:

1. First, the state machine checks if the LVDS out module is ready to send data by checking the lvds_out_ready signal. It also enables the differential output buffer so that the data generated will reach the differential output pins on the FPGA.
2. The data, address and command signals are set and the error code generator component in Figure 3-11 will compose the data block described in section 3.3.1.

3. The data block is split up and loaded into two PISO[I] shift registers.
4. The DDR mux will send the data bits coming from the shift registers at a data rate two times the LVDS clock rate. The shift registers take turn in presenting the next data bit to the DDR MUX. In this way the shift registers and the DDR MUX are driven by the same clock and generates a bit stream at double data rate.
5. Another DDR MUX has gnd and vcc as inputs instead of data from the shift registers. Because the clock that drives the DDR clock MUX is delayed, the generated data bit stream and the clock will be phase shifted. This is because the receiver should be able to sample the data stream with the delayed clock signal. The data on the two LVDS channels is sent to another node in the network and is received by an LVDS in module in that FPGA.

The LVDS in module receives data according to the list below:

1. First, the state machine resets the LVDS in module and then it waits for a message to be received.
2. The LVDS data is sent with one start bit (logic 1) and one stop bit (logic 0). The DDR receiver splits the incoming data bits using the delayed clock signal as a trigger. The data is split up into two bit streams received by two SIPO[II] shift registers. When the shift register that received the first bit is full, it will deassert the enable signal on itself and on the other shift register. It will also signal to the error code checker that the data registers are full.
3. The error code checker will take the received data block, regenerate the error checking code and compare it to the received data.
4. If the data is received correctly, the transmission_ok signal is asserted and read by the state machine. If the data is corrupt, it will assert the transmission_bad signal instead.

3.4.2.1 The error code generator

The error code included in the data block is generated in two stages by XOR gates in series. From 32 bits, 8 control bits are generated and the procedure is shown in Figure 3-12. The method is fast and easy to implement.

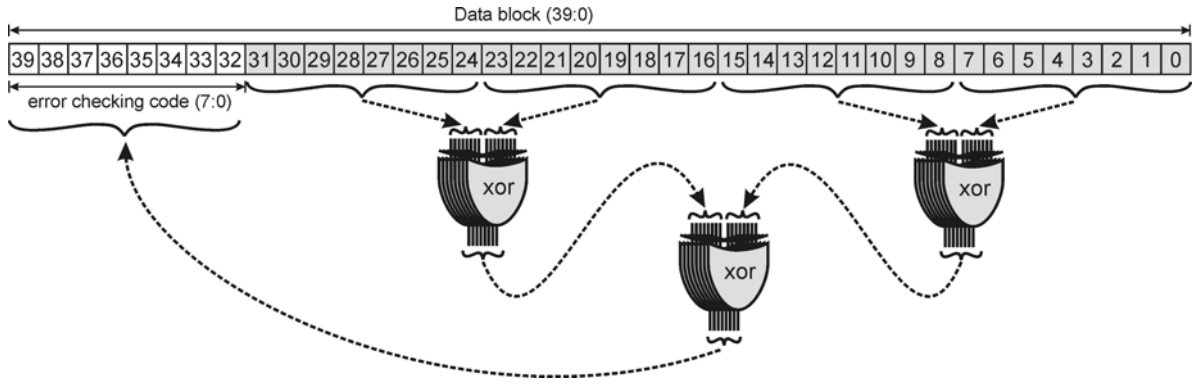


Figure 3-12: Error code generation using XOR gates.

3.4.3 Cascaded DCMs

Two cascaded digital clock managers are used to generate the clock signals in the FPGA. The quality of the clock signals decrease by arranging them in series. In this design, this is done anyway to achieve a high data rate. One DCM is capable to multiply the input clock signal. It is also able to produce four clock signals phase shifted by 90 degrees from each other. But one single DCM can not do these tasks at the same time. Therefore, two DCMs are needed. One to multiply the input clock,

[I] Parallel In Serial Out.
[II] Serial In Parallel Out.

and one to generate the shifted clock signals. The arrangement shown in Figure 3-13 produce all the clock frequencies needed to optimise the design. The input clock frequency used in this design has a frequency of 50 MHz. The cascaded DCM arrangement then produce the four phase shifted clock signals needed for the LVDS out module, one 50 MHz clock for the internal logic and one 25 MHz signal to drive the internal register manager and its controller modules.

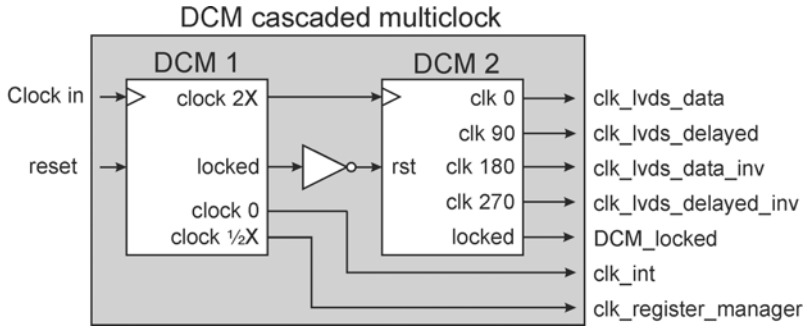


Figure 3-13: The arrangement of the two cascaded DCMs generating the clock signals driving the FPGAs internal logic.

3.4.4 RS232 controller

The RS232 controller module consists of one RS232 transmitter and one RS232 receiver. Both the transmitter and the receiver are accompanied by a 16 byte deep FIFO. The transfer speed can be chosen by the use of two divisor registers and the state of the FIFOs can be read from one register. The transmitter holding register and the receiver buffer register has the same address.

3.4.5 I^{2}C slave controller

The I^{2}C slave controller is programmed according to the system management bus specification [23]. This module is equipped with five 8-bit input registers and five 8-bit output registers. All of these are controlled via the internal register manager in the slave node.

3.4.6 Timers

There are four timers to coordinate the communication in the distributed system. All of these timers have one clock input, one reset and one time-out output signal as shown in Figure 3-14. A simulation of the LVDS slave transmission timer is shown in Figure 3-15.

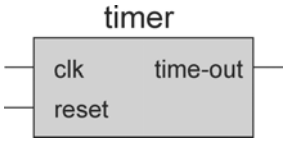


Figure 3-14: The entity configuration of the timer components.

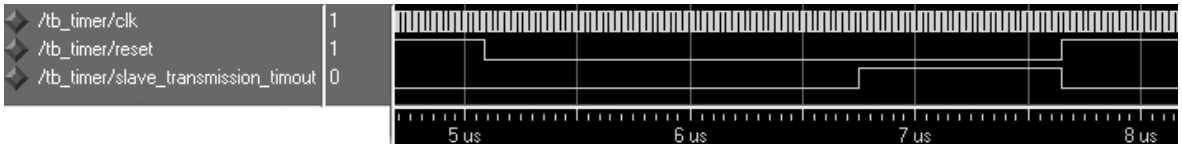


Figure 3-15: Simulation of the LVDS slave transmission timer.

To start the timer the reset signal is deasserted. When the timer has counted the clock pulses of the clock signal and achieved the specified value, a time-out signal is asserted. The time-out signal will be deasserted as soon as the reset signal is asserted again. The function of the timers are explained below:

- **LVDS master transmission timer:** This timer is used by the master node state machine to keep track of how long time it takes for the slave node to reply. If this timer times out, the message will be retransmitted or the address will be treated as if it is not accessible.
- **LVDS slave transmission timer:** The slave node state machine keeps track of how long time the access from the master node has taken using this timer. If this timer times out, the slave node will not be allowed to send a reply message to the master node and the ongoing access from the master node will be aborted.
- **IRQ timer:** This timer times out when there has been no access on the LVDS link for a specified time. This time-out gives the master node an opportunity to poll the slave nodes for ISA bus interrupt requests.
- **ISA bus 15 us timer:** The ISA slave uses this timer to assure that the access is not taking more than 15 microseconds. Because the ISA slave deasserts the CHRDY signal on the ISA bus till the distributed system transfer is completed, the RAM refresh cycles on some systems can be disturbed. To prevent this the ISA slave will abort the access and release the ISA bus when this timer times out.

3.4.7 ISA slave

The ISA slave module acts as a 16-bit device on the ISA bus. The data, address, SBHE# and command signals are sampled. If the address is in the specified range the CHRDY signal will be deasserted and the bus will be prolonged till the CHRDY signal is asserted again. The slave node will signal to either the internal registers or to the master node state machine. When a respond signal is attained from the addressed module or if the ISA bus 15 us timer will time-out, the ISA slave will release the bus by asserting the CHRDY signal again.

3.4.7.1 ISA slave interrupt handler

The master node state machine controls the ISA slave interrupt handler, IRQ out. The interrupt handler signals to the state machine which of the 11 interrupts to be polled next. When the state machine has polled an interrupt, it informs the interrupt handler if this interrupt has been requested or not.
If the interrupt has been requested, the corresponding interrupt line on the ISA bus is held low for three clock cycles (60 ns when using a 50 MHz clock) and is then released. This procedure will signal to the microprocessor on the ISA bus that this interrupt has been requested.
When the interrupt polling cycle has ended, the next interrupt can be polled the same way as described above.

3.4.8 ISA master

The ISA master module acts as the microprocessor mastering the ISA bus. The I/O devices connected to the ISA master module will receive the bus signals, just as if they were sent by the microprocessor. Only I/O device accesses and interrupt handling are supported. The I/O devices can be read from or written to using both 8-bit and 16-bit accesses. Memory devices and DMA are not supported.
The bus cycles can be prolonged and shortened by the use of the CHRDY and the NOWS# signals respectively. More information about the ISA bus can be found in section 2.4.

3.4.8.1 ISA master interrupt handler

The ISA master interrupt handler, IRQ in, samples the interrupt lines from the distributed ISA bus. If a positive edge is noticed, the interrupt request will be stored by the interrupt handler. When the slave

node is polled for this interrupt by the master node, the slave node state machine will request and receive the status of the polled interrupt from the interrupt handler.

3.5 The master node

All the logic in the master node is programmed into one Spartan 3 FPGA. A central state machine, master node state machine, handles the communication between the modules as shown in Figure 3-16. The instructions from the host computer are received by the ISA slave module and are sent to the slave nodes through the LVDS link. The state machine acts as a master on the link using the LVDS in and LVDS out modules. The dotted areas in the figure marks out the four different clock domains in the master node. This makes it possible to run the high-speed parts such as the LVDS modules at a higher frequency while some slower logic may run at a lower frequency.

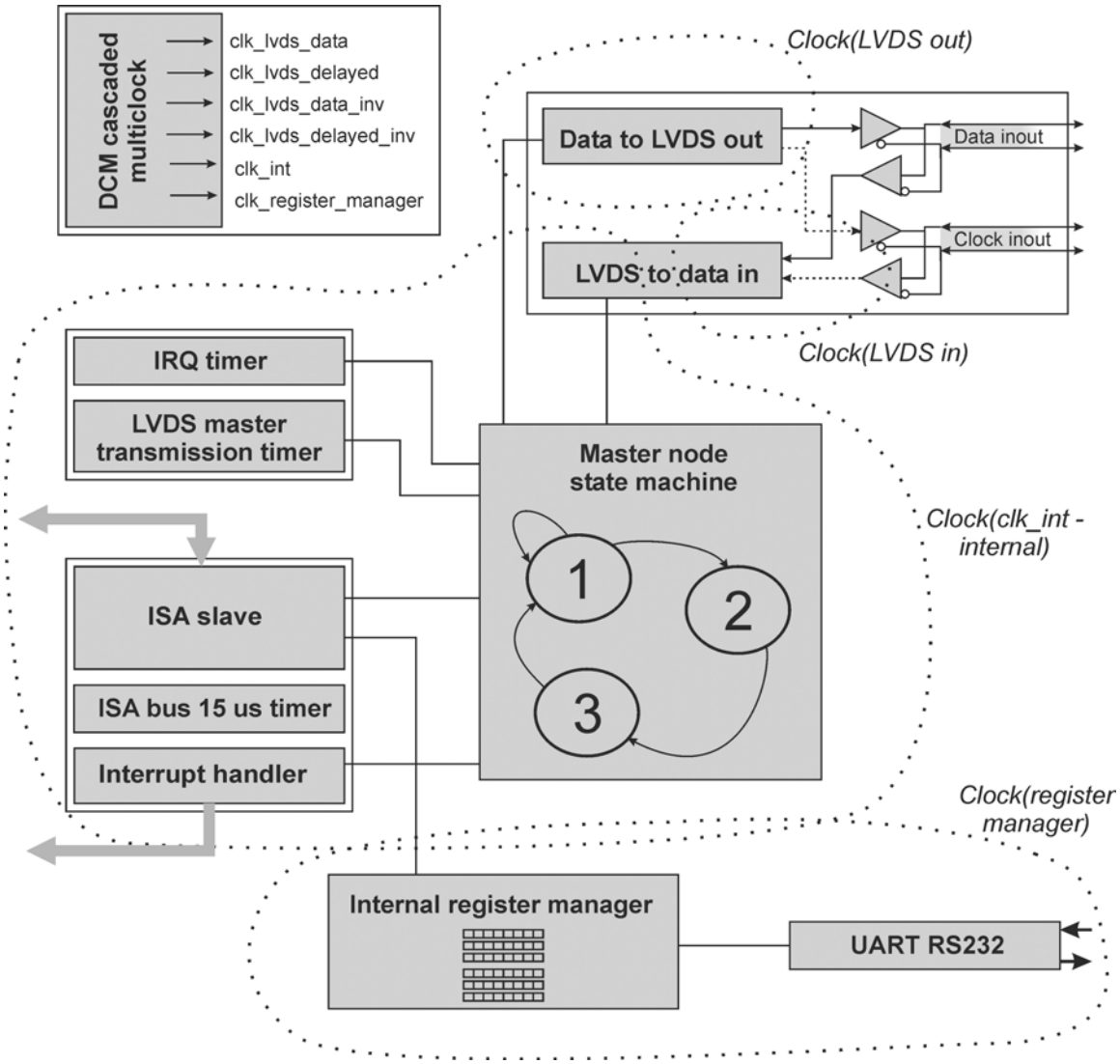


Figure 3-16: Interconnection diagram of the main internal modules in the master node FPGA.

The design is built in a hierarchal manner using several components. The components high up in the hierarchy are easily combined into a design and are therefore called modules. Some of these components are constructed from components that are more general. The component hierarchy and the components are found in the appendix.

3.5.1 The communication flow in the master node

Figure 3-16 shows the master nodes internal configuration, coordinating the communication between the nodes in the network. The command, data and address signals from the host computer is first analysed by the ISA slave module. The ISA slave module will receive and check the address. If the distributed system is addressed it will hold the bus until the distributed system has finished the transfer. If an internal register is addressed the data is sent to the internal register manager. If one of the slave nodes is addressed the data is sent to the state machine. The state machine uses the LVDS modules to communicate with the slave nodes and the timers to time coordinate the communication.
The master node state machine is implemented in VHDL and is a synchronous Mealy type state machine with synchronous outputs. This means that all the logic and outputs are synchronised with the clock driving the state machine. The state machine communicates with other modules using asynchronous command signals, which makes it possible to let the modules use different clock domains.
Figure 3-17 shows a state diagram of the state machine. The state machine will start in the idle state (ground state S0) after a reset. The text in every state circle explains the function of the state and the text by the arrow between the states indicates the jump condition. For detailed information of the state machine, consider the VHDL program code in Appendix B.

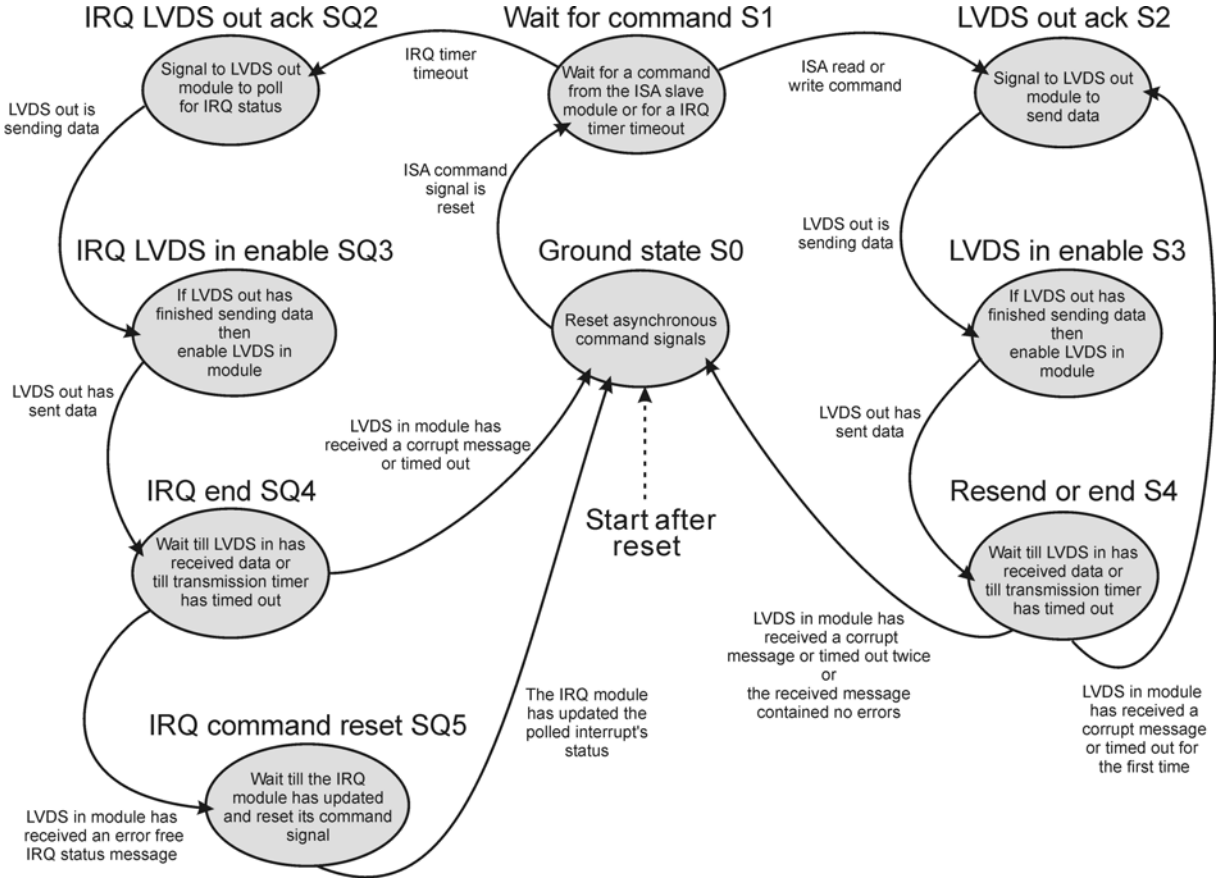


Figure 3-17: State diagram of the master node state machine.

Below the communication-flow in the state machine is explained. Both Figure 3-16 and Figure 3-17 are helpful for understanding the explanations.

3.5.1.1 Reading and writing data to a slave node

1. **Host computer:** First, the host computer sends a write or read command accompanied with data and address to the ISA bus on the host computer.
2. **ISA slave:** The ISA slave module in the master node will receive the message and check the address. If the distributed system is addressed it will hold the bus until the distributed system has finished the transfer. If a slave node is addressed the ISA slave will signal to the state machine if it is a read or write command.
3. **State S1:** The state machine waits in this state to receive a command from the ISA slave node. When the command is received, the state machine enables the LVDS output buffer and jumps to next state.
4. **State S2:** LVDS out module is signalled to send the data and address along with the read or write command. The state machine will jump to next state when the LVDS module signals that it is busy sending data.
5. **State S3:** When the LVDS out module is signalling that it has sent the data block and is ready, the output buffer will be disabled and the LVDS in module will be enabled to wait for a response from a slave node. The state machine will also jump to its next state when the data is sent.
6. **State S4:** Here the state machine waits till the LVDS in module has received the data from the addressed slave node. If the data is corrupt or if the transmission timer times out, the data will be sent again by enabling the output buffer and jumping to state S2. If the data is corrupt or the timer times out a second time, or if the data is received correctly, the state machine will jump to state S0 and signal to the ISA slave module to end the transmission cycle.
7. **State S0:** In this state the state machine will wait for the ISA slave module to reset the command signals. When the read and write command is reset the signals used by the state machine will reset and the state machine will jump to state S1.

3.5.1.2 ISA interrupt polling

When no data transfer is occupying the bus, the IRQ timer will time-out after a specified time. The state machine will then poll the slave nodes for one interrupt at a time. If one slave node has registered an interrupt corresponding to the polled interrupt number, it will reply to the master node that this interrupt has been requested. The state machine will act according to the following list:

1. **State S1:** If no command from the ISA slave is received during a specified time, the IRQ timer will time-out. The state machine will then enable the LVDS output buffer, assert the interrupt poll info bit in the data block, and jump to state SQ2.
2. **State SQ2:** Here the LVDS out module will be signalled to send the data block. The state machine will jump to next state when the LVDS module signals that it is busy sending data.
3. **State SQ3:** When the LVDS out module is signalling that it has sent the data and is ready, the output buffer will be disabled and the LVDS in module will be enabled to wait for a response from a slave node in the next state.
4. **State SQ4:** Here the state machine waits until the LVDS in module has received a message from a slave node. Only if a slave has received an interrupt request from its distributed ISA bus, it will reply to the master. If no slave node has received an interrupt, no reply will be sent and the LVDS master transmission timer will time out. If the data is corrupt or if the transmission timer times out, the state machine will end the interrupt poll cycle by jumping to state S0. If the data block is received correctly, the interrupt status is sent to the interrupt handler module (IRQ out) and the state machine jumps to state SQ5.
5. **State SQ5:** In this state the state machine waits for the interrupt handler to finish the possible interrupt signalling on the ISA bus and then it jumps to state S0.

3.5.2 Timing and area constraints

The design can be optimised by the use of timing and area constraints. These are set in the Xilinx ISE development tool. The used clock frequencies should be specified in the timing constraint editor. The high-speed data paths, operating temperature and supply voltage should also be specified. Table 3-5 below explains the constraints chosen for this design.

Constraint	Explanation
VOLTAGE = 1.2;	This is the operating voltage for the internal logic on the Spartan 3 starter kit board.
TEMPERATURE = 50 ;	Estimated operating temperature of the FPGA in Celsius degrees.
NET "lvds_clk_p" TNM_NET = "lvds_clk_p"; TIMESPEC "TS_lvds_clk_p" = PERIOD "lvds_clk_p" 100 MHz HIGH 50 % INPUT_JITTER 0.5 ns;	The input frequency of the LVDS data clock received from the slave node with an estimated jitter of 0.5 ns.
NET "clk50" TNM_NET = "clk50"; TIMESPEC "TS_clk50" = PERIOD "clk50" 50 MHz HIGH 50 %	The input frequency of the input clock feeding the DCMs in the FPGA.
NET "CLK_LVDS_DATA" USELOWSKEWLINES;	Use low skew in the LVDS clock from the DCM
NET "CLK_LVDS_DATA_INV" USELOWSKEWLINES;	Use low skew in the LVDS clock from the DCM
NET "CLK_LVDS_DELAYED" USELOWSKEWLINES;	Use low skew in the LVDS clock from the DCM
NET "CLK_LVDS_DELAYED_INV" USELOWSKEWLINES;	Use low skew in the LVDS clock from the DCM
NET "Data_from_lvds_driver" USELOWSKEWLINES;	Use low skew in the LVDS data input path.
NET "clock_from_lvds_driver" USELOWSKEWLINES;	Use low skew in the LVDS clock input path.
NET "Data_to_lvds_driver" USELOWSKEWLINES;	Use low skew in the LVDS data output path.
NET "clock_to_lvds_driver" USELOWSKEWLINES;	Use low skew in the LVDS clock output path.

Table 3-5: Master node timing constraints.

Area constrain could be used to manually decide where to put the logic or to try to fit all the logic into a smaller FPGA. In this design, all the logic fit into the FPGA chosen and because it may be very hard to manually place the design with a better result than achieved by the ISE placer, no area constraints are specified in this design.

3.5.3 FPGA resource utilisation

The implemented master node FPGA utilisation is shown in the Table 3-6. The global clock nets are all used and two out of four DCMs are used. Many I/O-pins are also used. On the other hand, only one fourth of the Slices are used. There are much BRAM left as well.
This means that lots of logic may still be implemented in the FPGA. The I^2C controller occupies 7% of the slices and the RS232 controller occupies 3% of the slices. The internal register manager occupies about 5 % of the Slices. Because the register manager will grow bigger by the use of more registers one may assume that at least 15 more RS232 controllers or at least 7 more I^2C controllers may be implemented in the FPGA, if needed.

Logic utilisation	Used	Available	Utilisation
Number of Slices	454	1920	23%
Number of Slice Flip Flops	501	3840	13%
Number of 4 input LUTs	783	3840	20%
Number of bonded IOBs	81	173	46%
Number of BRAMs	1	12	8%
Number of GCLKs	8	8	100%
Number of DCM_ADVs	2	4	50%

Table 3-6: Master node utilisation summary using the Spartan 3 xc3s200 FPGA device with the ft256 package.

3.6 The slave node

The slave node has several similarities with the master node. The logic is in this design programmed in one Spartan 3 FPGA. A central state machine, called the slave node state machine, handles the communication between the modules as shown in Figure 3-18. In other tested designs, two FPGAs are used together with one microcomputer. The devices in the slave node communicate between each other via an I^2C bus.

The state machine acts as a slave on the LVDS link using the LVDS in and LVDS out modules. The dotted areas in the figure marks out the four different clock domains in the master node. This makes it possible to run the high-speed parts such as the LVDS modules at a higher frequency while the some slower logic may run at a lower frequency.

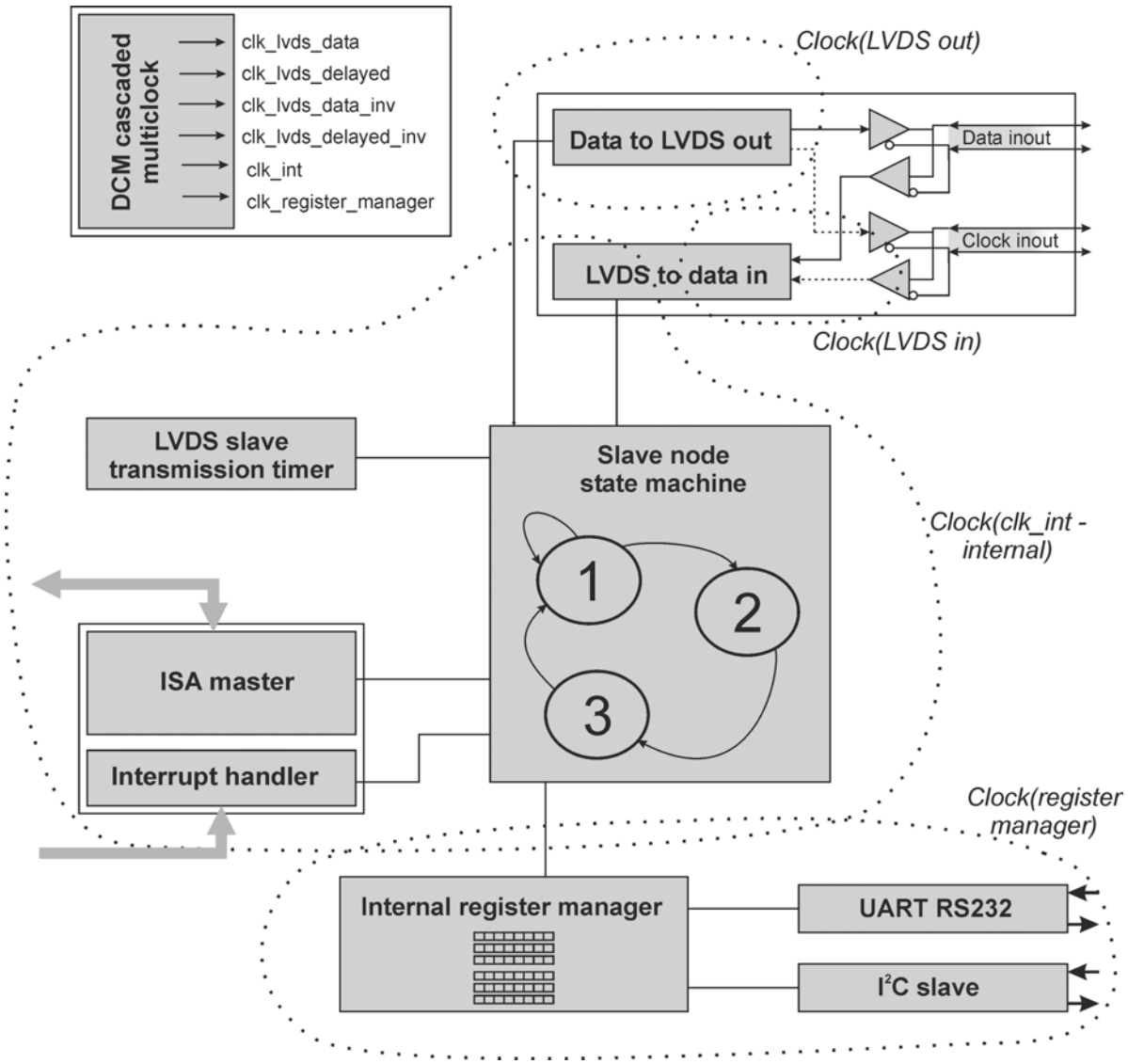


Figure 3-18: Interconnection diagram of the main internal modules in the slave node FPGA.

The component hierarchy and the components in the slave node design are found in the appendix.

3.6.1 The communication flow in the slave node

Figure 3-18 shows the slave node state machine coordinating the communication between the modules. First, the LVDS in module receives a message containing command, data and address. The state machine checks the message and if the address corresponds to this specific slave node, it sends the data to the addressed module. When the addressed module has ended its transmission, it signals the state machine. The state machine then signals the LVDS out module, to send a reply message to the master node.
The slave node state machine is implemented as the master node state machine. The state machine communicates with other modules using asynchronous command signals, which makes it possible to let the modules use different clock domains.
Figure 3-19 shows a state diagram of the state machine. The state machine will start in the ground state after a reset. The text in every state circle explains the function of the state and the text by the arrow between the states indicates the jump condition. For detailed information of the state machine, consider the VHDL program code in the appendix.

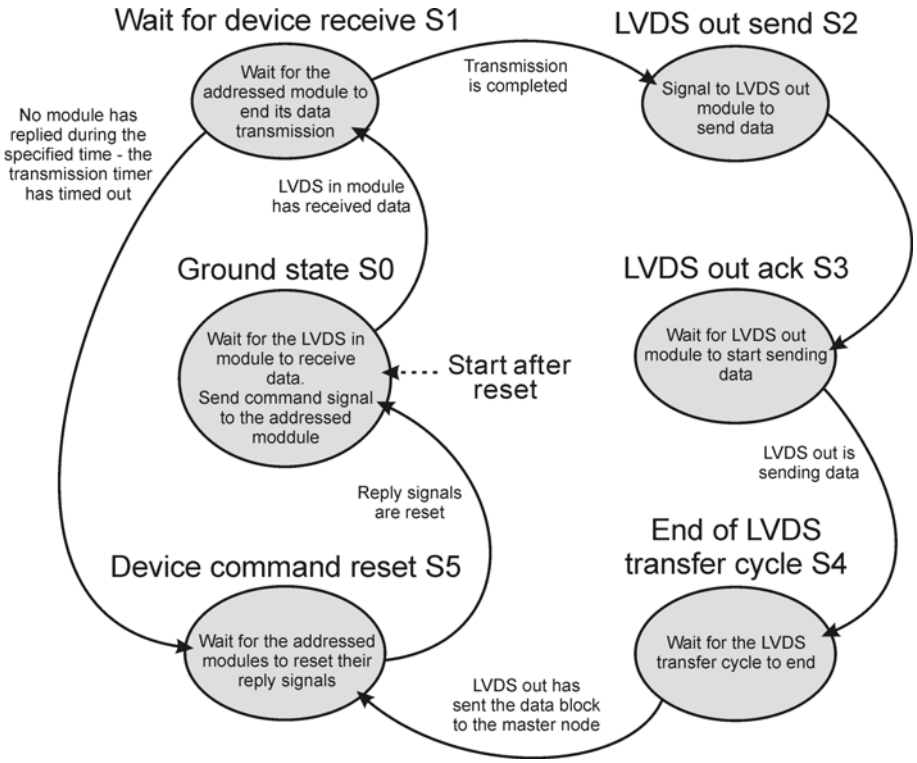


Figure 3-19: State diagram of the slave node state machine.

Below the communication-flow in the state machine is explained. Both Figure 3-18 and Figure 3-19 are helpful for understanding the explanations. Some different access scenarios can occur. The internal registers or the distributed ISA bus can be read from or written to. The master node can also poll for interrupt status. All of these scenarios are explained below:

1. **State S0:** First, the state machine waits for the LVDS in module to receive a data block. If the address corresponds to the slave node, it signals to the addressed module to perform the requested access. Then it jumps to state S1.
2. **State S1:** In this state the state machine waits for the addressed module to perform the requested access. The addressed module could be the internal register manager, the ISA master or the interrupt handler, IRQ in. When the module signals that it has performed the requested access and is ready, the state machine enables the LVDS output buffer and jumps to state S2.

3. **State S2:** Here the LVDS out module is signalled to send the respond message from the addressed module back to the master node. If it was a read access, the respond message will contain the read data. If it was an interrupt poll access, the respond message will contain the status of the polled interrupt. If it was a write access, the respond message will just be sent as an acknowledge message. Then the state machine will jump to state S3.
4. **State S3:** Here the state machine waits for the LVDS out module to signal that it is busy sending data. When this signal is received, the state machine will jump to state S4.
5. **State S4:** This state waits for the LVDS out module to finish sending data. When the data block is sent the state machine disables the LVDS output buffer and jumps to state S5.
6. **State S5:** In this state the state machine waits for the internal modules to reset their communication signals. When these signals are reset the state machine can end this access cycle and jump to state S0.

3.6.2 Timing and area constraints

As for the master node, the design can be optimised by the use of timing and area constraints. The Table 3-7 below explains the constraints chosen for this design.

Constraint	Explanation
VOLTAGE = 1.2;	This is the operating voltage for the internal logic on the Spartan 3 starter kit board.
TEMPERATURE = 50 ;	Estimated operating temperature of the FPGA in Celsius degrees.
NET "lvds_clk_p" TNM_NET = "lvds_clk_p"; TIMESPEC "TS_lvds_clk_p" = PERIOD "lvds_clk_p" 100 MHz HIGH 50 % INPUT_JITTER 0.5 ns;	The input frequency of the LVDS data clock received from the slave node with an estimated jitter of 0.5 ns.
NET "clk50" TNM_NET = "clk50"; TIMESPEC "TS_clk50" = PERIOD "clk50" 50 MHz HIGH 50 %	The input frequency of the input clock feeding the DCMs in the FPGA.
TIMESPEC "TS_I2C_SCLK_in" = PERIOD "I2C_SCLK_in" 1 MHz HIGH 50 %;	The input frequency of the input clock from the I^2C bus.
NET "CLK_LVDS_DATA" USELOWSKEWLINES;	Use low skew in the LVDS clock from the DCM
NET "CLK_LVDS_DATA_INV" USELOWSKEWLINES;	Use low skew in the LVDS clock from the DCM
NET "CLK_LVDS_DELAYED" USELOWSKEWLINES;	Use low skew in the LVDS clock from the DCM
NET "CLK_LVDS_DELAYED_INV" USELOWSKEWLINES;	Use low skew in the LVDS clock from the DCM
NET "Data_from_lvds_driver" USELOWSKEWLINES;	Use low skew in the LVDS data input path.
NET "clock_from_lvds_driver" USELOWSKEWLINES;	Use low skew in the LVDS clock input path.
NET "Data_to_lvds_driver" USELOWSKEWLINES;	Use low skew in the LVDS data output path.
NET "clock_to_lvds_driver" USELOWSKEWLINES;	Use low skew in the LVDS clock output path.

Table 3-7: Slave node timing constraints.

3.6.3 FPGA resource utilisation

The implemented slave node FPGA utilisation is shown in Table 3-8. The slave node utilisation is similar to the master node. This means that lots of logic may still be implemented in the FPGA. The I^2C controller is only implemented in the slave node and occupies 7% of the Slices. Because the register manager will grow bigger by the use of more registers one may assume that at least 7 more I^2C controllers may be implemented in the FPGA.

Logic utilisation	Used	Available	Utilisation
Number of Slices	532	1920	27%
Number of Slice Flip Flops	653	3840	17%
Number of 4 input LUTs	925	3840	24%
Number of bonded IOBs	95	173	54%
Number of BRAMs	1	12	8%
Number of GCLKs	8	8	100%
Number of DCM_ADVs	2	4	50%

Table 3-8: Slave node utilisation summary using the Spartan 3 xc3s200 FPGA device with the ft256 package.

3.7 Physical channel distribution and voltage levels

3.7.1 Voltage levels

On the Spartan 3 starter kit board and on the Hectronic H4070 board, the FPGA is fed with three voltage levels. These are V_{CCO} = 3.3 V, V_{CCAUX} = 2.5 V and V_{CCINT} = 1.2 V.
The V_{CCO} supplies the I/O banks, the V_{CCAUX} supplies the DCMs and some I/O structures, and the V_{CCINT} supplies the internal logic within the FPGA [7].
The BLVDS differential standard chosen to drive the LVDS link has a common mode voltage of 1.25 V and a voltage swing of 350 mV.
The digital output signals from the slave node FPGA to the distributed ISA bus has the voltage levels 3.3 V or 0 V. The signals on the host computer bus are connected to the master node. The only output signals used here are the CHRDY and the IO16# signals. These signals are never driven high, they are either high impedance or driven low to 0 V.

3.7.2 LVDS link distribution

The two LVDS channels are transmitted in a CAT5 type cable. The bus configuration is a source synchronous clock design. Four wires in the cables are used forming two LVDS pairs. One holding the source generated clock signal and the other the data signal. Only two terminating resistors of usually 50 to 100 Ω are used to terminate each channel in both ends. More information about the set-up is in the design evaluation chapter.

3.8 Timing analysis of the distributed network

In this section, theoretical timing diagram of the communication in the distributed network are presented.

3.8.1 Access time for a device on the distributed ISA bus

The timing diagram in Figure 3-20 shows the host computer making a read access from a device on the distributed ISA bus. The ISA bus clock is 8.33 MHz and the LVDS baud rate is 200 Mbaud. The generated clock signal driving the FPGA has a frequency of 50 MHz. Note that the width of the time periods marked out by capital letters are not in proportion to the actual time elapsed in that period. The actual time elapsed are presented further down.

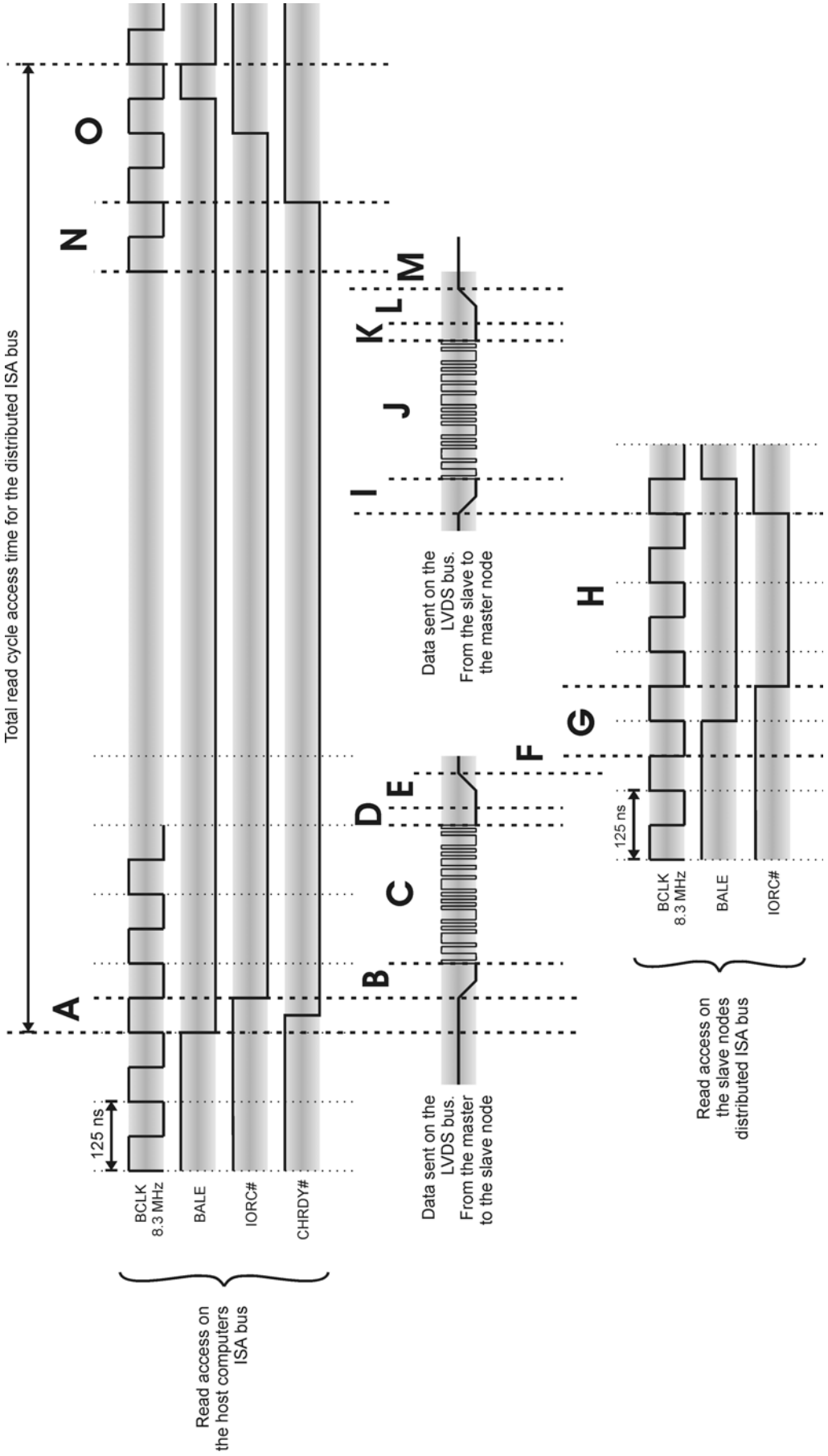


Figure 3-20: The timing diagram shows a distributed ISA bus access made from the host computer.

The letters in the list below corresponds to the letters in the figure, marking different time-periods.

A. **67 + 0..20 ns:** The variable delay of 0 to 20 ns depends on the ISA bus clock phase relative to the 50 MHz internal sampling clock. The 67 ns is the time between the address latch BALE and the read command.
B. **60 + 0..20 ns:** This delay depends on the internal pipe-lined logic in the LVDS out module that has to load the DDR and the shift registers.
C. **205 ns:** The time it takes to send 41 bits of data on the LVDS link, at 200 Mbps.
D. **20 ns:** The delay in the DDR and the shift registers in the LVDS in module.
E. **20 + 0..20 ns:** Time for the error code checker component to check the data block and to synchronise it to the internal clock.
F. **0..125 ns:** The time to synchronise the internal clock to the ISA bus clock on the slave node.
G. **125 ns:** Time before asserting the ISA bus read command.
H. **192 or 562 ns:** The command is asserted 192 ns for an ordinary 16-bit read access and 562 ns for an ordinary 8-bit access.
I. **40 + 0..30 ns:** The internal logic delay in the slave node state machine and in the LVDS out module.
J. **205 ns:** The time it takes to send 41 bits of data at 200 Mbps, on the LVDS link.
K. **20 ns:** The delay in the LVDS in module.
L. **20 + 0..20 ns:** The time for the error code checker component to check the data block and to synchronise it to the internal clock.
M. **20 ns:** The time for the ISA slave module to receive the transmission_ok signal from the master node state machine.
N. **67 + 0..192:** The delay between putting the data on the bus and to assert the ISA CHRDY signal. This delay can probably be optimised to 0..125 ns.
O. **312 ns:** The time taken for the host computer to end the ISA bus cycle and to be able to start a new access again.

Several of the delays above can be optimised. The delay marked with the letter J where the data block is sent from the slave node to the master node, could be optimised to 105 ns. This is done by only sending the requested 16 data bits and four bits of error checking code. If the start bit is included, only 21 bits need to be transmitted instead of 41. The delay marked with the letter N, where the data is put on the ISA bus some time period earlier than the CHRDY signal is asserted, could be optimised to 0..125 ns. The delay was introduced as a safety measure so that the data bits would have plenty of time to stabilise before the bus cycle would end. This is probably not necessary and the CHRDY signal could be asserted at the same time when the data is put on the bus.
When the time delays in the list above are summarised, the result in Table 3-9 is achieved. The ordinary ISA bus cycle times are also in the table as well as the possible optimised access times. The results presented in the table shows that the access time for the distributed ISA bus takes three to four times longer than for the ordinary ISA bus.

Type of access	Shortest time	Longest time
16-bit read access	1370 ns	1800 ns
8-bit read access	1740 ns	2170 ns
Optimised 16-bit read access	1200 ns	1570 ns
Optimised 8-bit read access	1570 ns	1940 ns
Ordinary 16-bit read access	375 ns	-
Ordinary 8-bit read access	750 ns	-

Table 3-9: The access time to the distributed ISA bus compared to the ordinary ISA bus access.

3.8.2 Access time for an internal register

The time to access a register is faster than to make a distributed ISA bus access. The time to access a register in the slave node is roughly 400 ns shorter than the distributed 16-bit access. The time to access a register in the master node is the same as for the ordinary 16-bit access, i.e. 375 ns.

Chapter 4 – Results

4.1 Software related trouble shooting and solved problems

The development of this design has had many difficult stages where the code had to be analysed and tested several times to achieve the wanted circuit behaviour. Some of these problems and their solutions are presented in this section.

4.1.1 Generation of high speed LVDS in a low cost FPGA

Because the speed grade in the FPGA used is -4, it is quite hard to generate high-speed data. The modules requiring high-speed are especially the LVDS in and LVDS out modules. To optimise these modules for speed, careful programming using the schematic editor and device specific logic has been made. The resulting transfer rate achieved is around 400 Mbaud according to the post-place-and-route timing analyser tool. The practical achieved data speed is limited by reflections in the LVDS bus configuration as discussed in section 4.4.2.

4.1.2 Synchronising asynchronous signals before entering a state machine

When a synchronous state machine is implemented, it is very important to synchronise the input signals. The input signals that are used in a jump-condition in the state machine are not allowed to change when the state machine is performing the jump. If this occurs the state machine may jump to wrong state or to an undefined state. Synchronising the input signals by the use of flip-flops before entering the state machine easily solves this.

4.1.3 Generation of high-speed clock signals

The generation of high-speed clock signals has been a big issue. There are several ways to do this and the best way, in this case, is to use the IP-core including two cascaded DCMs. Because the LVDS transmitter requires four signals shifted 90 degrees between each other, it is quite hard to generate them with high quality. The maximum frequency of the four clock signals achieved are 200 MHz generated from a 100 MHz input clock. For a long time, during the development, only 50 MHz of high quality signals could be achieved, resulting in a baud rate of 100 Mbaud. Later on, after trying lots of DCMs and other design configurations, stable communication at 200 Mbaud was achieved.

4.2 Timing analysis of the design

The figures below show the timing summary of the design. The timing summary shows the estimated timing after the synthesis and the post place and route timing report shows the estimated timing after the design has been routed in the FPGA.
The timing summary of the LVDS link out component is shown in Figure 4-1. The requested clock rate is 150 MHz and will generate a baud rate of 300 Mbaud. The estimated (actual) value is 4.75 ns at most and this will accept an input clock signal of 210 MHz resulting in a maximum LVDS baud rate of 420 Mbaud.

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 5.078ns (Maximum Frequency: 196.928MHz)
   Minimum input arrival time before clock: 2.715ns
   Maximum output required time after clock: 7.165ns
   Maximum combinational path delay: No path found


Multi pass post place and route constraint timing report:
-------------------------------------------------------

  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
  TS_lvds_data_clk = PERIOD TIMEGRP "lvds_d | 6.666ns    | 4.748ns    | 3
  ata_clk" 150 MHz HIGH 50%                 |            |            |
--------------------------------------------------------------------------------
  TS_lvds_data_clk_inv = PERIOD TIMEGRP "lv | 6.666ns    | 4.112ns    | 1
  ds_data_clk_inv" 150 MHz HIGH 50%         |            |            |
--------------------------------------------------------------------------------

All constraints were met.
```

Figure 4-1: Timing analysis of the LVDS link out component.

The timing summary of the LVDS link in component is shown in Figure 4-2. The requested clock rate is 150 MHz and will result in a baud rate of 300 Mbaud, just as for the LVDS link out component. The estimated (actual) value is 5.63 ns at most and this will accept an input clock signal of 175 MHz resulting in a maximum LVDS baud rate of 350 Mbaud, supported by this component.

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 6.381ns (Maximum Frequency: 156.715MHz)
   Minimum input arrival time before clock: 1.901ns
   Maximum output required time after clock: 7.281ns
   Maximum combinational path delay: No path found


Multi pass post place and route constraint timing report:
-------------------------------------------------------

  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
  TS_lvds_data_clk_inv = PERIOD TIMEGRP "lv | N/A        | N/A        | N/A
  ds_data_clk_inv" 150 MHz HIGH 50%         |            |            |
--------------------------------------------------------------------------------
  TS_lvds_data_clk = PERIOD TIMEGRP "lvds_d | 6.666ns    | 5.632ns    | 0
  ata_clk" 150 MHz HIGH 50%                 |            |            |
--------------------------------------------------------------------------------

All constraints were met.
```

Figure 4-2: Timing analysis of the LVDS link in component.

The timing summary of the slave node top-level design is shown in Figure 4-3. The input clock (TS_XLXI_107_U1_CLK0_BUF) is 50 MHz corresponding to 20 ns requested value. The requested value for the LVDS input and output signal (TS_lvds_clk_p) are 100 MHz (10 ns). The I^2C input clock is named TS_I2C_SCLK_in. The clock signals generating the data in the LVDS out module is named TS_XLXI_107_U2_CLK0_BUF and TS_XLXI_107_U2_CLK180_BUF. The low frequency clock signal of 25 MHz, that is driving slower logic in the FPGA, is named TS_XLXI_107_U1_CLKDV_BUF. All of the signals mentioned above are set with an appropriate constraint. All of the constraints have been met.

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 14.232ns (Maximum Frequency: 70.264MHz)
   Minimum input arrival time before clock: 6.153ns
   Maximum output required time after clock: 8.912ns
   Maximum combinational path delay: No path found


Multi pass post place and route constraint timing report:
---------------------------------------------------------

  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
  TS_lvds_clk_p = PERIOD TIMEGRP "lvds_clk_ | 10.000ns   | 5.634ns    | 0
  p" 100 MHz HIGH 50% INPUT_JITTER 0.5 ns   |            |            |
--------------------------------------------------------------------------------
  TS_I2C_SCLK_in = PERIOD TIMEGRP "I2C_SCLK | 1000.000ns | 16.310ns   | 4
  _in" 1 MHz HIGH 50%                       |            |            |
--------------------------------------------------------------------------------
  TS_XLXI_107_U1_CLK0_BUF = PERIOD TIMEGRP  | 20.000ns   | 11.328ns   | 2
  "XLXI_107_U1_CLK0_BUF" TS_clk50 HIGH      |            |            |
      50%                                   |            |            |
--------------------------------------------------------------------------------
  TS_XLXI_107_U1_CLKDV_BUF = PERIOD TIMEGRP | 40.000ns   | 24.596ns   | 5
   "XLXI_107_U1_CLKDV_BUF" TS_clk50 / 2     |            |            |
       HIGH 50%                             |            |            |
--------------------------------------------------------------------------------
  TS_XLXI_107_U2_CLK0_BUF = PERIOD TIMEGRP  | 10.000ns   | 6.724ns    | 3
  "XLXI_107_U2_CLK0_BUF"         TS_XLXI_10 |            |            |
  7_U1_CLK2X_BUF HIGH 50%                   |            |            |
--------------------------------------------------------------------------------
  TS_XLXI_107_U2_CLK180_BUF = PERIOD TIMEGR | 10.000ns   | 8.364ns    | 2
  P "XLXI_107_U2_CLK180_BUF"         TS_XLX |            |            |
  I_107_U1_CLK2X_BUF PHASE 5 ns HIGH 50%    |            |            |
--------------------------------------------------------------------------------

All constraints were met.
```

Figure 4-3: Timing analysis of the slave node top-level design.

The timing summary of the master node top-level design is shown in Figure 4-4. The input clock (TS_XLXI_123_U1_CLK0_BUF) is 50 MHz corresponding to 20 ns requested value. The requested value for the LVDS input and output signal (TS_lvds_clk_p) are 100 MHz (10 ns). The clock signals generating the data in the LVDS out module is named TS_XLXI_123_U2_CLK0_BUF and TS_XLXI_123_U2_CLK180_BUF. The low frequency clock signal of 25 MHz, that is driving slower logic in the FPGA, is named TS_XLXI_123_U1_CLKDV_BUF. All of the signals mentioned above are set with an appropriate constraint. All of the constraints have been met.

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 13.468ns (Maximum Frequency: 74.250MHz)
   Minimum input arrival time before clock: 3.847ns
   Maximum output required time after clock: 8.912ns
   Maximum combinational path delay: No path found


Multi pass post place and route constraint timing report:
---------------------------------------------------------

  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
  TS_lvds_clk_p = PERIOD TIMEGRP "lvds_clk_ | 10.000ns   | 7.038ns    | 0
  p" 100 MHz HIGH 50% INPUT_JITTER 0.5 ns   |            |            |
--------------------------------------------------------------------------------
  TS_XLXI_123_U1_CLK0_BUF = PERIOD TIMEGRP  | 20.000ns   | 14.650ns   | 1
  "XLXI_123_U1_CLK0_BUF" TS_clk50 HIGH      |            |            |
      50%                                   |            |            |
--------------------------------------------------------------------------------
  TS_XLXI_123_U1_CLKDV_BUF = PERIOD TIMEGRP | 40.000ns   | 20.524ns   | 4
   "XLXI_123_U1_CLKDV_BUF" TS_clk50 / 2     |            |            |
       HIGH 50%                             |            |            |
--------------------------------------------------------------------------------
  TS_XLXI_123_U2_CLK0_BUF = PERIOD TIMEGRP  | 10.000ns   | 5.405ns    | 1
  "XLXI_123_U2_CLK0_BUF"         TS_XLXI_12 |            |            |
  3_U1_CLK2X_BUF HIGH 50%                   |            |            |
--------------------------------------------------------------------------------
  TS_XLXI_123_U2_CLK180_BUF = PERIOD TIMEGR | 10.000ns   | 9.034ns    | 2
  P "XLXI_123_U2_CLK180_BUF"         TS_XLX |            |            |
  I_123_U1_CLK2X_BUF PHASE 5 ns HIGH 50%    |            |            |
--------------------------------------------------------------------------------

All constraints were met.
```

Figure 4-4: Timing analysis of the master node top-level design.

4.3 Evaluation software

A test program on the host computer was specially programmed by a colleague of mine to test the network. This program was programmed in C language and was able to read and write both 8-bit and 16-bit values to a chosen address on the ISA bus. The program checked if read data was the same as the data written. If errors occurred, an error report was generated presenting the written and read value that differed. The program was usually set to do thousands of accesses in a row. Values were written to different registers in the network to test the RS232, I^2C and ISA bus interfaces at the network nodes. When a LVDS link is referred to have a certain transfer speed, in the sections below, several thousands accesses have been made using this program without any error reported.

4.4 Hardware related trouble shooting and solved problems

4.4.1 ISA reset and IO16# signals

The I016# and reset signals on the ISA bus have to be driven with more current than the other signals on the ISA bus. In the set-up used, all the signals on the ISA bus is connected to the FPGA via 100 Ω resistors. This works well for all the signals except the two mentioned. Some computers and devices have strong pull-ups on these signals and the resistors used for these signals have to be modified to 30 Ω so that the signal will be fast enough.

4.4.2 LVDS bus termination problems

The termination of the LVDS bus is of great importance when getting the communication to work. If the network is not terminated correctly the signals will be corrupt due to reflections. In the tested configurations below this has been a big issue. Even tough the point-to-point link configuration works perfectly, other network configurations have been hard to stabilise. Consider the different test set-ups below for a more detailed explanation.

4.4.3 Stabilising the LVDS channel

When no slave or master node is driving the LVDS link, the potential in the two wires are not stable. This results in some false message generation. To avoid this, internal pull-up and pull-down resistors are used in the master node FPGA. Usually external resistors are used, and a more optimal value could then be chosen. If problems with the communication are found, external resistors that are more accurate to the current design should be used.

4.4.4 FPGA breakdown

When using the on-chip LVDS drivers in different set-ups the logic in the FPGAs sometimes got damaged due to problems with the LVDS links. The damaged logic in the FPGA made it very hot and the FPGA had to be discarded. The breakdown of the FPGAs only occurred when several FPGAs was connected to the same bus using the multipoint configuration.
The reason for this could be reflections on the LVDS bus due to bad termination. If a transceiver is sending a message on the bus and gets a reflection, the input buffer close to the transmitter will receive both the sent voltage level plus the reflected one. If the reflection is strong enough, the added voltage levels may damage the FPGA. Sometimes, even when the terminations were chosen very carefully, the FPGAs got destroyed.
The reason for breakdown could also be an unstable LVDS bus causing two different transmitters to drive the bus at the same time. If the bus is unstable, the different slave nodes will receive noise-generated messages at random. Eventually the error checking code will match the randomly received data bits and if at the same time the address bits match the slave node, a respond message will be generated. If another slave node or if the master node generates a message at the same time, two transmitters would be driving the bus at the same time, destroying each other.
Generally a multipoint LVDS bus is very hard to terminate correctly [24]. When the multipoint configuration was tested, two Hectronic H4070 boards were used and a short cable connected the boards together. Unmatched connectors and on-board clock signals could make the LVDS bus unstable, generating reflections and false messages.

4.5 Hardware evaluation of the design

4.5.1 Oscilloscope plots of ISA bus transmissions

Figure 4-5 shows a distributed ISA bus access generated by the host computer by the use of the IOWC# signal. It also shows the CHRDY signal generated by the distributed network. The CHRDY signal is held low until the distributed ISA bus access has ended. The distributed network then releases the signal and the host computer ends the bus cycle. Figure 4-6 shows an interrupt request generated by the distributed network. An interrupt is requested by holding the interrupt line low for a short time and then release it again. In this design, the interrupt request signal is chosen to be held low for 60 ns[I] according to the oscilloscope measurement in Figure 4-6.

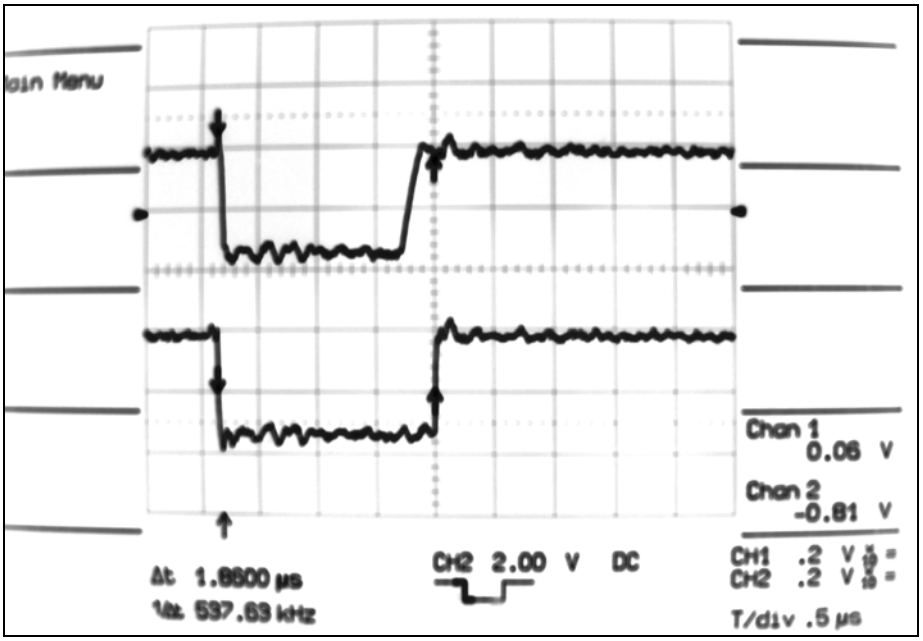


Figure 4-5: An ISA bus access from the host computer to the master node in the distributed network. The upper channel displays the CHRDY signal and lower channel shows the IOWC# generated by the host computer.

[I] 60 ns corresponds to three 50 MHz clock cycles.

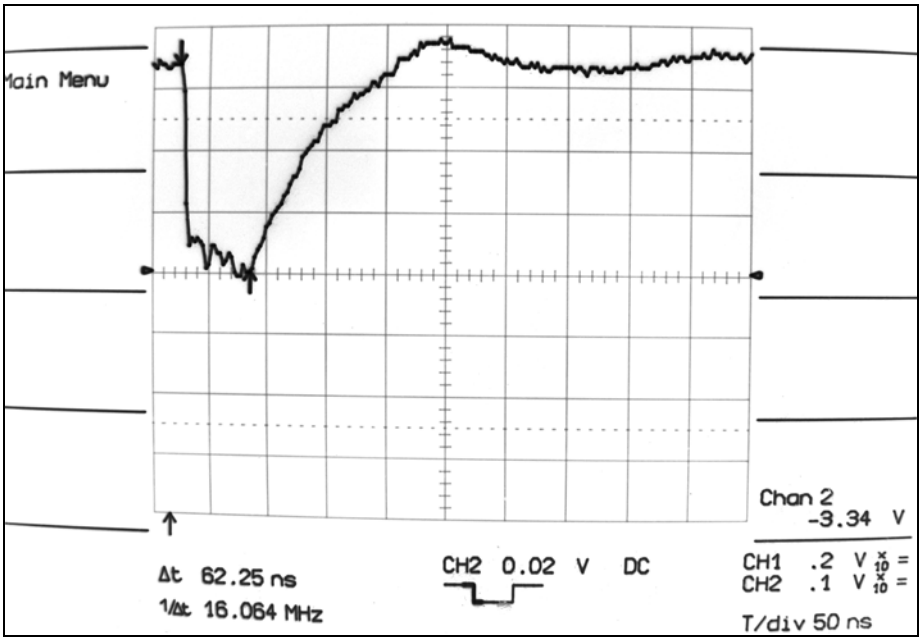


Figure 4-6: The channel shows an interrupt request generated by the distributed network.

4.5.2 Oscilloscope plots of LVDS bus transmissions

Because LVDS is a high-speed signal with frequencies up to 200 MHz in this project, the highest data rates could not be analysed by the oscilloscope at hand. Therefore, the signal frequency shown in the figures below are of only 25 MHz. Preliminary the plots below are presented to demonstrate the transmission using DDR and a source synchronous clock.
Figure 4-7 shows how data is transmitted. One data bit is sampled at every rising and falling edge of the clock signal. For example in this figure the data sequence 0-0-1-1-1-1-0-1-0-1 is transmitted.

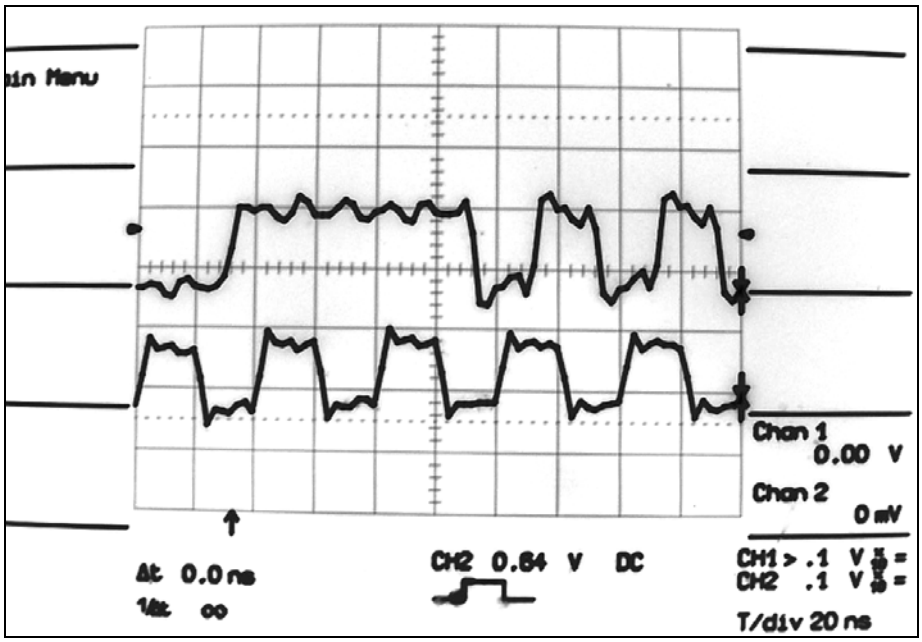


Figure 4-7: LVDS signals using DDR. This means that one data bit is sampled at every rising and falling edge of the clock signal. The upper channel shows the data signal and the lower channel shows the clock signal.

In Figure 4-8 a 41-bit data access message is sent. In this figure no reply message is sent. When a message is sent the LVDS driver will be set in a high impedance state as shown in Figure 4-9. If a slave node with the accessed address exists on the bus, it will reply to the master node as shown in Figure 4-10. Here the reply message is sent very short after the access because an internal register in the slave node was accessed. Notice the high impedence state (tri-state) in the middle of the figure when no driver is driving the bus.

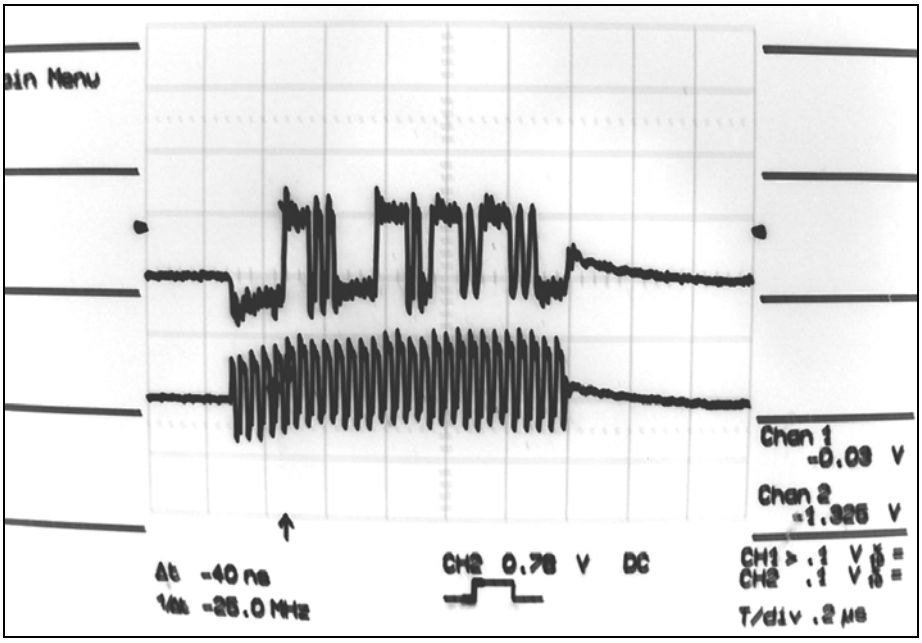


Figure 4-8: Data request message sent from the master node to the slave node. The slave node did not send a reply message.

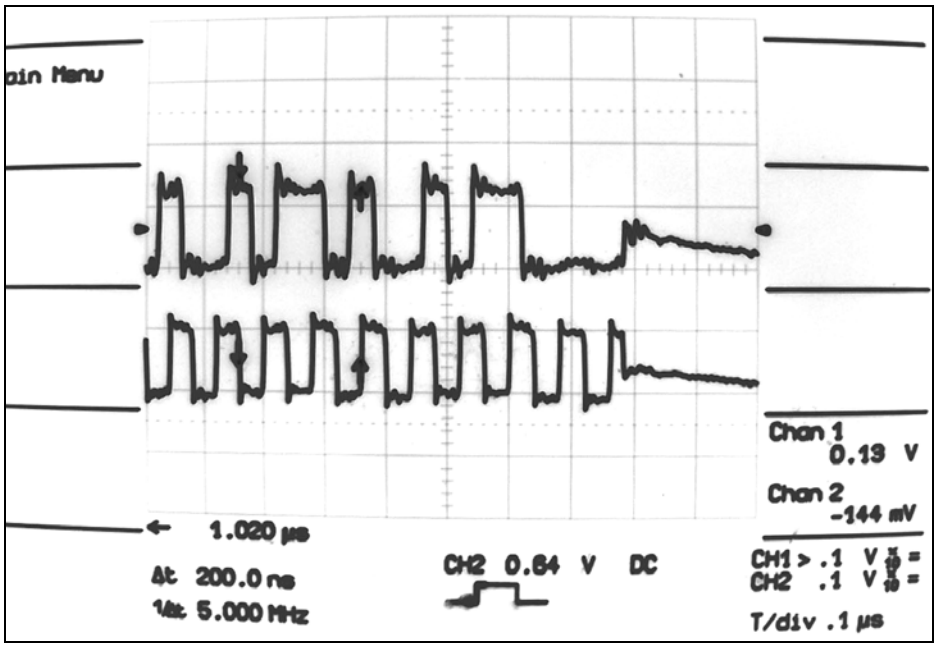


Figure 4-9: The last bits of a message sent using DDR and the bus is then released when no transmitter is driving it.

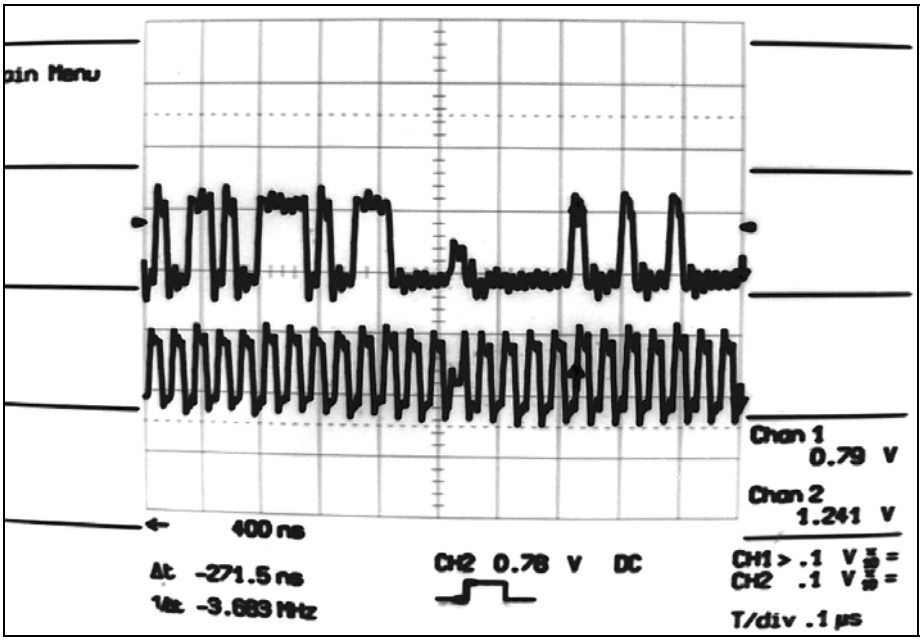


Figure 4-10: The ending of a LVDS message requesting data from a register in a slave node FPGA and the beginning of the reply message generated by the slave node containing the data in the accessed register. Notice the high impedence state in the middle when no driver is driving the bus.

4.5.3 Test using two Spartan 3 starter kit boards

4.5.3.1 The set-up

The design in this section is tested using two Spartan 3 starter kit boards as shown in Figure 4-11 and Figure 4-12. One board acts as a master node and the other one acts as a slave node. Internal pull-ups on the LVDS link are used in the master node FPGA instead of external ones. This is to stabilise the link, so that external noise does not fool the receivers to mistake the noise as data bits, when no node is driving the bus. The cable between the boards is a 2 m twisted pair CAT5 type cable. The cable is terminated in both ends.

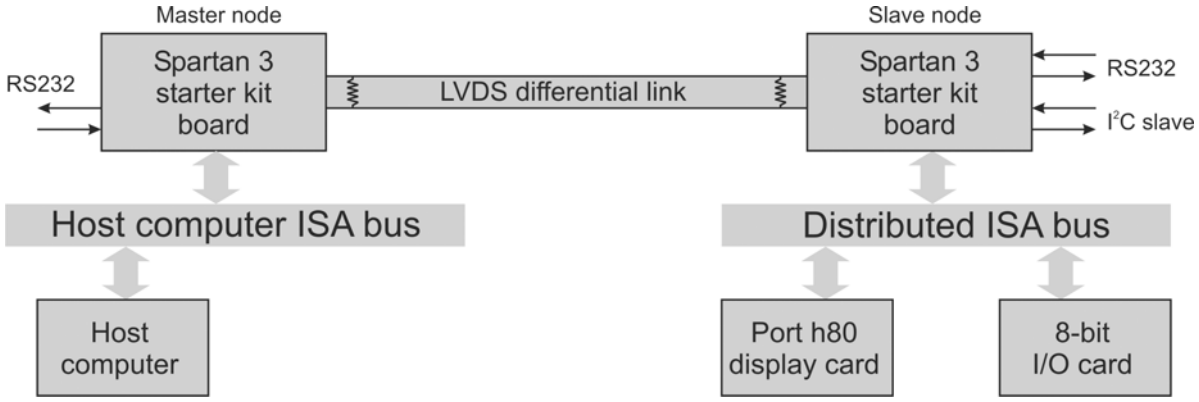


Figure 4-11: The test set-up using two Spartan 3 starter kit boards.

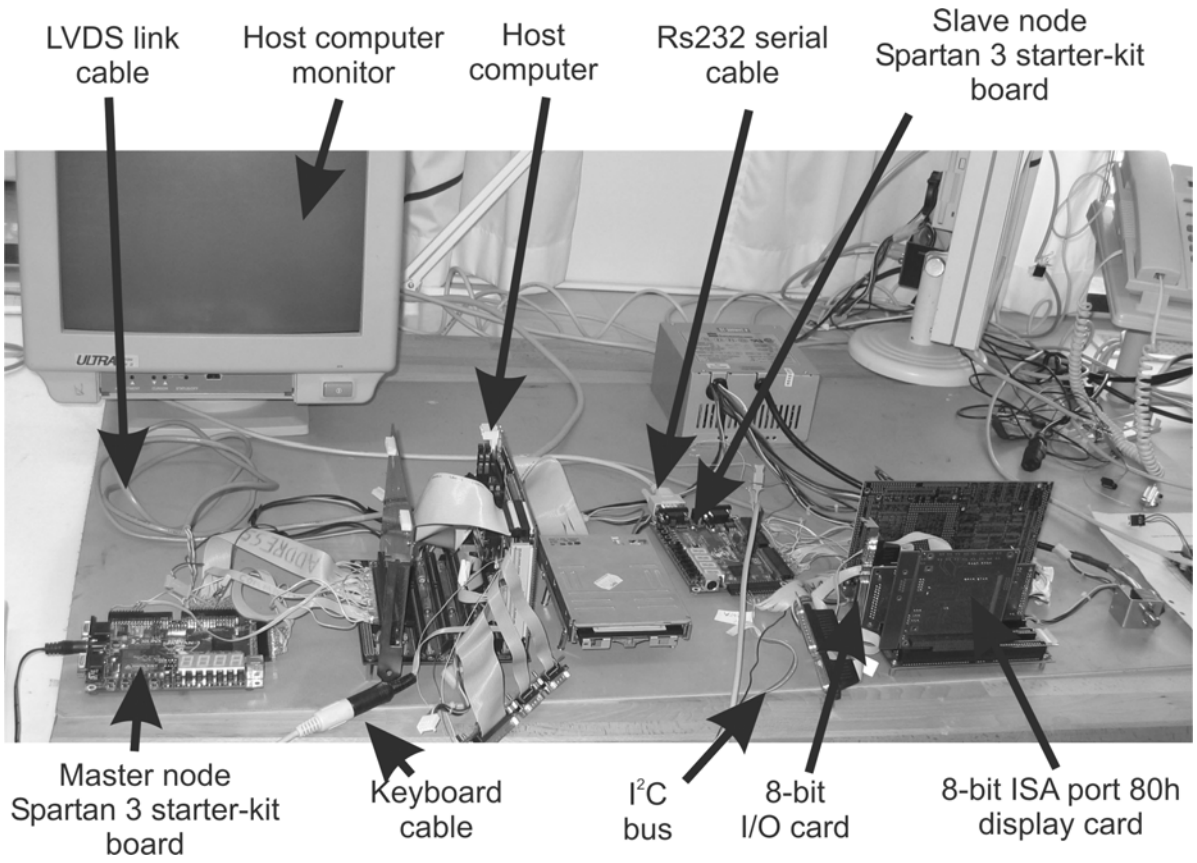


Figure 4-12: Test set-up using two Spartan 3 starter kit boards.

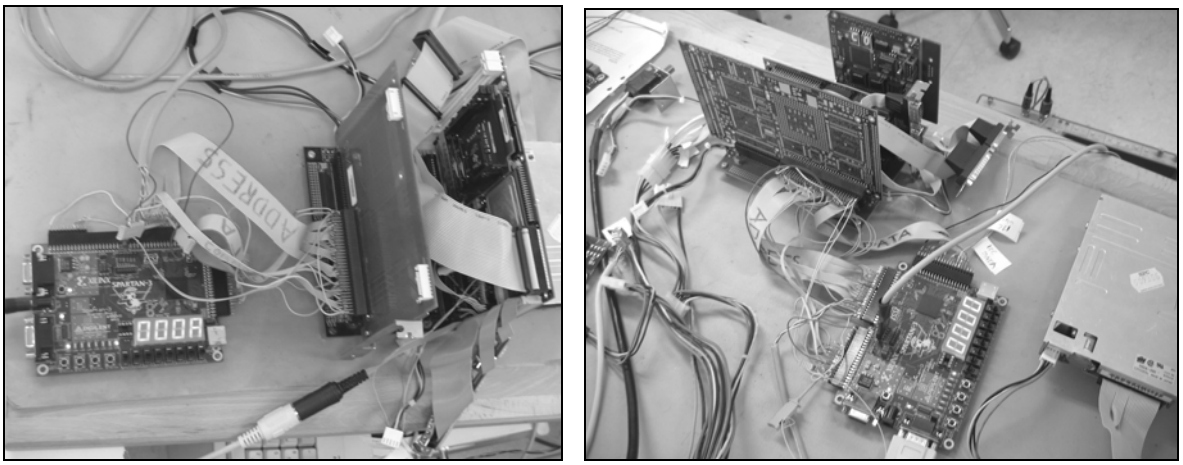


Figure 4-13: Left: A Spartan 3 starter kit board programmed as a mater node connected to the host computer via the ISA bus. Right: A Spartan 3 starter kit board programmed as a slave node connected to one 8-bit I/O card and one port 80h display card.

When the test is performed, the ISA bus is checked by the use of an oscilloscope. The LVDS transfer is controlled by the tester component in the slave and master node FPGA. This component shows the transfer results on a LED display and will turn on some specific diodes to indicate errors. The errors indicated here is if a timer has timed out or if an error has occurred in the transmission (when the error checking bits does not match). A host computer is used to read and write to registers on the distributed data bus. The port 80 display card is displaying the value written to this address and the 8-bit I/O card holds some registers that can be read from and written to.

4.5.3.2 Test results

This set-up has been tested at different transfer rates at different LVDS bus lengths. When the LVDS link is stable, all the design as described in chapter 3 works perfectly. This is the result of serious testing using all the features of the network, including the I^2C, ISA and RS232 interfaces, interrupts and internal register accesses.
The maximum transfer rate achieved in this set-up was 200 Mbaud on the LVDS link. The lengths of the LVDS bus tested were up to 2 m. It also worked at lower data rates. It was hard to get the termination to work properly because the resistors couldn't be put close enough to the FPGA driver pins, due to the board layout. When no termination was used the transmission worked best.
The resulting ISA bus transfer rate is about 1600 ns for a 16-bit access and roughly 1900 ns for an 8-bit access. The resulting bit rate is then 10 Mbps and 4 Mbps respectively. If an internal register is accessed in the slave node the access time is about 1200 ns. If two registers are accessed at a time 16 bits may be written every 1200 ns. This will result in a transfer rate of 13 Mbps. This should be compared to the ordinary ISA bus access in a PC, where the ordinary 16-bit access has a transfer rate of 43 Mbps and the ordinary 8-bit access has a transfer rate of 10.7 Mbps. More information about the transfer rate in the ISA bus and the distributed ISA bus is found in the sections 2.4 and 3.8 respectively.

4.5.4 Test using one Hectronic H4070 board

4.5.4.1 The set-up

This set-up, as shown in Figure 4-14, is very similar to the set-up using the two starter kit boards above. This set-up uses one H4070 board as shown in Figure 4-15. The difference is that mainly the LVDS link was tested, while no interfaces except the ISA slave in the master node were. The registers in the slave node were written to and read from. During these accesses, test logic in the FPGA checked if the packages was received correctly on the LVDS bus. One LED-diode was lit when the message was received with no errors and another one was lit when the error checking code did not match. Different base addresses could be chosen for the slave node by the use of a 4-bit hexadecimal switch. The slave node only replied when the accessed address matched with the address set by the hexadecimal switch.
The big difference from the set-up using the starter kit boards above is the configuration of the LVDS bus. The bus is configured as multipoint and not as point-to-point.

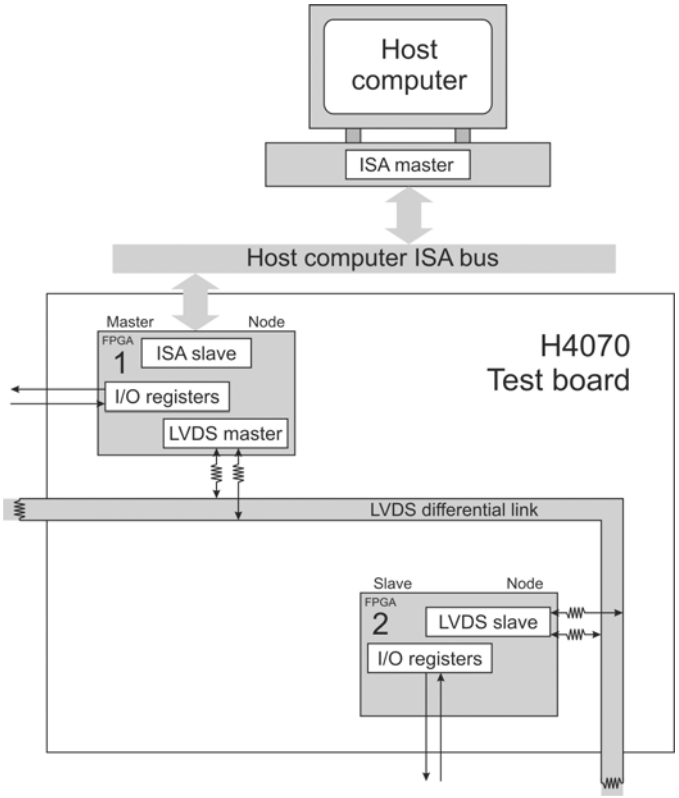


Figure 4-14: Set-up using one Hectronic H4070 board connected to the host computer via the ISA bus.

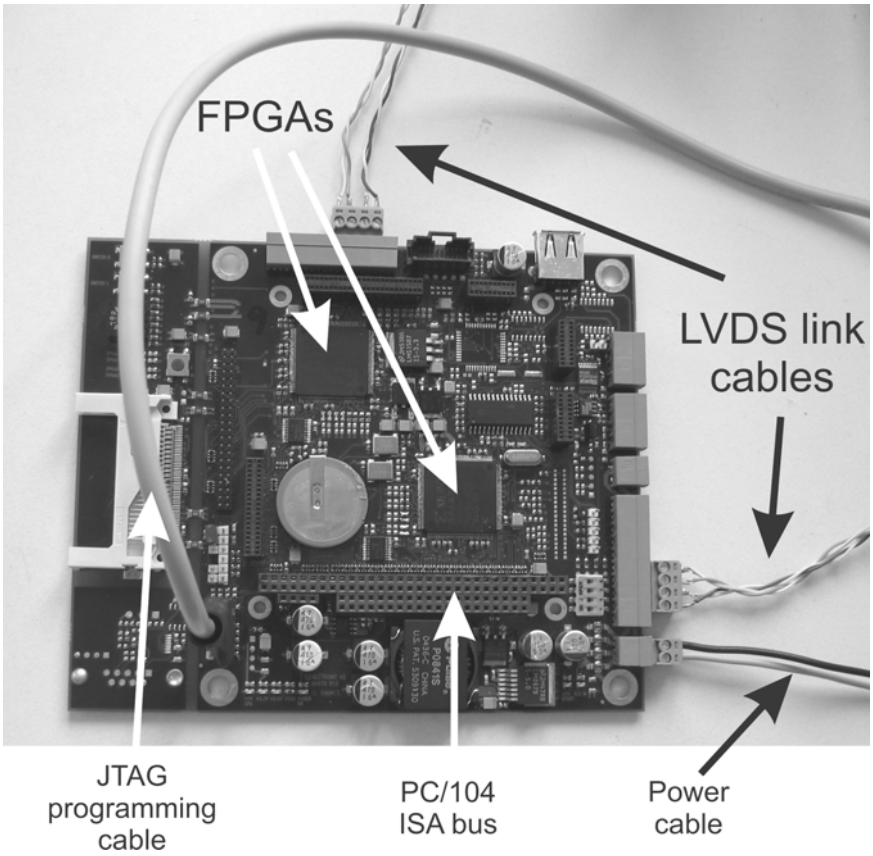


Figure 4-15: The Hectronic H4070 test board.

4.5.4.2 Test results

Here it turned out to be quite hard to terminate the LVDS bus correctly. When this was done a throughput of 400 Mbps was achieved at most. The network worked best with serial termination resistors of 40 Ω and parallel termination resistors of 100 Ω.
Because of the difficulty to achieve a stable LVDS link, this arrangement should not be used when only two FPGAs are communicating over an LVDS link. A point-to-point configuration is a much better approach.

4.5.5 Test using two Hectronic H4070 boards

4.5.5.1 The set-up

The only difference of this set-up and the set-up above is that two Hectronic H4070 cards are used and all the nodes are connected to the same LVDS bus. The 4070 board is shown in Figure 4-15 and the set-up is shown in Figure 4-16 and has one master node and three slave nodes.

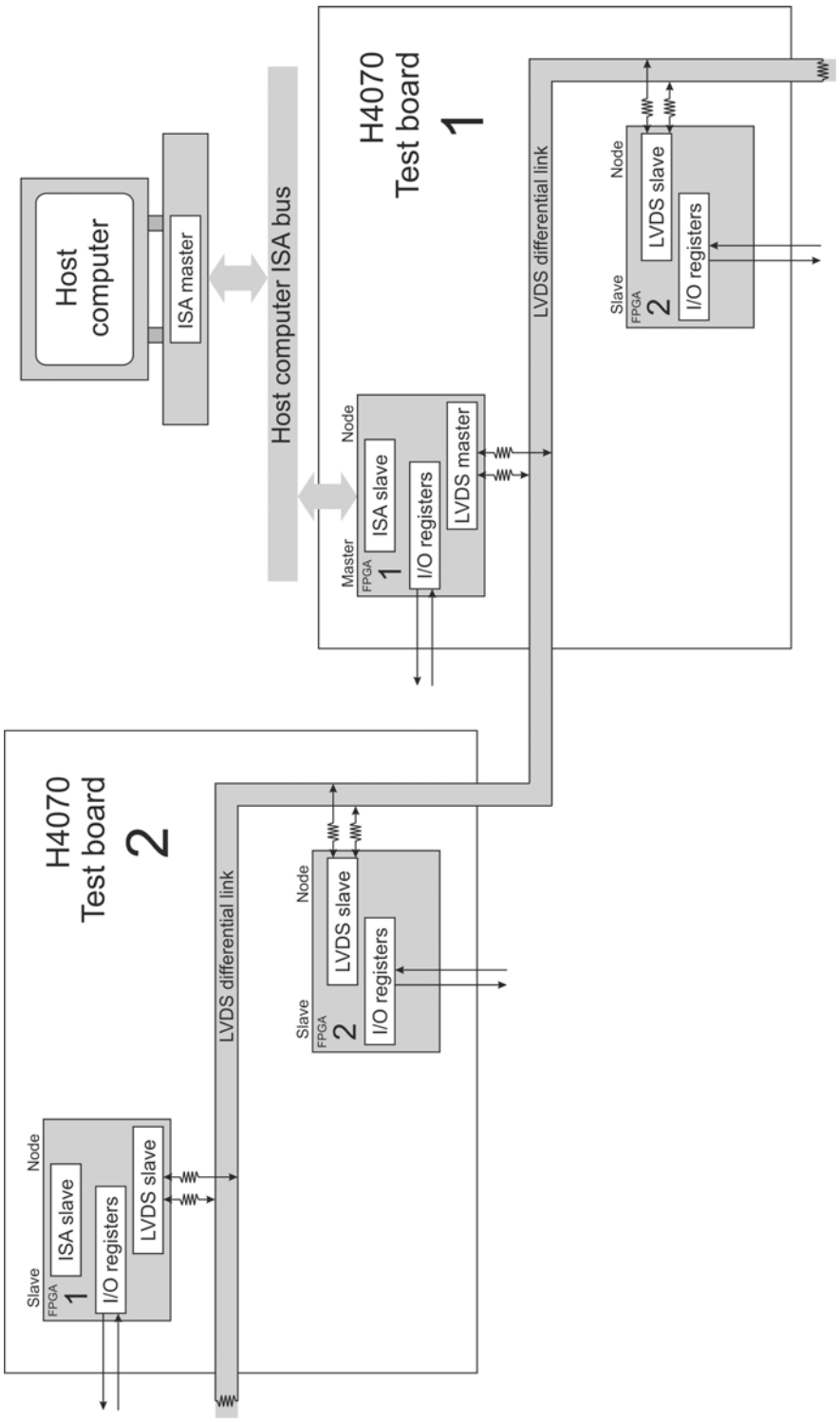


Figure 4-16: Set-up using two Hectronic H4070 boards connected to the host computer via the ISA bus.

4.5.5.2 Test results

This configuration was never managed to work. A wide range of termination resistors were used but messages were only received correctly at very rare occasions. The reflections and the LVDS bus instability also destroyed several FPGAs that had to be changed. The reason for this is that a multipoint network is very hard to configure [24].

Chapter 5 – Conclusion

5.1 Conclusion

The distributed ISA bus network designed in this project is forming a good and flexible base for further development. The program code is carefully built in a module like manner, which makes the programmed modules easy to recombine to suit other applications.
Only four wires are used to carry the information between the network nodes and the transfer speed is at least 200 Mbaud point-to-point using the LVDS standard. This results in an access time for the distributed ISA bus network only three to four times longer than for the ordinary ISA bus. When registers in a slave node are accessed the transfer rate is up to 12 Mbps.
The protocols supported by the network are the I^2C, ISA and RS232 standards. The remaining interfaces needed for a complete test system are not designed in this project and have to be designed in the future.
The only problem that has to be investigated further is to stabilise the multipoint LVDS link. Experience shows that the LVDS drivers in the FPGA are very sensitive to unmatched networks and that a multipoint network is very hard to configure [24]. To solve this problem the network design could easily be reconfigured into a star network using only point-to-point LVDS links. Because a point-to-point configuration is simple to stabilise, using only two terminating resistors, this could be a good solution. Only if the network has small dimensions, the number of transceivers is limited and only short stubs are implemented, the multipoint network could work.
All the basic requirements of the distributed network, except for the physical LVDS link if a multipoint network is considered, have been met so far. If a suitable solution for the physical LVDS link could be found, the distributed network will have great opportunities to be further developed into a complete test system.

5.2 The next step

The next step is to find a stable solution for the LVDS link and to develop more interface modules. Because the system is easy to expand, only modules handling these interfaces has to be constructed. When a new module is added it easily communicates to the distributed network via internal registers in the FPGA. In the main design, only the internal register manager module and the address space has to be slightly modified.
When the complete test system is working, the code should be optimised. This includes adjustments of internal communication timers, compacting of the transmitted data blocks transmitted by the LVDS links and optimisation of the clock frequencies used by the logic in the FPGAs.
The most important step to take next is to develop a stable physical LVDS link. When different network configurations are evaluated, external drivers to protect the sensitive FPGAs are needed. These could possibly also improve the signal integrity. Below are some guideline steps recommended by me to improve the LVDS link.

- **Consider the network configuration:** Because a multipoint network is hard to configure, another network configuration could be chosen. A star network configuration that only uses point-to-point transmissions is a much easier approach. If there are no special needs for a multipoint configuration, a star network should be considered.

- **Use external drivers:** Because several FPGAs have been destroyed during the network configuration testing, external drivers should be chosen. These are much cheaper and are easy to change for new ones. External drivers are also less sensitive than the on-chip FPGA drivers because of thicker silicon. The driver can also be placed closer to the LVDS bus, shortening the stub length. It could be possible that an external driver has lower impedance and produces a better LVDS signal, as well.
- **Build a new test board:** Several small test boards should be built especially to test the multipoint configuration. Putting these boards together, an optimal network could be configured and the true physical limitations of the network could be found.

Chapter 6 – Discussion

Below different development strategies are discussed. These strategies points out several development directions for the present system. These directions have great chances of making this system a complete, highly flexible and robust test system.

6.1 The future test system design

The systems shown in the Figure 6-1 and Figure 6-2 gives an idea of how the final test system could look like. The test object will have contact, via all its ports and buses, to the distributed network. The test object will be able to receive test programs and test instructions from the host computer via for example the ISA bus or the RS232 serial port. Communication tests can be made via the test objects interfaces and the test result is reported back to the host computer. Because many slave nodes can be connected to the master node at the same time, several test objects may be tested at the same time as well. Different test configurations can be chosen and two examples of such configurations are explained below.

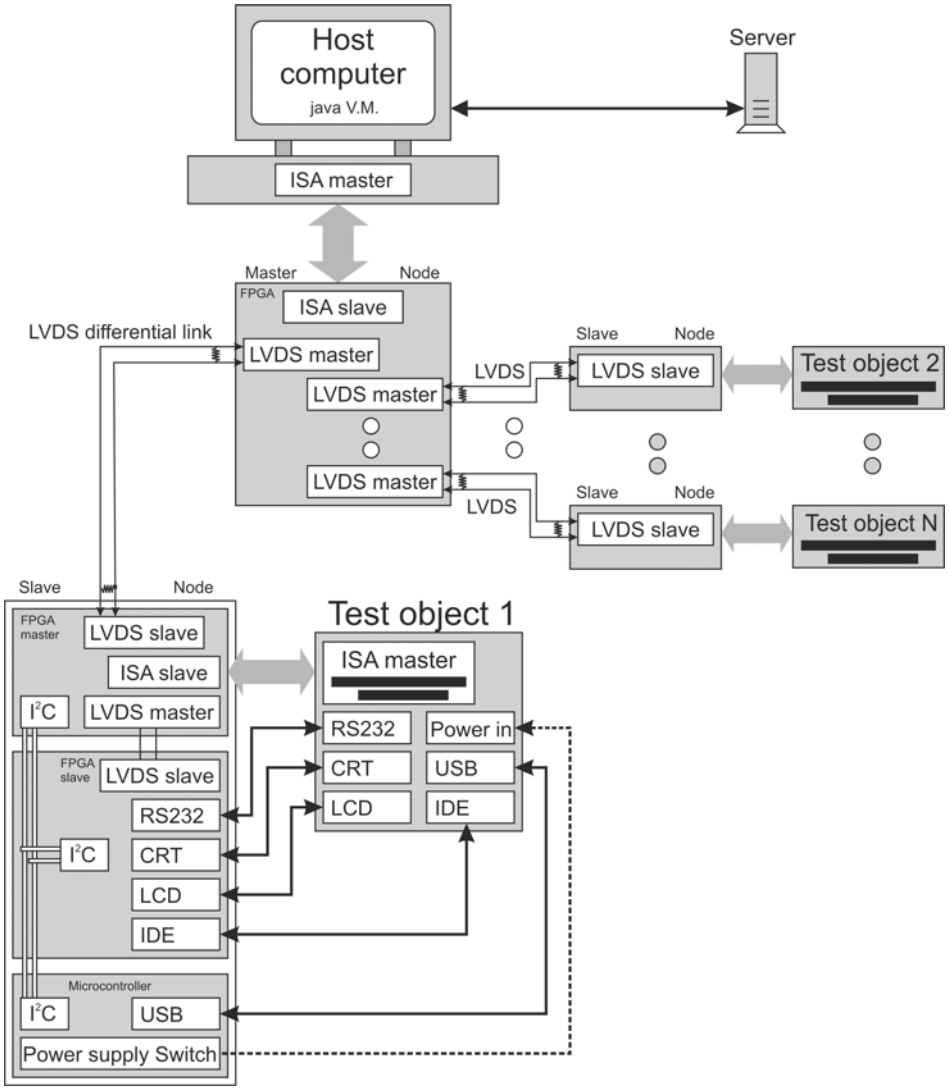


Figure 6-1: A possible test configuration. All of the interfaces on the test objects are connected to one slave node respectively.

6.1.1 Two examples of test configurations

In this section two possible test procedures are explained. This is to give an idea and a base for a discussion of how the test should be performed. In test procedure 1, the test object tests itself by communicating to its own interfaces via the slave node. In test procedure 2, the host computer manages the testing of all the interfaces of the test object, one by one. The test starts in the same way for both procedures. The start of the test procedure is explained below:

1. First, the test object powers up and the port 80 codes are sent via the serial port and are collected in a FIFO in the slave node.
2. The host computer polls all the slave nodes in the distributed network and stores a list of available slave nodes. The port 80 codes in the specific slave nodes are also uploaded to the host computer.
3. The host computer sends the test operating system code to a FIFO in the slave node. This test operating system is to be downloaded and run by the test object. It will then perform communication tests of the test objects peripherals. Test status registers are also set in the slave node to inform the test object that it can start loading the test operating system.
4. The test object reads a test status register from a specific address on the ISA bus. If the test operating system is available in the FIFO the test object starts to download the code. If no message is found on the ISA bus, the test object signals that on the serial port. If the test operating system code was ready but the ISA bus did not work, the signal on the serial port will inform the test system about the error. The test operating system could then, as a second option, be loaded via the serial port.
5. When the test object has completed the download, the test operating system will be run.

At this stage, when the test operating system is running on the test object, test procedures may commence. Two examples of test procedures are given below.

6.1.1.1 Test procedure 1

The main feature in this example is that the test object tests its own interfaces. The test object communicates through its ISA bus to the slave node and can in this way communicate to itself through the other interfaces on the slave node. When the test is finished, the test object reports the result back to the host computer.

The advantage of this configuration is that it is fast and simple. This is because the test object tests itself and only uses the logic and interfaces corresponding to its slave node. No transfers between the nodes in the network are needed during the test phase. The disadvantage could be that it is harder to modify the test. Not only the instructions on the host computer has to be changed. The test operating system has to be modified as well. Interfaces from several slave nodes can not be connected to the same test object. This could be a disadvantage if the test object contains more interfaces than supported by one slave node.

6.1.1.2 Test procedure 2

In test procedure 2, the host computer is mastering the test totally. The test operating system is running on the test object and checks a few test status registers on the slave node, via its ISA bus, for test instructions. The host computer then writes test instructions to the test status registers on the slave node, instructing the test object to perform specific tasks. For example, a task could be to read or write data to a specific port. If the task was to write, the host computer then will read from the specific interface on the slave node, connected to that port, to check if the interface is working. In this way, all interfaces of the test object can be tested.

The advantage of this configuration is that the host computer has complete control of the test. Also, if only one port is to be tested, this is easily configured in the host computer. Another advantage is that the interfaces of the test object may be connected to several slave nodes as shown in Figure 6-2. This is because the host computer reaches all the nodes in the test system, in contrast to the test object that only reaches the registers in the slave node connected to its ISA bus.

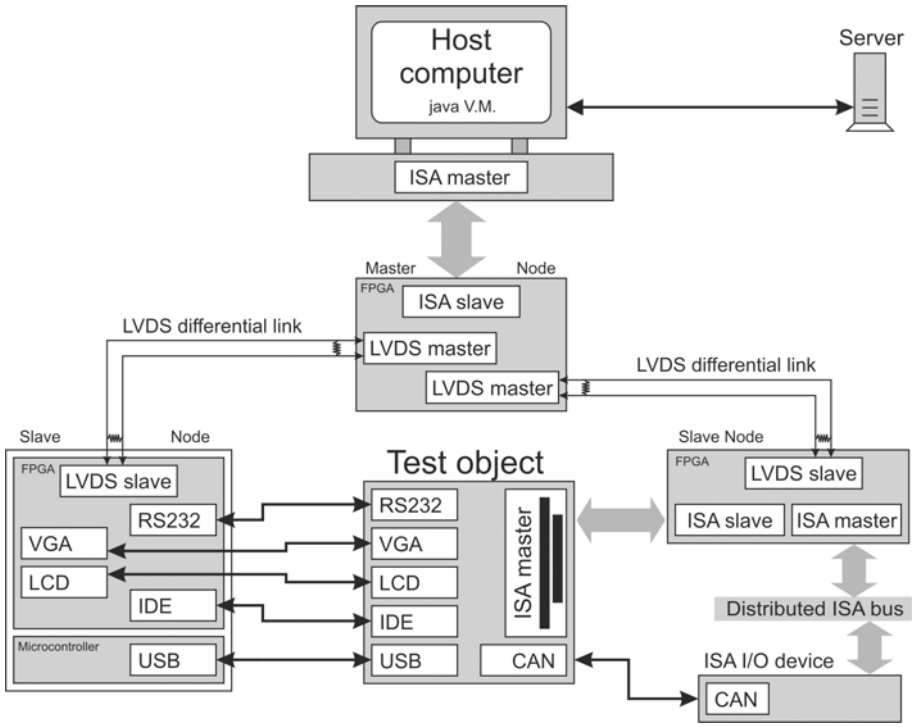


Figure 6-2: A possible test configuration. The interfaces on the test object are connected to two slave nodes.

6.1.2 Implementation of a center node

If the host computer cannot be close to the slave nodes, an extra center node can be implemented as shown in Figure 6-3. This configuration is quite easy to implement but requires one more node type, the center node.

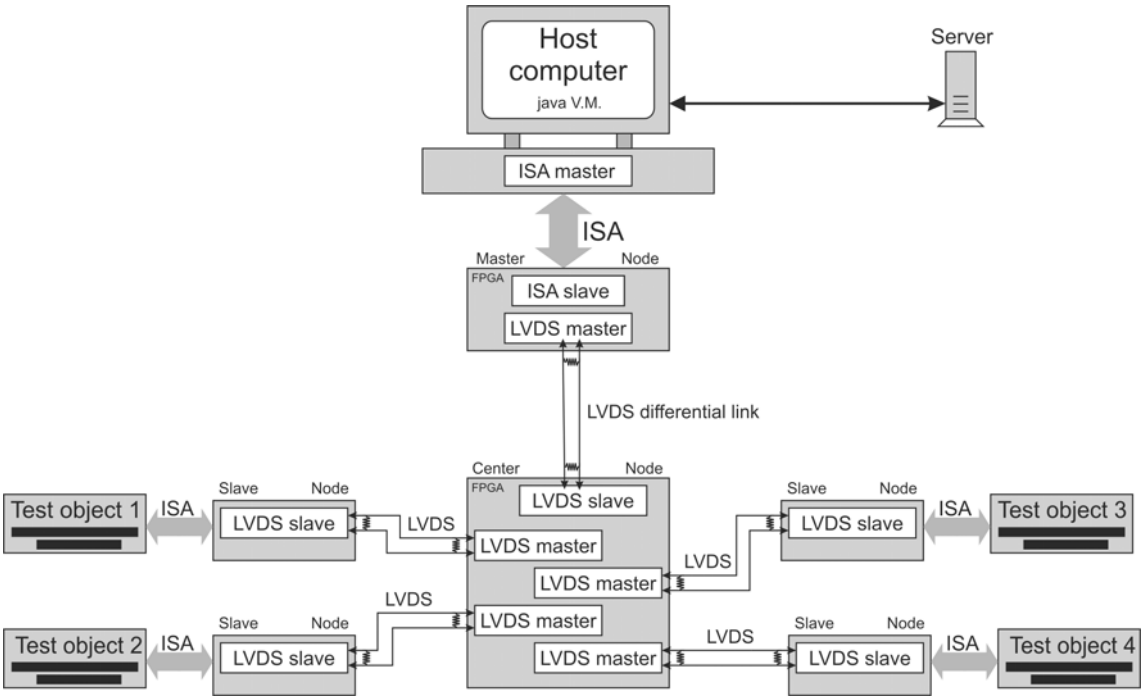


Figure 6-3: The distributed ISA bus network in a star configuration with an extra LVDS link from the master node to the center node.

6.1.3 A bigger FPGA instead of two smaller ones

The Hectronic H4070 board has two Spartan 3 XCS200 FPGAs in TQ144 package. These FPGAs can be soldered by hand and are good for development purpose. In the final system there would probably be more cost effective to use only one FPGA, but choosing a bigger one.
Because the slave node or the master node uses only one quarter of the internal logic in the XCS200 FPGA, they will probably be big enough even for the final test system. The most cost effective way is to choose a package with enough pin counts. The XCS200 FPGA supports up to 173 user pins. If this is not enough the next larger XCS400 FPGA supports up to 264 user pins and even more I/O pins can be achieved by choosing even bigger FPGAs.

6.1.4 Steps needed to develop a complete test system

This section explains what needs to be developed to complete the test system. The main part of the distributed test system has been developed in this project. New modules to manage all the interfaces, that need to be tested, have to be developed. The main parts of the distributed system have not to be changed much. The register manager has only to be expanded into more registers and the address space used by the nodes is easily modified. To optimise the upload to the test object a FIFO could be designed. The possible future development tasks are explained below.

6.1.4.1 Implementing a FIFO speeding up the program code upload to the test object

The program code has to be uploaded to the test object. This could be done via several ports or buses, but the ISA bus might be the most suitable one. A memory mapped ISA interface can be used to address all the program code to be uploaded. Because the FPGAs BRAM memory is limited, it can not store all the program code at once. To solve this problem only parts of the code can be uploaded and stored in the FPGA at a time. A FIFO device is a good approach to buffer the data. The FIFO could store big blocks of data and one block could be read from the test object via the ISA bus, while another data block could be uploaded from the host computer at the same time. The status of the FIFO and the base address of the data block could be read from registers and the test object could load one data block at a time. This will not need a big address space and the ISA modules supporting only I/O device accesses might be enough. The Figure 6-4 shows the implementation of the FIFO.

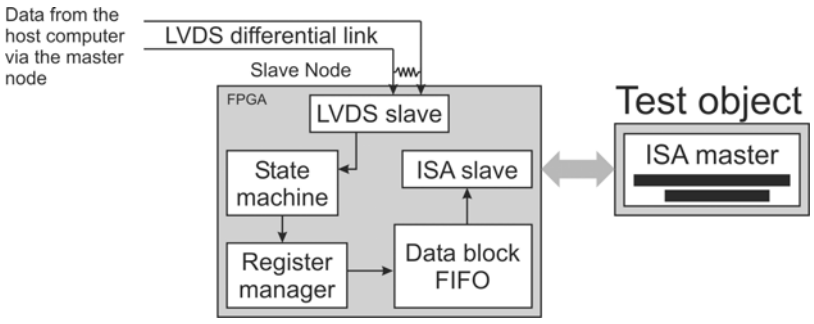


Figure 6-4: Possible implementation of the program upload to the test object using a data block FIFO.

6.1.4.2 Developing new modules in the FPGA

In this project, only a few interfaces have been made. To perform a complete test of the test object, new modules has to be constructed so that all the interfaces on the test object can be connected. These modules are easily implemented in FPGAs as they only connect via the register manager. When a new module is connected, more registers in the register manager have to be added and the address space has to be modified to include the new registers. The RS232 and the I^2C modules are examples of interface modules that have only been connected via the register manager.

6.1.4.3 Memory mapped ISA

The ISA modules are in this design I/O-device mapped. The address space available is here very limited. If more address space is needed, a good approach is to change the ISA modules to be memory mapped. In the memory-mapped configuration, the address space is broader and the ISA bus access cycle looks slightly different. This has to be modified in the new design but is not very time consuming. A change to memory mapped ISA could therefore be a good approach in the final design.

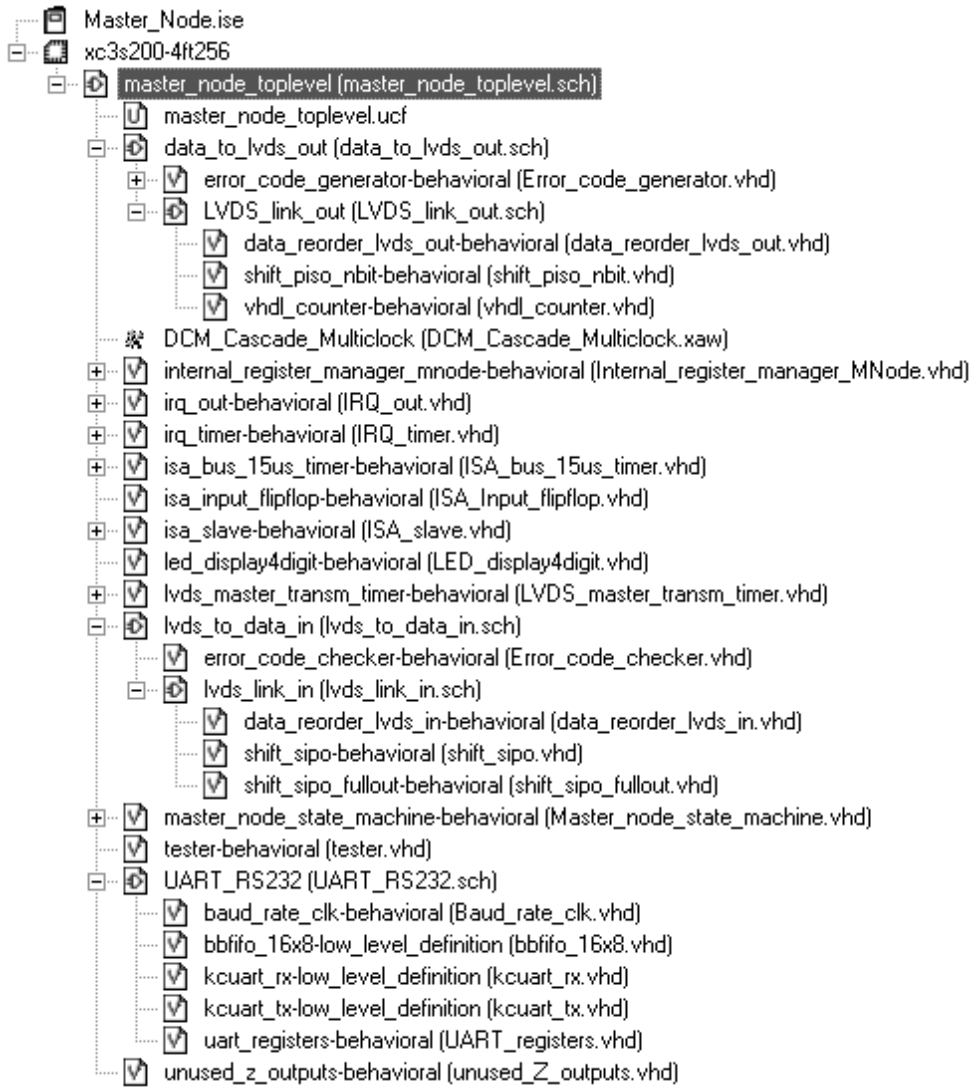
6.1.4.4 Optimising

The purpose of the design in this project is to try different configurations and to form a good and flexible base for further developments. From this design a complete test system can be built. Some of the code is made very flexible so that it will be easy to change it in case different ideas come up. These parts of the code have therefore no need to be optimised in this development stage. The optimisation of the communication has to be made at a later stage when a more complete design is tested properly. Some of the major areas where the design could be optimised are explained in the list below.

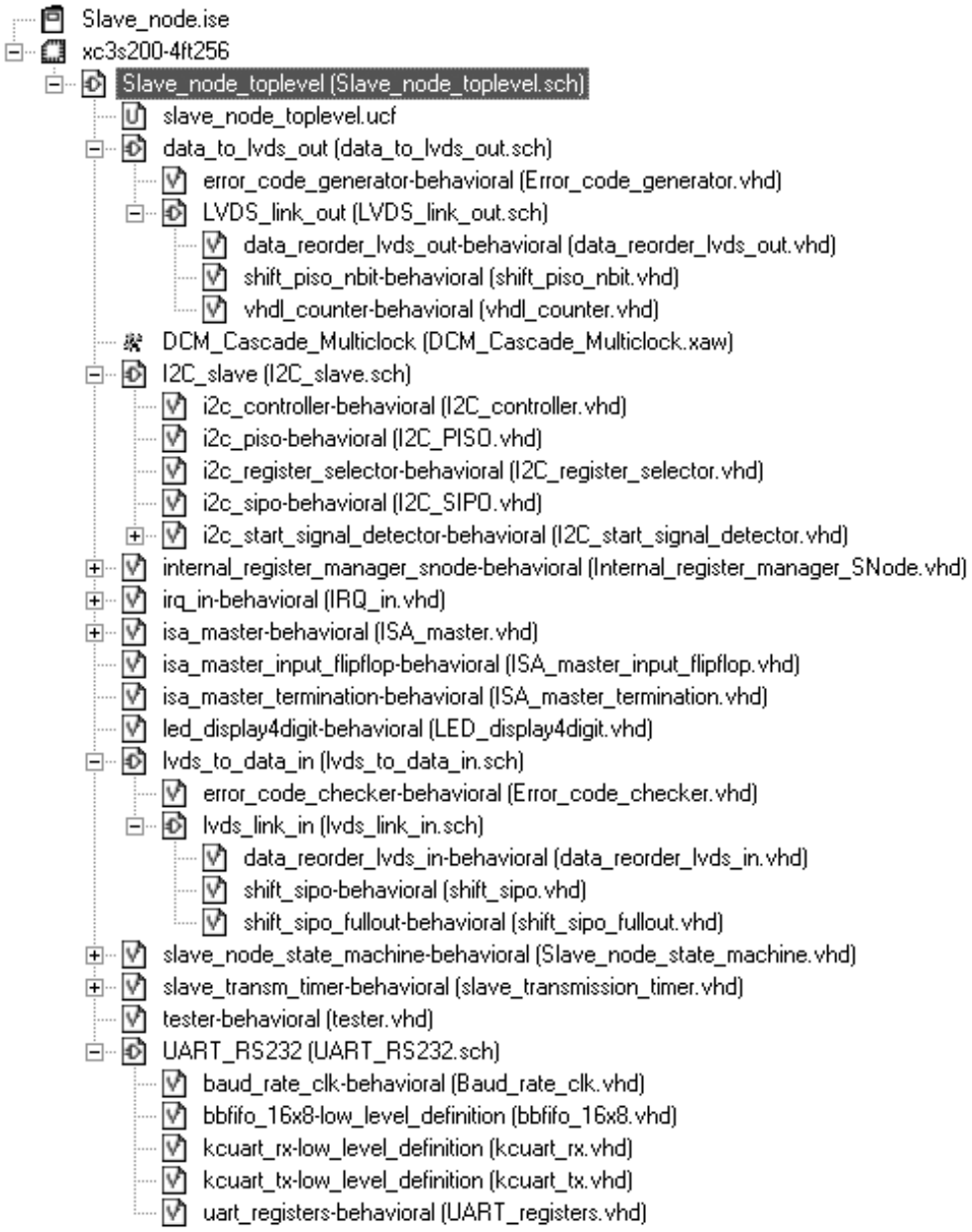
- **LVDS data block:** The data block transferred between the master node and the slave node is the same as the data block transferred back. The Figure 3-7 on page 36 shows the general-purpose data block where many of the bits transferred are never used. For example, 3 info-bits are never used and the 8 error checking bits are taking up much space. When data is transferred back from the slave node to the master node the address bits are never used. To improve the LVDS link cycle time these bits could be removed or decreased.
- **LVDS transmission speed:** A 50 MHz clock is used to generate the data and clock signals in the LVDS out module. In this design, 2 DCMs are cascaded to generate the higher clock frequencies of 100 to 150 MHz. If an input clock of a higher data rate is used, only one DCM is needed and higher quality will be achieved in the clock and data signals. If a higher data-rate and a better communication channel could be used the LVDS transmission cycle would be shortened.
- **Internal timers in the FPGA:** The timers used in the FPGA are controlling the data flow. Adjusting the timers can shorten the LVDS link cycle. If the LVDS master transmission timer and the LVDS slave transmission timer are adjusted to shorten the acceptable response time from the slave node, the bus cycle will be shorter when the addressed node does not exist.
- **Clock frequencies in the FPGA:** The FPGA has several internal clock domains. This is because some code is not able to run as fast as some code in other parts of the FPGA. These frequencies can be optimised to run at the highest allowed speed in all parts of the FPGA respectively.

Appendix A – Design hierarchy

A.1 Master node code hierarchy



A.2 Slave node code hierarchy



Appendix B – Source code

B.1 Error_code_generatior.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node and Slave Node
-- Module Name:    Error_code_generator - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Generates the error checking code in the
--                 data block transmitted by lvds between
--                 modules.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Error_code_generator is
--This module has a 8 ns delay => 125 Mhz max
    Port ( a_out : out std_logic_vector(39 downto 0);
           Data_in_1 : in std_logic_vector(15 downto 0);
           Address_in_1 : in std_logic_vector(9 downto 0);
           Read1_write0_1 : in std_logic;
           SBHE_1 : in std_logic;
           Infobits_1 : in std_logic_vector(3 downto 0));

end Error_code_generator;

architecture Behavioral of Error_code_generator is
signal a : std_logic_vector(39 downto 0);
signal ERR_1 : std_logic_vector(7 downto 0);
signal ERR_2 : std_logic_vector(7 downto 0);
signal ERR_code : std_logic_vector(7 downto 0);

signal Data_in : std_logic_vector(15 downto 0);
signal Address_in : std_logic_vector(9 downto 0);
signal Read1_write0 : std_logic;
signal SBHE : std_logic;
signal Infobits : std_logic_vector(3 downto 0);
begin

  a_out <= a;     -- The output data block including both data
                  -- and error check bits.
  a(15 downto 0) <= Data_in;
  a(25 downto 16) <= address_in;
  a(29 downto 26)<= Infobits;
  a(30) <= SBHE;
  a(31) <= Read1_write0;
  a(39 downto 32) <= ERR_code;

  Data_in <= Data_in_1;
  Address_in <= Address_in_1;
  Read1_write0 <= Read1_write0_1;
  SBHE <= SBHE_1;
  Infobits <= Infobits_1;

  --The following code generates the error check code in two
  --steps using xor gates.

  ERR_1(0) <= a(0) xor a(16);
  ERR_1(1) <= a(1) xor a(17);
  ERR_1(2) <= a(2) xor a(18);
  ERR_1(3) <= a(3) xor a(19);
  ERR_1(4) <= a(4) xor a(20);
  ERR_1(5) <= a(5) xor a(21);
  ERR_1(6) <= a(6) xor a(22);
  ERR_1(7) <= a(7) xor a(23);

  ERR_2(0) <= a(8) xor a(24);
  ERR_2(1) <= a(9) xor a(25);
  ERR_2(2) <= a(10) xor a(26);
  ERR_2(3) <= a(11) xor a(27);
  ERR_2(4) <= a(12) xor a(28);
  ERR_2(5) <= a(13) xor a(29);
  ERR_2(6) <= a(14) xor a(30);
  ERR_2(7) <= a(15) xor a(31);

  ERR_code(0) <= ERR_1(0) xor ERR_2(0);
  ERR_code(1) <= ERR_1(1) xor ERR_2(1);
  ERR_code(2) <= ERR_1(2) xor ERR_2(2);
  ERR_code(3) <= ERR_1(3) xor ERR_2(3);
  ERR_code(4) <= ERR_1(4) xor ERR_2(4);
  ERR_code(5) <= ERR_1(5) xor ERR_2(5);
  ERR_code(6) <= ERR_1(6) xor ERR_2(6);
  ERR_code(7) <= ERR_1(7) xor ERR_2(7);

end Behavioral;
```

B.2 Data_reorder_LVDS_out.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node and Slave Node
-- Module Name:    data_reorder_lvds_out - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Reorders data wires between buses
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity data_reorder_lvds_out is
```

```
data_out_first(6) <= data_in(12);
data_out_first(7) <= data_in(14);
data_out_first(8) <= data_in(16);
data_out_first(9) <= data_in(18);
data_out_first(10) <= data_in(20);
data_out_first(11) <= data_in(22);
data_out_first(12) <= data_in(24);
data_out_first(13) <= data_in(26);
data_out_first(14) <= data_in(28);
data_out_first(15) <= data_in(30);
data_out_first(16) <= data_in(32);
data_out_first(17) <= data_in(34);
data_out_first(18) <= data_in(36);
data_out_first(19) <= data_in(38);
data_out_first(20) <= '1';

data_out_second(0) <= '0';
data_out_second(1) <= data_in(1);
data_out_second(2) <= data_in(3);
data_out_second(3) <= data_in(5);
```

```
Port ( data_out_first : out std_logic_vector(20 downto 0);
data_out_second : out std_logic_vector(20 downto 0);
data_in : in std_logic_vector(39 downto 0));

end data_reorder_lvds_out;

architecture Behavioral of data_reorder_lvds_out is
begin

-- Data from input is split up into two data buses connected to
-- two shift registers. Data from the shift registers is then
-- sent by the use of DDR.

data_out_first(0) <= data_in(0);
data_out_first(1) <= data_in(2);
data_out_first(2) <= data_in(4);
data_out_first(3) <= data_in(6);
data_out_first(4) <= data_in(8);
data_out_first(5) <= data_in(10);
data_out_second(4) <= data_in(7);
data_out_second(5) <= data_in(9);
data_out_second(6) <= data_in(11);
data_out_second(7) <= data_in(13);
data_out_second(8) <= data_in(15);
data_out_second(9) <= data_in(17);
data_out_second(10) <= data_in(19);
data_out_second(11) <= data_in(21);
data_out_second(12) <= data_in(23);
data_out_second(13) <= data_in(25);
data_out_second(14) <= data_in(27);
data_out_second(15) <= data_in(29);
data_out_second(16) <= data_in(31);
data_out_second(17) <= data_in(33);
data_out_second(18) <= data_in(35);
data_out_second(19) <= data_in(37);
data_out_second(20) <= data_in(39);

end Behavioral;
```

B.3 Shift_PISO_nbit.vhd

```
----------------------------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    shift_piso_nbit - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Shift register parallel in serial out
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shift_piso_nbit is
    generic (reg_width: integer:= 21);     --reg_width = number of bits on the input
    Port ( clk, reset, shift_en, load : in std_logic;
           d_in : in std_logic_vector(reg_width-1 downto 0);
           shift_out : out std_logic);
end shift_piso_nbit;

architecture Behavioral of shift_piso_nbit is
signal shift_reg:std_logic_vector(reg_width-1 downto 0);
signal loaded:std_logic:='0';
begin
  process(clk,reset)
  begin
    if reset='1' then
      shift_reg <= (others => '0');
      loaded <= '0';
    elsif clk'event and clk='1' then
      if load='0' then
        loaded<='0';
      end if;
      if load='1' and loaded='0' then     -- waits for the load signal to go low again before
                                          -- reloading the shift register
        loaded<='1';
        shift_reg <= d_in;
      elsif shift_en='1' then
        shift_reg(reg_width-1 downto 1) <=   shift_reg(reg_width-2 downto 0);   --Shifts the register
        shift_reg(0) <= '0';
      end if;
    end if;
  end process;
  shift_out <= shift_reg(shift_reg'high);  --shift_reg(highestbit) => outputs the highest bit
end Behavioral;
```

B.4 VHDL_counter.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    vhdl_counter - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Keeps track of the data shifted out through
--                 the shiftregisters. When count_full = '1'
--                 all data is shifted out.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity vhdl_counter is
    Port ( clk_to_count, reset : in std_logic;
           start_count : in std_logic;
           count_full : out std_logic);
end vhdl_counter;
```

```
architecture Behavioral of vhdl_counter is
signal a : std_logic_vector(4 downto 0):= "00000";
signal count_is_started, do_count : std_logic;
begin
p0:process(clk_to_count, reset)
begin
  if reset = '1' then
    a <= "00000";
    count_full <= '1';
    count_is_started <= '0';
    do_count <= '0';
  elsif rising_edge(clk_to_count) then
    if start_count = '0' then
      count_is_started <= '0';
    elsif count_is_started = '0' then
-- The signal start_count has to go low again before restart of
-- timer
      count_is_started <= '1';
      do_count <= '1';
    end if;
    if a < "10110" and do_count = '1' then
      a <=  a + 1;
      count_full <= '0';
    elsif do_count = '1' then
      a <= "00000";
      count_full <= '1';
      do_count <= '0';
    end if;
  end if;
end process;
end Behavioral;
```

B.5 Internal_register_manager_MNode.vhd

```
----------------------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    Internal_register_manager_MNode - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Handles the registers in the Masternode
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Internal_register_manager_MNode is
    Port ( clk, reset : in std_logic;
           Data_in : in std_logic_vector(15 downto 0);
           Data_out : out std_logic_vector(15 downto 0);
           Address_in : in std_logic_vector(9 downto 0);
           Read : in std_logic;
           Write : in std_logic;
           SBHE : in std_logic;
           reg_transmission_ok : out std_logic;

           UART_Transmitter_holding_reg : out std_logic_vector(7 downto 0);
           UART_Transmitter_holding_reg_write : out std_logic;
           UART_Receiver_buffer_reg : in std_logic_vector(7 downto 0);
           UART_Receiver_buffer_reg_read : out std_logic;
           Master_bus_control_reg : out std_logic_vector(7 downto 0);
           UART_Divisor_low_byte : out std_logic_vector(7 downto 0);
           UART_Divisor_high_byte : out std_logic_vector(7 downto 0);
           Fifo_status_reg : in std_logic_vector(7 downto 0);
           Master_node_conf_low_byte : out std_logic_vector(7 downto 0);
           Master_node_conf_high_byte : out std_logic_vector(7 downto 0));

end Internal_register_manager_MNode;

architecture Behavioral of Internal_register_manager_MNode is
--  Generates internal signals so that the data on the outputs may be read back again
--  This should have the same effect as if 'buffer' is used instead of 'out'.
```

```
--  Because the 'buffer' declaration has sometimes been misinterpreted by the compiler
--  this method is used instead, for safety reson.

signal UART_Transmitter_holding_reg_sig : std_logic_vector(7 downto 0);  --Declaration of a register
signal UART_Transmitter_holding_reg_write_sig : std_logic;  --Signals to the UART that data has been written from register
signal UART_Receiver_buffer_reg_read_sig : std_logic;       --Signals to the UART that data has been read from register
signal Master_bus_control_reg_sig : std_logic_vector(7 downto 0);
signal UART_Divisor_low_byte_sig : std_logic_vector(7 downto 0);
signal UART_Divisor_high_byte_sig : std_logic_vector(7 downto 0);

signal reg_transmission_ok_sig : std_logic;                 --Signals that data register read or write request is done.

signal Scratch_reg_sig : std_logic_vector(7 downto 0);
signal Master_node_conf_low_byte_sig : std_logic_vector(7 downto 0);
signal Master_node_conf_high_byte_sig : std_logic_vector(7 downto 0);

begin

UART_Transmitter_holding_reg <= UART_Transmitter_holding_reg_sig;
UART_Transmitter_holding_reg_write <= UART_Transmitter_holding_reg_write_sig;
UART_Receiver_buffer_reg_read <= UART_Receiver_buffer_reg_read_sig;

reg_transmission_ok <= reg_transmission_ok_sig;

Master_bus_control_reg <= Master_bus_control_reg_sig;
UART_Divisor_low_byte <= UART_Divisor_low_byte_sig;
UART_Divisor_high_byte <= UART_Divisor_high_byte_sig;
Master_node_conf_low_byte <= Master_node_conf_low_byte_sig;
Master_node_conf_high_byte <= Master_node_conf_high_byte_sig;
Master_node_conf_low_byte <= Master_node_conf_low_byte_sig;
Master_node_conf_high_byte <= Master_node_conf_high_byte_sig;

process(clk, reset)
begin
  if reset = '1' then
    UART_Transmitter_holding_reg_sig <= (others => '0');
    UART_Transmitter_holding_reg_write_sig <= '0';
    UART_Receiver_buffer_reg_read_sig <= '0';
    Master_bus_control_reg_sig <= (others => '0');
    UART_Divisor_low_byte_sig <= (others => '0');
    UART_Divisor_high_byte_sig <= (others => '0');
    reg_transmission_ok_sig <= '0';
    Scratch_reg_sig <= (others => '0');
    Master_node_conf_low_byte_sig <= (others => '0');
    Master_node_conf_high_byte_sig <= (others => '0');

  elsif rising_edge(clk) then
    if UART_Transmitter_holding_reg_write_sig = '1' or UART_Receiver_buffer_reg_read_sig = '1' then
      UART_Transmitter_holding_reg_write_sig <= '0';
      UART_Receiver_buffer_reg_read_sig <= '0';
    elsif reg_transmission_ok_sig = '1' then
      if read = '0' and write = '0' then
        reg_transmission_ok_sig <= '0';
      end if;
    elsif read = '1' then ----------------------------Read from registers-----------------------
      reg_transmission_ok_sig <= '1';
      if SBHE = '1' then --SBHE=1 => Signal bus is not high enabled - 8bit
        case address_in is
          when "11" & X"E8" => --1000
            Data_out(7 downto 0) <= UART_Receiver_buffer_reg;
            UART_Receiver_buffer_reg_read_sig <= '1';
          when "11" & X"E9" => --1001
            Data_out(7 downto 0) <= Master_bus_control_reg_sig;
          when "11" & X"EA" => --1002
            Data_out(7 downto 0) <= UART_Divisor_low_byte_sig;
          when "11" & X"EB" => --1003
            Data_out(7 downto 0) <= UART_Divisor_high_byte_sig;
          when "11" & X"EC" => --1004
            Data_out(7 downto 0) <= Fifo_status_reg;
          when "11" & X"ED" => --1005
            Data_out(7 downto 0) <= Scratch_reg_sig;
          when "11" & X"EE" => --1006
            Data_out(7 downto 0) <= Master_node_conf_low_byte_sig;
          when "11" & X"EF" => --1007
            Data_out(7 downto 0) <= Master_node_conf_high_byte_sig;
          when others =>
            Data_out(7 downto 0) <= X"FF";
        end case;
      else    --SBHE=0 => Signal bus is high enabled - 16bit
        case address_in is
          when "11" & X"E8" => --1000
            Data_out(7 downto 0) <= UART_Receiver_buffer_reg;
            UART_Receiver_buffer_reg_read_sig <= '1';
            Data_out(15 downto 8) <= Master_bus_control_reg_sig;
          when "11" & X"E9" => --1001
            Data_out(15 downto 8) <= Master_bus_control_reg_sig;
          when "11" & X"EA" => --1002
```

```
            Data_out(7 downto 0) <= UART_Divisor_low_byte_sig;
            Data_out(15 downto 8) <= UART_Divisor_high_byte_sig;
          when "11" & X"EB" => --1003
            Data_out(15 downto 8) <= UART_Divisor_high_byte_sig;
          when "11" & X"EC" => --1004
            Data_out(7 downto 0) <= Fifo_status_reg;
            Data_out(15 downto 8) <= Scratch_reg_sig;
          when "11" & X"ED" => --1005
            Data_out(15 downto 8) <= Scratch_reg_sig;
          when "11" & X"EE" => --1006
            Data_out(7 downto 0) <= Master_node_conf_low_byte_sig;
            Data_out(15 downto 8) <= Master_node_conf_high_byte_sig;
          when "11" & X"EF" => --1007
            Data_out(15 downto 8) <= Master_node_conf_high_byte_sig;
          when others =>
            Data_out <= X"FFFF";
        end case;
      end if;
    elsif write = '1' then ----------------------------Write to registers-------------------------
      reg_transmission_ok_sig <= '1';
      if SBHE = '1' then --SBHE=1 => Signal bus is not high enabled - 8bit
        case address_in is
          when "11" & X"E8" => --1000
            UART_Transmitter_holding_reg_sig <= Data_in(7 downto 0);
            UART_Transmitter_holding_reg_write_sig <= '1';
          when "11" & X"E9" => --1001
            Master_bus_control_reg_sig <= Data_in(7 downto 0);
          when "11" & X"EA" => --1002
            UART_Divisor_low_byte_sig <= Data_in(7 downto 0);
          when "11" & X"EB" => --1003
            UART_Divisor_high_byte_sig <= Data_in(7 downto 0);
          when "11" & X"ED" => --1005
            Scratch_reg_sig <= Data_in(7 downto 0);
          when "11" & X"EE" => --1006
            Master_node_conf_low_byte_sig <= Data_in(7 downto 0);
          when "11" & X"EF" => --1007
            Master_node_conf_high_byte_sig <= Data_in(7 downto 0);
          when others => null;
        end case;
      else    --SBHE=0 => Signal bus is high enabled - 16bit
        case address_in is
          when "11" & X"E8" => --1000
            UART_Transmitter_holding_reg_sig <= Data_in(7 downto 0);
            UART_Transmitter_holding_reg_write_sig <= '1';
            Master_bus_control_reg_sig <= Data_in(15 downto 8);
          when "11" & X"E9" => --1001
            Master_bus_control_reg_sig <= Data_in(15 downto 8);
          when "11" & X"EA" => --1002
            UART_Divisor_low_byte_sig <= Data_in(7 downto 0);
            UART_Divisor_high_byte_sig <= Data_in(15 downto 8);
          when "11" & X"EB" => --1003
            UART_Divisor_high_byte_sig <= Data_in(15 downto 8);
          when "11" & X"ED" => --1005
            Scratch_reg_sig <= Data_in(15 downto 8);
          when "11" & X"EE" => --1006
            Master_node_conf_low_byte_sig <= Data_in(7 downto 0);
            Master_node_conf_high_byte_sig <= Data_in(15 downto 8);
          when "11" & X"EF" => --1007
            Master_node_conf_high_byte_sig <= Data_in(15 downto 8);
          when others => null;
        end case;
      end if;
    end if;
  end if;
end process;
end Behavioral;
```

B.6 IRQ_out.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    IRQ_out - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Generates the interrupt signals to the host
--                 computer
-- Revision:       14
-- Revision date:  20 June 2005
```

```
        when "0111" => --------------------IRQ7----------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ7_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ7_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ7_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
```

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


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity IRQ_out is
    Port ( clk, reset: in std_logic;
      IRQ_in : in std_logic;
      IRQ_poll_ok : in std_logic;
      IRQ_next_ready_out : out std_logic;
--indicates that module is ready to get IRQ poll result from
--the IRQ specified in the 4 bit number on the IRQ_count bus
      IRQ3, IRQ4, IRQ5, IRQ6, IRQ7, IRQ9, IRQ10, IRQ11, IRQ12,
      IRQ13, IRQ14: out std_logic;
      IRQ_count : out std_logic_vector(3 downto 0));
end IRQ_out;

architecture Behavioral of IRQ_out is
signal IRQ3_sig, IRQ4_sig, IRQ5_sig, IRQ6_sig, IRQ7_sig,
IRQ9_sig, IRQ10_sig, IRQ11_sig, IRQ12_sig, IRQ13_sig, IRQ14_sig:
std_logic;
signal IRQ_state : std_logic_vector(3 downto 0);
signal IRQ_signal_state : std_logic_vector(1 downto 0);
-- 00=signal to 0, 01=wait, 10=wait, 11=signal to Z (=1)
signal IRQ_next_ready : std_logic;
begin
IRQ_count <= IRQ_state;
IRQ_next_ready_out <= IRQ_next_ready;

IRQ3 <= '0' when IRQ3_sig = '0' else 'Z';  --defines a latch
--with signal = '0' and an enable signal
IRQ4 <= '0' when IRQ4_sig = '0' else 'Z';
IRQ5 <= '0' when IRQ5_sig = '0' else 'Z';
IRQ6 <= '0' when IRQ6_sig = '0' else 'Z';
IRQ7 <= '0' when IRQ7_sig = '0' else 'Z';
IRQ9 <= '0' when IRQ9_sig = '0' else 'Z';
IRQ10 <= '0' when IRQ10_sig = '0' else 'Z';
IRQ11 <= '0' when IRQ11_sig = '0' else 'Z';
IRQ12 <= '0' when IRQ12_sig = '0' else 'Z';
IRQ13 <= '0' when IRQ13_sig = '0' else 'Z';
IRQ14 <= '0' when IRQ14_sig = '0' else 'Z';

process(clk, reset)
begin
  if reset='1' then
    IRQ_state <= "0011"; --IRQ3 of all
                         --IRQ=(3,4,5,6,7,9,10,11,12,13,14)
    IRQ_next_ready <= '0';
    IRQ_signal_state <= "00";
    IRQ3_sig <= '1';
    IRQ4_sig <= '1';
    IRQ5_sig <= '1';
    IRQ6_sig <= '1';
    IRQ7_sig <= '1';
    IRQ9_sig <= '1';
    IRQ10_sig <= '1';
    IRQ11_sig <= '1';
    IRQ12_sig <= '1';
    IRQ13_sig <= '1';
    IRQ14_sig <= '1';
  elsif rising_edge(clk) then
    if IRQ_poll_ok = '1' and IRQ_next_ready = '1' then
      case IRQ_state is
        when "0011" => --------------------IRQ3-----------------
          if IRQ_in = '1' then
--if the irq has been pulled on a slave node the master node
--pulls the corresponding interrupt and holds it for 3 clock
--cycles (internal clk)

            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ3_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ3_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ3_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ3_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "0100";   --sets the next IRQ to be
                                     --polled
            end if;
              IRQ7_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "1001";
            end if;
          else
            IRQ7_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "1001";
          end if;
        when "1001" => --------------------IRQ9----------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ9_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ9_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ9_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ9_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "1010";
            end if;
          else
            IRQ9_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "1010";
          end if;
        when "1010" => --------------------IRQ10---------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ10_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ10_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ10_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ10_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "1011";
            end if;
          else
            IRQ10_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "1011";
          end if;
        when "1011" => --------------------IRQ11---------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ11_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ11_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ11_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ11_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "1100";
            end if;
          else
            IRQ11_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "1100";
          end if;
        when "1100" => --------------------IRQ12---------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ12_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ12_sig <= '0';
```

```
          else
            IRQ3_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "0100";
          end if;
        when "0100" => --------------------IRQ4-----------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ4_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ4_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ4_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ4_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "0101";
            end if;
          else
            IRQ4_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "0101";
          end if;
        when "0101" => --------------------IRQ5-----------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ5_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ5_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ5_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ5_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "0110";
            end if;
          else
            IRQ5_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "0110";
          end if;
         when "0110" => --------------------IRQ6-----------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ6_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ6_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ6_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ6_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "0111";
            end if;
          else
            IRQ6_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "0111";
          end if;
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ12_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ12_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "1101";
            end if;
          else
            IRQ12_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "1101";
          end if;
        when "1101" => --------------------IRQ13---------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ13_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ13_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ13_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ13_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "1110";
            end if;
          else
            IRQ13_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "1110";
          end if;
        when "1110" => --------------------IRQ14---------------
          if IRQ_in = '1' then
            if IRQ_signal_state = "00" then
              IRQ_signal_state <= "01";
              IRQ14_sig <= '0';
            elsif IRQ_signal_state = "01" then
              IRQ_signal_state <= "10";
              IRQ14_sig <= '0';
            elsif IRQ_signal_state = "10" then
              IRQ_signal_state <= "11";
              IRQ14_sig <= '0';
            elsif IRQ_signal_state = "11" then
              IRQ_signal_state <= "00";
              IRQ14_sig <= '1';
              IRQ_next_ready <= '0';
              IRQ_state <= "0011";
            end if;
          else
            IRQ14_sig <= '1';
            IRQ_next_ready <= '0';
            IRQ_signal_state <= "00";
            IRQ_state <= "0011";
          end if;
        when others =>
          IRQ_state <= "0011";
          IRQ_next_ready <= '0';
      end case;
    elsif IRQ_poll_ok = '0' then
      IRQ_next_ready <= '1';
    end if;
  end if;
end process;
end Behavioral;
```

B.7 IRQ_timer.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    IRQ_timer - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Generates the timeout signal indicating that
-- no data has been transferred on the bus in the specified
-- interval
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity IRQ_timer is
    Port ( clk : in std_logic;
           IRQ_timer_reset : in std_logic;
           IRQ_timeout : out std_logic);
end IRQ_timer;
```

```
architecture Behavioral of IRQ_timer is
signal counter : std_logic_vector(15 downto 0);
begin

--To generate timout at 15 us
--Vid MHZ clk -----count to
--10              150 = X"0096"
--25              375 = X"0177"
--50              750 = X"02EE"
--100             1500 = X"05DC"
--200             3000 = X"0BB8"

  timer:process(clk, IRQ_timer_reset)
  begin
    if IRQ_timer_reset = '1' then
      IRQ_timeout <= '0';
      counter <= (others => '0');
    elsif clk'event and clk ='1' then
      if counter <= X"0022" then --clk = 50 MHz   => X"0022" =
                                 --702 ns IRQ_poll
        counter <= counter +1;
      else
        IRQ_timeout <= '1';   --indicates that no data has been
--transferren on the bus in the specified interval and next
--interrupt could then be polled.
      end if;
    end if;
  end process;
end Behavioral;
```

B.8 ISA_bus_15us_timer.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    ISA_bus_15us_timer - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Generates a timeout signal after about 13 us
-- of ISA bus holding. The ISA bus is held by the use of the
-- CHRDY signal. The timeout will relese the bus.
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity ISA_bus_15us_timer is
    Port ( clk, reset : in std_logic;
           ISA_15us_timeout : out std_logic);
end ISA_bus_15us_timer;
```

```
architecture Behavioral of ISA_bus_15us_timer is
signal counter : std_logic_vector(15 downto 0);
begin

--To generate timout at 15 us
--Vid MHZ clk -----count to
--10              150 = X"0096"
--25              375 = X"0177"
--50              750 = X"02EE"
--100             1500 = X"05DC"
--200             3000 = X"0BB8"

  timer:process(clk, reset)
  begin
    if reset = '1' then
      ISA_15us_timeout <= '0';
      counter <= (others => '0');
    elsif clk'event and clk ='1' then
      if counter <= X"02BC" then --clk = 50 MHz => X"02BC"
        counter <= counter +1;
      else
        ISA_15us_timeout <= '1';
      end if;
    end if;
  end process;

end Behavioral;
```

B.9 ISA_Input_flipflop.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    ISA_Input_flipflop - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Samples the asynchronous ISA bus and
-- generatates synchronous output signals to the
-- master_node_state_machine.
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ISA_Input_flipflop is
    Port ( clk : in std_logic;

           BCLK : in std_logic;
           IOWC : in std_logic;
           IORC : in std_logic;
           AEN : in std_logic;
           SBHE : in std_logic;
           BALE : in std_logic;
           SA : in std_logic_vector(9 downto 0);
           BCLK_out : out std_logic;
           IOWC_out : out std_logic;
           IORC_out : out std_logic;
           AEN_out : out std_logic;
           SBHE_out : out std_logic;
           BALE_out : out std_logic;
           SA_out : out std_logic_vector(9 downto 0));
end ISA_Input_flipflop;

architecture Behavioral of ISA_Input_flipflop is

begin
  Dvippa: process(clk)  -- Inferes a D-flipflop to sample the
                        -- ISA bus with.
  begin
    if rising_edge(clk) then
      BCLK_out <= BCLK;
      IOWC_out <= IOWC;
      IORC_out <= IORC;
      AEN_out <= AEN;
      SBHE_out <= SBHE;
      SA_out <= SA;
      BALE_out <= BALE;
    end if;
  end process;
end Behavioral;
```

B.10 ISA_slave.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    ISA_slave - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    This state-machine acts as a slave device on
the ISA bus
--                      connected to the host computer.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ISA_slave is
    Port ( SA : in std_logic_vector(9 downto 0);
           BCLK : in std_logic;
           AEN : in std_logic; --Check that AEN = '0' => no DMA
           SBHE : in std_logic;
           IORC : in std_logic;
           IOWC : in std_logic;
           IO16 : out std_logic;
           CHRDY : out std_logic;
           SD : inout std_logic_vector(15 downto 0) := (others
=> 'Z');
           BALE : in std_logic;
           clk : in std_logic;
           reset : in std_logic;
           sw0 : in std_logic;
--if sw0 = '0' => activates internal register (on the master)
--at 2E8 tom 2EF
--if sw0 = '1' the addressed data in the interval is sent to to
--the slaves.
           address_output : out std_logic_vector(9 downto 0);
           data_output : out std_logic_vector(15 downto 0);
           SBHE_out : out std_logic;
           dev_data_in : in std_logic_vector(15 downto 0);
        if BCLK='0' then
          IO16_sig <= '0';
          CHRDY_sig <= '0';
          reset_chrdy_timer <= '0';
          data_output <= (others => '1');
          SD <= (others => 'Z');
          dev_read <= '0';
          dev_write <= '0';
          reg_read <= '0';
          reg_write <= '0';
          state <= write_delay2_SW2;
        else
          IO16_sig <= '0';
          CHRDY_sig <= '0';
          reset_chrdy_timer <= '0';
          data_output <= (others => '1');
          SD <= (others => 'Z');
          dev_read <= '0';
          dev_write <= '0';
          reg_read <= '0';
          reg_write <= '0';
        end if;
      when write_delay2_SW2 =>
        if BCLK='1' then    --Bus data is stable (after waiting
                            --some BCLKcycles)
          IO16_sig <= '0';
          CHRDY_sig <= '0';
          reset_chrdy_timer <= '0';
          data_output <= SD;
          SD <= (others => 'Z');
          dev_read <= '0';
          reg_read <= '0';
          if (SA(9 downto 3) = "1111101" and sw0='0') or (SA(9
downto 0) = "1111101001") then
--3E8,3EA to 3EF is switchable registers. Switched by slave_bit
--in 3E9. 3E9 is always connected to the Master
            reg_write <= '1';
            dev_write <= '0';
          else
            reg_write <= '0';
            dev_write <= '1';
          end if;
          state <= write_waitfortransmission_SW3;
        else
```

```
          reg_data_in : in std_logic_vector(15 downto 0);
          dev_transmission_ok : in std_logic;   --indicates if
--the device module has data ready on bus
          reg_transmission_ok : in std_logic;   --indicates if
--the register module has data ready on bus
          dev_read : out std_logic;
          dev_write : out std_logic;
          reg_read : out std_logic;
          reg_write : out std_logic;
          reset_chrdy_timer : out std_logic);
end ISA_slave;

architecture Behavioral of ISA_slave is
type state_type is (Groundstate_S0, Address_latch_SRW7,
Command_SRW8, write_delay1_SW1, write_delay2_SW2,
write_waitfortransmission_SW3, Read_waitfortransmission_SR4,
Read_CHRDYdelay1_SR5, Read_CHRDYdelay2_SR5,
Read_CHRDYdelay3_SR5, Endcycle_SRW9);
signal state : state_type:= Endcycle_SRW9;
signal CHRDY_sig, IO16_sig : std_logic;
begin

CHRDY <= '0' when CHRDY_sig = '0' else 'Z';
IO16 <= '0' when IO16_sig = '0' else 'Z';

p0: process(CLK, reset)
begin
  if reset = '1' then
    state <= Groundstate_S0; --Wait for next bus cycle so that
                             --groundstate_S0 triggers in
                             --beginning of command
    IO16_sig <= '1';
    dev_read <= '0';
    dev_write <= '0';
    reg_read <= '0';
    reg_write <= '0';
    CHRDY_sig <= '1';
    reset_chrdy_timer <= '1';
    data_output <= (others => '1');
    address_output <= (others => '1');
    SBHE_out <= '1';
    SD <= (others => 'Z');
  elsif CLK'event and CLK = '1' then  -- Sampling clock faster
                                      -- than 16 MHz if BCLK is
                                      -- 8 MHZ
    case state is  --ISA-State machine
      when Groundstate_S0 =>
        if BALE = '1' then  -- triggers on BALE and starts the
                            -- bus sampling cycle
          state <= Address_latch_SRW7;
          IO16_sig <= '1';
          dev_read <= '0';
          dev_write <= '0';
          reg_read <= '0';
          reg_write <= '0';
          CHRDY_sig <= '1';
          reset_chrdy_timer <= '1';
          data_output <= (others => '1');
          address_output <= (others => '1');
          SBHE_out <= '1';
          SD <= (others => 'Z');
        end if;
      when Address_latch_SRW7 => --Checks if address is in range
         if BALE = '0' then
           if dev_transmission_ok='0' and
reg_transmission_ok='0' and AEN='0' and
           (SA(9 downto 0) = "0010000000" or
            SA(9 downto 1) = "100111100" or
            SA(9 downto 3) = "1011101" or  --2E8 tom 2EF  --744
            SA(9 downto 3) = "1111101"   --3E8 tom 3EF    --1000
          )then  --hex: 278, 279, 80 == ok and AEN and
                 --transmissionsignals is reset
             state <= Command_SRW8;
             IO16_sig <= '0';
             dev_read <= '0';
             dev_write <= '0';
             reg_read <= '0';
             reg_write <= '0';
             CHRDY_sig <= '1';
             reset_chrdy_timer <= '1';
             data_output <= (others => '1');
             address_output <= SA;
             SBHE_out <= SBHE;
             SD <= (others => 'Z');
           else
             state <= Groundstate_S0;
             IO16_sig <= '1';
           end if;
      when write_delay1_SW1 =>
           IO16_sig <= '0';
           CHRDY_sig <= '0';
           reset_chrdy_timer <= '0';
           data_output <= (others => '1');
           SD <= (others => 'Z');
           dev_read <= '0';
           dev_write <= '0';
           reg_read <= '0';
           reg_write <= '0';
         end if;
      when write_waitfortransmission_SW3 =>
        if dev_transmission_ok='1' or reg_transmission_ok='1'
then
           IO16_sig <= '0';
           CHRDY_sig <= '1';
           reset_chrdy_timer <= '1';
           data_output <= (others => '1');
           SD <= (others => 'Z');
           dev_read <= '0';
           --dev_write <= '0';
           reg_read <= '0';
           --reg_write <= '0';
           state <= Endcycle_SRW9;
         else
           IO16_sig <= '0';
           CHRDY_sig <= '0';
           reset_chrdy_timer <= '0';
           SD <= (others => 'Z');
           dev_read <= '0';
           reg_read <= '0';
         end if;
      when Read_waitfortransmission_SR4 =>
-- wait for reg/dev transmission to complete
         if dev_transmission_ok = '1' then  --=> will put data
--on ISA bus and wait for next buscycle
           IO16_sig <= '0';
           CHRDY_sig <= '0';
           reset_chrdy_timer <= '0';
           data_output <= (others => '1');
           SD <= dev_data_in;
           --dev_read <= '0';
           dev_write <= '0';
           --reg_read <= '0';
           reg_write <= '0';
           state <= Read_CHRDYdelay1_SR5;
         elsif reg_transmission_ok = '1' then  --=> will put
--data on ISA bus and wait for next buscycle
           IO16_sig <= '0';
           CHRDY_sig <= '0';
           reset_chrdy_timer <= '0';
           data_output <= (others => '1');
           SD <= reg_data_in;
           --dev_read <= '0';
           dev_write <= '0';
           --reg_read <= '0';
           reg_write <= '0';
           state <= Read_CHRDYdelay1_SR5;
         else
           IO16_sig <= '0';
           CHRDY_sig <= '0';
           reset_chrdy_timer <= '0';
           data_output <= (others => '1');
           SD <= (others => 'Z');
           dev_write <= '0';
           reg_write <= '0';
         end if;
      when Read_CHRDYdelay1_SR5 =>  -- data wait 1 - wait state
-- 1-3 holds data on the ISA bus a  while before releasing the
-- bus (CHRDY='1')
         if BCLK='1' then
           IO16_sig <= '0';
           CHRDY_sig <= '0';
           reset_chrdy_timer <= '0';
           data_output <= (others => '1');
           --dev_read <= '0';
           dev_write <= '0';
           --reg_read <= '0';
           reg_write <= '0';
           state <= Read_CHRDYdelay2_SR5;
         else
           IO16_sig <= '0';
           CHRDY_sig <= '0';
           reset_chrdy_timer <= '0';
           data_output <= (others => '1');
           --dev_read <= '0';
           dev_write <= '0';
           --reg_read <= '0';
```

```
            dev_read <= '0';
            dev_write <= '0';
            reg_read <= '0';
            reg_write <= '0';
            CHRDY_sig <= '1';
            reset_chrdy_timer <= '1';
            data_output <= (others => '1');
            address_output <= (others => '1');
            SBHE_out <= SBHE;
            SD <= (others => 'Z');
          end if;
        else
          IO16_sig <= '1';
          dev_read <= '0';
          dev_write <= '0';
          reg_read <= '0';
          reg_write <= '0';
          CHRDY_sig <= '1';
          reset_chrdy_timer <= '1';
          data_output <= (others => '1');
          address_output <= (others => '1');
          SBHE_out <= '1';
          SD <= (others => 'Z');
        end if;
      when Command_SRW8 =>   --waits for an ISA bus command
        if IOWC='0' then
          state <= write_delay1_SW1;
          IO16_sig <= '0';
          dev_read <= '0';
          dev_write <= '0';
          reg_read <= '0';
          reg_write <= '0';
          CHRDY_sig <= '0';
          reset_chrdy_timer <= '0';
          data_output <= (others => '1');
          address_output <= SA;
          SBHE_out <= SBHE;
          SD <= (others => 'Z');
        elsif IORC='0' then
          state <= Read_waitfortransmission_SR4;
          IO16_sig <= '0';
          dev_write <= '0';
          reg_write <= '0';
          CHRDY_sig <= '0';
          reset_chrdy_timer <= '0';
          data_output <= (others => '1');
          address_output <= SA;
          SBHE_out <= SBHE;
          SD <= (others => 'Z');
          if (SA(9 downto 3) = "1111101" and sw0='0') or
            (SA(9 downto 0) = "1111101001") then
--3E8,3EA to 3EF is switchable registers. Switched by slave_bit
--in 3E9. 3E9 is always connected to the Master
            reg_read <= '1';
            dev_read <= '0';
          else
            reg_read <= '0';
            dev_read <= '1';
          end if;
        elsif BALE = '1' then
          state <= Address_latch_SRW7;
          IO16_sig <= '1';
          dev_read <= '0';
          dev_write <= '0';
          reg_read <= '0';
          reg_write <= '0';
          CHRDY_sig <= '1';
          reset_chrdy_timer <= '0';
          data_output <= (others => '1');
          address_output <= SA;
          SBHE_out <= SBHE;
          SD <= (others => 'Z');
        else
          IO16_sig <= '0';
          dev_read <= '0';
          dev_write <= '0';
          reg_read <= '0';
          reg_write <= '0';
          CHRDY_sig <= '1';
          reset_chrdy_timer <= '1';
          data_output <= (others => '1');
          address_output <= SA;
          SBHE_out <= SBHE;
          SD <= (others => 'Z');
        end if;
      when write_delay1_SW1 =>  --Waits till written data has
-- stabilized on the bus before reading
          reg_write <= '0';
        end if;
      when Read_CHRDYdelay2_SR5 =>        -- data wait 2
        if BCLK='0' then
          IO16_sig <= '0';
          CHRDY_sig <= '0';
          reset_chrdy_timer <= '0';
          data_output <= (others => '1');
          --dev_read <= '0';
          dev_write <= '0';
          --reg_read <= '0';
          reg_write <= '0';
          state <= Read_CHRDYdelay3_SR5;
        else
          IO16_sig <= '0';
          CHRDY_sig <= '0';
          reset_chrdy_timer <= '0';
          data_output <= (others => '1');
          --dev_read <= '0';
          dev_write <= '0';
          --reg_read <= '0';
          reg_write <= '0';
        end if;
      when Read_CHRDYdelay3_SR5 =>        -- data wait 3
        if BCLK='1' then
          IO16_sig <= '0';
          CHRDY_sig <= '1';
          reset_chrdy_timer <= '1';
          data_output <= (others => '1');
          --dev_read <= '0';
          dev_write <= '0';
          --reg_read <= '0';
          reg_write <= '0';
          state <= Endcycle_SRW9;
        else
          IO16_sig <= '0';
          CHRDY_sig <= '0';
          reset_chrdy_timer <= '0';
          data_output <= (others => '1');
          --dev_read <= '0';
          dev_write <= '0';
          --reg_read <= '0';
          reg_write <= '0';
        end if;

      when Endcycle_SRW9 =>  --End of transmission. Wait for
                             --next bus cycle and command reset
        if IORC='1' and IOWC='1' then
          IO16_sig <= '1';
          dev_read <= '0';
          dev_write <= '0';
          reg_read <= '0';
          reg_write <= '0';
          CHRDY_sig <= '1';
          reset_chrdy_timer <= '1';
          data_output <= (others => '1');
          address_output <= (others => '1');
          SD <= (others => 'Z');
          state <= Groundstate_S0;
        else
          IO16_sig <= '0';
          CHRDY_sig <= '1';
          reset_chrdy_timer <= '1';
          data_output <= (others => '1');
          address_output <= (others => '1');
        end if;
      when others =>
        state <= Endcycle_SRW9;--Takes care of undefined states
        IO16_sig <= '1';
        dev_read <= '0';
        dev_write <= '0';
        reg_read <= '0';
        reg_write <= '0';
        CHRDY_sig <= '1';
        reset_chrdy_timer <= '1';
        data_output <= (others => '1');
        address_output <= (others => '1');
        SBHE_out <= '1';
        SD <= (others => 'Z');
    end case;
  end if;
end process;
end Behavioral;
```

B.11 LED_display4digit.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node and Slave Node
-- Module Name:    LED_display4digit - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Displays a 16 bit value on the led display on
--                 the Spartan 3 development board. For debugging
--                 purposes only.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LED_display4digit is
    Port ( Value_16bit : in std_logic_vector(15 downto 0);    --16 bitS Value, 4 digits of 4 bits each
           clk, disp_on : in std_logic;                       --clk fast a couple of MHz, disp_on lights the digits
           an : out std_logic_vector(3 downto 0);
           segments_atodp : out std_logic_vector(7 downto 0));
end LED_display4digit;

architecture Behavioral of LED_display4digit is
signal loadvalue :  std_logic:='1';
signal statevar : std_logic_vector(1 downto 0):="00";
signal digvalue : std_logic_vector(3 downto 0);
signal delay : std_logic_vector(15 downto 0):="0000000000000000";

begin
  process(clk,disp_on)
  begin
  if disp_on='0' then
     an <= "1111";                                -- Turns all figures in the display off
  elsif clk'event and clk='1' then
    delay <= delay + 1;
    if loadvalue='1' and delay = "00000000" then   -- after the specified delay the next figure is chosen
      loadvalue<='0';
      case statevar is
        when "00" =>
          an <= "1110";
          digvalue(0) <= Value_16bit(0);
          digvalue(1) <= Value_16bit(1);
          digvalue(2) <= Value_16bit(2);
          digvalue(3) <= Value_16bit(3);
       when "01" =>
          an <= "1101";
          digvalue(0) <= Value_16bit(4);
          digvalue(1) <= Value_16bit(5);
          digvalue(2) <= Value_16bit(6);
          digvalue(3) <= Value_16bit(7);
        when "10" =>
          an <= "1011";
          digvalue(0) <= Value_16bit(8);
          digvalue(1) <= Value_16bit(9);
          digvalue(2) <= Value_16bit(10);
          digvalue(3) <= Value_16bit(11);
        when "11" =>
          an <= "0111";
          digvalue(0) <= Value_16bit(12);
          digvalue(1) <= Value_16bit(13);
          digvalue(2) <= Value_16bit(14);
          digvalue(3) <= Value_16bit(15);
        when others => null;
      end case;
    end if;
    if loadvalue='0' then
      loadvalue<='1';
      statevar <= statevar + 1;  --activates next figure
      case digvalue is           --sets the segment bit pattern to the corresponding hex number
        when "0000" => segments_atodp <= "00000011";  --0  last bit is a decimal point
        when "0001" => segments_atodp <= "10011111";  --1
        when "0010" => segments_atodp <= "00100101";  --2
        when "0011" => segments_atodp <= "00001101";  --3
        when "0100" => segments_atodp <= "10011001";  --4
        when "0101" => segments_atodp <= "01001001";  --5
        when "0110" => segments_atodp <= "01000001";  --6
        when "0111" => segments_atodp <= "00011111";  --7
        when "1000" => segments_atodp <= "00000001";  --8
```

```
        when "1001" => segments_atodp <= "00001001";  --9
        when "1010" => segments_atodp <= "00010001";  --A
        when "1011" => segments_atodp <= "11000001";  --b
        when "1100" => segments_atodp <= "01100011";  --C
        when "1101" => segments_atodp <= "10000101";  --d
        when "1110" => segments_atodp <= "01100001";  --E
        when "1111" => segments_atodp <= "01110001";  --F
        when others => null;
      end case;
    end if;
  end if;
  end process;
end Behavioral;
```

B.12 LVDS_master_transm_timer.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    LVDS_master_transm_timer - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Generates a timeout signal when the
--                 addressed slave node has not generated a
--                 reply in the specified time interval.
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LVDS_master_transm_timer is
    Port ( clk, reset : in std_logic;
           master_transmission_timout : out std_logic);
end LVDS_master_transm_timer;
```

```
architecture Behavioral of LVDS_master_transm_timer is
signal counter : std_logic_vector(15 downto 0);
begin

--To generate timout at 15 us
--Vid MHZ clk -----count to
--10               150 = X"0096"
--25               375 = X"0177"
--50               750 = X"02EE"
--100              1500 = X"05DC"
--200              3000 = X"0BB8"

  timer:process(clk, reset)
  begin
    if reset = '1' then
      master_transmission_timout <= '0';
      counter <= (others => '0');
    elsif clk'event and clk ='1' then --X"0B4"  && clk = 50 MHz
                                      --=> 3,600 us
      if counter <= X"0B4" then       --X"0019" && clk = 50 MHz
                                      --=> 0,540 us
        counter <= counter + 1;       --X"0032" && clk = 50 MHz
                                      --=> 1 us
      else
        master_transmission_timout <= '1';
      end if;
    end if;
  end process;

end Behavioral;
```

B.13 Error_code_checker.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node and Slave Node
-- Module Name:    Error_code_checker - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Compares error checking code in the data
block with data
--                     transmitted by lvds between modules.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity Error_code_checker is
    Port ( clk, reset : in std_logic;
           LVDS_link_in_ready : in std_logic;
           a : in std_logic_vector(39 downto 0);  --Data block
--received from lvds link
           Data_out : out std_logic_vector(15 downto 0);
           Transmission_ok : out std_logic;     --Indicates that
--reception is finished and NO errors detected
```

```
  ERR_2(3) <= a(11) xor a(27);
  ERR_2(4) <= a(12) xor a(28);
  ERR_2(5) <= a(13) xor a(29);
  ERR_2(6) <= a(14) xor a(30);
  ERR_2(7) <= a(15) xor a(31);
  ERR_code(0) <= ERR_1(0) xor ERR_2(0);
  ERR_code(1) <= ERR_1(1) xor ERR_2(1);
  ERR_code(2) <= ERR_1(2) xor ERR_2(2);
  ERR_code(3) <= ERR_1(3) xor ERR_2(3);
  ERR_code(4) <= ERR_1(4) xor ERR_2(4);
  ERR_code(5) <= ERR_1(5) xor ERR_2(5);
  ERR_code(6) <= ERR_1(6) xor ERR_2(6);
  ERR_code(7) <= ERR_1(7) xor ERR_2(7);

process(clk,reset)
begin
  if reset='1' then
    address <= (others => '1');
    Data_out <= (others => '1');
    Infobits <= (others => '0');
    Read1_write0 <= '0';
    SBHE <= '0';
    Transmission_ok <= '0';
    Transmission_bad <= '0';
    Check_error_code <= '0';
    done <= '0';
  elsif rising_edge(clk) then
    if LVDS_link_in_ready = '1' and Check_error_code = '0' then
      Check_error_code <= '1';
    end if;
```

```
          Transmission_bad : out std_logic;    --Indicates that
--reception is finished and errors detected
          Read1_write0 : out std_logic;
          SBHE : out std_logic;
          Infobits : out std_logic_vector(3 downto 0);
          address : out std_logic_vector(9 downto 0));
end Error_code_checker;

architecture Behavioral of Error_code_checker is
signal Check_error_code, done : std_logic;
signal ERR_1 : std_logic_vector(7 downto 0);
signal ERR_2 : std_logic_vector(7 downto 0);
signal ERR_code : std_logic_vector(7 downto 0);
begin

--Generates the error checking code from data

  ERR_1(0) <= a(0) xor a(16);
  ERR_1(1) <= a(1) xor a(17);
  ERR_1(2) <= a(2) xor a(18);
  ERR_1(3) <= a(3) xor a(19);
  ERR_1(4) <= a(4) xor a(20);
  ERR_1(5) <= a(5) xor a(21);
  ERR_1(6) <= a(6) xor a(22);
  ERR_1(7) <= a(7) xor a(23);

  ERR_2(0) <= a(8) xor a(24);
  ERR_2(1) <= a(9) xor a(25);
  ERR_2(2) <= a(10) xor a(26);
```

```
    if Check_error_code = '1' and done = '0' then
--Because the lvds receiver has another clk than the internal
--logic this wait step is introduced so that the data is be
--stablilized.
      if ERR_code = a(39 downto 32) and a(26) = '0' then
        Data_out <= a(15 downto 0);   --a(26) = resend_request
        address <= a(25 downto 16);   --a(28) = poll request
        Transmission_ok <= '1';       --Error checking ok
        Transmission_bad <= '0';
        Read1_write0 <= a(31);
        SBHE <= a(30);
        Infobits <= a(29 downto 26);
        done <= '1';
      else
        Data_out <= (others => '1');
        address  <= (others => '1');
        Transmission_ok <= '0';
        Transmission_bad <= '1';      -- Error check not ok
        Read1_write0 <= '0';
        SBHE <= '0';
        Infobits <= (others => '0');
        done <= '1';
      end if;
    end if;
  end if;
end process;

end Behavioral;
```

B.14 Data_reorder_lvds_in.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node and Slave Node
-- Module Name:    data_reorder_lvds_in - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Reorders data between buses. Adds data from
--                 shift registers to an 40 bit data bus.
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity data_reorder_lvds_in is
    Port ( data_even    : in std_logic_vector(20 downto 0);
           data_odd     : in std_logic_vector(20 downto 0);
           dataut       : out std_logic_vector(39 downto 0));
end data_reorder_lvds_in;

architecture Behavioral of data_reorder_lvds_in is    --Reorder
the data wires from the shift registers
--to create the data block 'dataut'
begin
--  One stop bit is unconnected  == 0;    <= data_odd(0);

    dataut(0)   <= data_even(0);
    dataut(1)   <= data_odd(1);
    dataut(2)   <= data_even(1);
    dataut(3)   <= data_odd(2);
```

```
    dataut(4)   <= data_even(2);
    dataut(5)   <= data_odd(3);
    dataut(6)   <= data_even(3);
    dataut(7)    <= data_odd(4);
    dataut(8)    <= data_even(4);
    dataut(9)    <= data_odd(5);
    dataut(10)   <= data_even(5);
    dataut(11)   <= data_odd(6);
    dataut(12)   <= data_even(6);
    dataut(13)  <= data_odd(7);
    dataut(14)  <= data_even(7);
    dataut(15)  <= data_odd(8);
    dataut(16)  <= data_even(8);
    dataut(17)  <= data_odd(9);
    dataut(18)  <= data_even(9);
    dataut(19)  <= data_odd(10);
    dataut(20)  <= data_even(10);
    dataut(21)  <= data_odd(11);
    dataut(22)  <= data_even(11);
    dataut(23)  <= data_odd(12);
    dataut(24)  <= data_even(12);
    dataut(25)  <= data_odd(13);
    dataut(26)  <= data_even(13);
    dataut(27)   <= data_odd(14);
    dataut(28)   <= data_even(14);
    dataut(29)   <= data_odd(15);
    dataut(30)   <= data_even(15);
    dataut(31)   <= data_odd(16);
    dataut(32)   <= data_even(16);
    dataut(33)  <= data_odd(17);
    dataut(34)  <= data_even(17);
    dataut(35)  <= data_odd(18);
    dataut(36)  <= data_even(18);
    dataut(37)  <= data_odd(19);
    dataut(38)  <= data_even(19);
    dataut(39)  <= data_odd(20);

--  One startbit is unconnected == 1;    <= data_even(20);

end Behavioral;
```

B.15 Shift_sipo.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node and Slave Node
-- Module Name:    shift_sipo - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Shift register - serial in, parallel out
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shift_sipo is
    generic (reg_width: integer:= 21);  --reg_width = number of
                                        --bits on the output
    Port ( clk, reset, shift_en, d_in : in std_logic;
           shift_out : out std_logic_vector(reg_width-1 downto
0));
end shift_sipo;
```

```
architecture Behavioral of shift_sipo is  --The shift register
signal shift_reg:std_logic_vector(reg_width-1 downto 0);

begin
  p0:process(clk,reset)
  begin
    if reset='1' then
      shift_reg <= (others => '0');
    elsif clk'event and clk='1' then
      if shift_en='1' then
                shift_reg(reg_width-1 downto 1) <=
                shift_reg(reg_width-2 downto 0);
                shift_reg(0) <= d_in;
      end if;
    end if;
  end process;

shift_out <= shift_reg;

end Behavioral;
```

B.16 Shift_sipo_fullout.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    shift_sipo_fullout - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Shift register - serial in, parallel out.
--                 Indicates when a high bit has reached the
--                 highest position to signal that 40 bits from
--                 the lvds line has been received.
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shift_sipo_fullout is
    generic (reg_width: integer:= 21);
-- reg_width = number of bits on the output
    Port ( clk, reset, shift_en, d_in : in std_logic;
           full_out : out std_logic;
           shift_out : out std_logic_vector(reg_width-1 downto
0));
end shift_sipo_fullout;
```

```
architecture Behavioral of shift_sipo_fullout is
-- The shift register has the highest bit as an output the
-- startbit ('1') will then in the highest position indicate
-- that the register is full.
signal shift_reg:std_logic_vector(reg_width-1 downto 0);
begin
  p0:process(clk,reset)
  begin
    if reset='1' then
      shift_reg <= (others => '0');
      elsif clk'event and clk='1' then
      if shift_en='1' and shift_reg(shift_reg'high)='0' then
        shift_reg(reg_width-1 downto 1) <=
        shift_reg(reg_width-2 downto 0);
        shift_reg(0) <= d_in;
      end if;
    end if;
  end process;
  full_out <= shift_reg(shift_reg'high);  -- If the startbit
--has reached this position the register is full

shift_out <= shift_reg;

end Behavioral;
```

B.17 Master_node_state_machine.vhd

```
----------------------------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    Master_node_state_machine - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    This is the main state machine that handles the data flow
--                     between devices.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Master_node_state_machine is
    Port ( clk : in std_logic;
           reset : in std_logic;
           master_transmission_timeout : in std_logic;
           IRQ_timeout : in std_logic;
           lvds_out_ready : in std_logic;
           irq_next_ready : in std_logic;
           lvds_transmission_ok : in std_logic;
           lvds_transmission_bad : in std_logic;
           IRQ_count : in std_logic_vector(3 downto 0);      -- holds the number of the interrupt to be polled
           read_data : in std_logic;
           write_data : in std_logic;
           SBHE_in : in std_logic;
           Data_from_ISA_in : in std_logic_vector(15 downto 0);
           Data_from_lvds_in : in std_logic_vector(15 downto 0);
           Address_in : in std_logic_vector(9 downto 0);

           transmission_timer_reset : out std_logic;
           IRQ_timer_reset : out std_logic;
           transmission_ok : out std_logic;                  -- indicates to the isa bus that data transferred cycle is finished
           IRQ_poll_ok : out std_logic;                      -- indicates that the irq polling transmission is ok
           IRQ_line_status : out std_logic;                  -- indicates if the polled irq line has been interrupted
           LVDS_receive_reset : out std_logic;               -- reset the input registers in the lvds input module
           LVDS_transmit_enable : out std_logic;             -- enables the transmitter latch in the lvds io pad
           LVDS_data_send : out std_logic;                   -- signals to lvds module to send data
           read1_write0_out : out std_logic;
           SBHE_out : out std_logic;
           Data_out : out std_logic_vector(15 downto 0);     -- data to lvds to be sent to slave nodes
           Data_to_ISA : out std_logic_vector(15 downto 0);
           Address_out : out std_logic_vector(9 downto 0);
           info_bits_out : out std_logic_vector(3 downto 0);
           no_timout ,timeout_2nd ,timeout_1st : out std_logic); -- indicators for debugging purpose to indicate
                                                                 -- if transmission times out

end Master_node_state_machine;

architecture Behavioral of Master_node_state_machine is
--declares the different states in the state machine
type state_type is (Ground_state_S0, State_wait_for_command_S1, State_IRQ_LVDS_ack_SQ2, State_IRQ_enable_receptiption_SQ3,
State_IRQ_end_SQ4, State_IRQ_command_reset_SQ5, State_LVDS_ack_S2, State_enable_receptiption_S3, State_resend_or_end_S4);
signal State : state_type;
signal resend_count : std_logic; -- keeps track of if the data has been resent and should therefore not be resent again if
--                               errors.
begin

Mealy_syncout:process(clk,reset)  -- Mealy Statemachine with synchronous outputs
begin
  if reset='1' then
    State <= Ground_state_S0;
    IRQ_poll_ok <= '0';
    IRQ_timer_reset <= '1';
    transmission_ok <= '0';
    LVDS_receive_reset <= '1';
    LVDS_transmit_enable <= '0';
    transmission_timer_reset <= '1';
    resend_count <= '0';
    LVDS_data_send <= '0';
    IRQ_line_status <= '0';
    Data_to_ISA <= X"FFFF";
  elsif rising_edge(clk) then
    case State is
    when Ground_state_S0=>            -- Waits for commands from the ISA module to reset
      if read_data ='0' and write_data ='0' then
        State <= State_wait_for_command_S1;
```

```
      IRQ_poll_ok <= '0';
      transmission_ok <= '0';
      LVDS_receive_reset <= '1';
      LVDS_transmit_enable <= '0';
      transmission_timer_reset <= '1';
      IRQ_timer_reset <= '0';
      resend_count <= '0';
      LVDS_data_send <= '0';
    else
      IRQ_poll_ok <= '0';
      IRQ_timer_reset <= '1';
      transmission_ok <= '1';
      transmission_timer_reset <= '1';
    end if;
  when State_wait_for_command_S1=>  -- Waits for command from the ISA module
    if (read_data ='1' or write_data ='1') and lvds_out_ready='1' then          -- reads or write ISA bus data
      State <= State_LVDS_ack_S2;
      LVDS_transmit_enable <= '1';
      IRQ_timer_reset <= '1';
      read1_write0_out <= read_data;
      SBHE_out <= SBHE_in;
      Data_out <= Data_from_ISA_in;
      Address_out <= Address_in;
      info_bits_out <= "0000";
    elsif IRQ_timeout = '1' and lvds_out_ready='1' and irq_next_ready = '1' then  -- starts next interrupt poll
      State <= State_IRQ_LVDS_ack_SQ2;
      LVDS_transmit_enable <= '1';
      IRQ_timer_reset <= '1';
      read1_write0_out <= '0';
      SBHE_out <= '0';
      Data_out <= X"000" & IRQ_count;
      Address_out <= (others => '0');
      info_bits_out <= "0100";   --interrupt poll info bit
    end if;
  when State_IRQ_LVDS_ack_SQ2 =>  ------------------------------IRQ STATES---------------------------------
    if lvds_out_ready = '0' then  -- waits for the lvds module to start the data send transfer
      State <= State_IRQ_enable_receptiption_SQ3;
      LVDS_data_send <= '0';
    else
      transmission_timer_reset <= '0';
      LVDS_data_send <= '1';
    end if;
  when State_IRQ_enable_receptiption_SQ3=>
    if lvds_out_ready = '1' then
      State <= State_IRQ_end_SQ4;
      LVDS_transmit_enable <= '1';
      LVDS_receive_reset <= '1';
    end if;
  when State_IRQ_end_SQ4=>                --When data is sent stis state wait for an answer from the addressed module
    if lvds_transmission_ok = '1' then    --lvds reception is ok
      State <= State_IRQ_command_reset_SQ5;
      IRQ_poll_ok <= '1';
      IRQ_timer_reset <= '1';
      IRQ_line_status <= Data_from_lvds_in(0);
    elsif master_transmission_timeout = '1' or lvds_transmission_bad = '1' then
      -- Data is not received in specified time or is corrupt
      State <= State_wait_for_command_S1;
      LVDS_receive_reset <= '1';
      transmission_timer_reset <= '1';
      IRQ_poll_ok <= '0';
      IRQ_timer_reset <= '0';
    else
      LVDS_transmit_enable <= '0';
      LVDS_receive_reset <= '0';
    end if;
  when State_IRQ_command_reset_SQ5=>    -- Wait for IRQ command to reset
    if irq_next_ready = '0' or master_transmission_timeout = '1' then
      State <= State_wait_for_command_S1;
      IRQ_timer_reset <= '0';
      IRQ_poll_ok <= '0';
      transmission_timer_reset <= '1';
    end if;                      ---------------------- END OF IRQ STATES --------------------------------
  when State_LVDS_ack_S2=>  -- waits for the lvds module to start the data send transfer
    if lvds_out_ready = '0' then
      State <= State_enable_receptiption_S3;
      LVDS_data_send <= '0';
    else
      transmission_timer_reset <= '0';
      LVDS_data_send <= '1';
    end if;
  when State_enable_receptiption_S3=>
    if lvds_out_ready = '1' then
      State <= State_resend_or_end_S4;
      LVDS_transmit_enable <= '1';
      LVDS_receive_reset <= '1';
    end if;
  when State_resend_or_end_S4=>         --When data is sent stis state wait for an answer from the addressed module
```

```
      if lvds_transmission_ok = '1' then  --lvds reception is ok
        State <= Ground_state_S0;
        transmission_ok <= '1';
        transmission_timer_reset <= '1';
        no_timout <= '1';
        Data_to_ISA <= Data_from_lvds_in;
      elsif master_transmission_timeout = '1' or lvds_transmission_bad = '1' then
-- Data is not received in specified time or is corrupt
        if resend_count = '0' then        --if 'resend_count' = '0' data is resent for the first time
          State <= State_LVDS_ack_S2;
          LVDS_transmit_enable <= '1';
          transmission_timer_reset <= '1';
          LVDS_receive_reset <= '1';
          resend_count <= '1';
          timeout_1st <= '1';
        else
          State <= Ground_state_S0;
          Data_to_ISA <= X"FFFF";       --If transmission has failed two times the data reported is X"FFFF" indicating no answer
          LVDS_receive_reset <= '1';
          transmission_timer_reset <= '1';
          timeout_2nd <= '1';
        end if;
      else
        no_timout <= '0';
        timeout_2nd <= '0';
        timeout_1st <= '0';
        LVDS_transmit_enable <= '0';
        LVDS_receive_reset <= '0';
      end if;
    when others => State <= Ground_state_S0;
  end case;
  end if;
end process;
end Behavioral;
```

B.18 Baud_rate_clk.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node and Slave Node
-- Module Name:    Baud_rate_clk - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Handles the baud rate timing in the RS232
--                 modules.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Baud_rate_clk is
    Port ( clk : in std_logic;
           baud_count_MSByte : in std_logic_vector(7 downto 0);
           baud_count_LSByte : in std_logic_vector(7 downto 0);
           en_16_x_baud : out std_logic);
-- the clock rate generated on en_16_x_baud should be 16 times
-- the baud rate
end Baud_rate_clk;
```

```
architecture Behavioral of Baud_rate_clk is
signal baud_count_sig : std_logic_vector(15 downto 0) :=
X"0000";
begin

--BAUD_count = clk divisor = clk_rate / (baud_rate * 16) = has
--to be within 5 % from exact value
--BAUD_count = 50 MHz / (110 baud * 16) = 28409  = X"6EF9"
--(hex)
--BAUD_count = 50 MHz / (2400 baud * 16) = 1302 = X"0516" (hex)
--BAUD_count = 50 MHz / (9600 baud * 16) = 325.5  = X"0145"
--(hex)
--BAUD_count = 50 MHz / (19200 baud * 16) = 162.7 = X"00A2"
--(hex)

baud_timer: process(clk)
begin
  if clk'event and clk='1' then
    if baud_count_sig +1 = baud_count_MSByte &
baud_count_LSByte then  --count to the value set in the reg
      baud_count_sig <= X"0000";
      en_16_x_baud <= '1';
    else
      baud_count_sig <= baud_count_sig +1;
      en_16_x_baud <= '0';
    end if;
  end if;
end process baud_timer;
end Behavioral;
```

B.19 UART_registers.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node and Slave Node
-- Module Name:    UART_registers - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Maps data and data registers onto other
--                 registers hanled by
--                 the register manager.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity UART_registers is
    Port ( Transmitter_holding_reg : in std_logic_vector(7
downto 0);
           Transmitter_holding_reg_write : in std_logic;
-- indicates that the register has been written to
           Receiver_buffer_reg : out std_logic_vector(7 downto
0);
           Receiver_buffer_reg_read : in std_logic;
-- indicates that the register has been read from
           Divisor_low_byte : in std_logic_vector(7 downto 0);
           Divisor_high_byte : in std_logic_vector(7 downto 0);
           Fifo_status_reg : out std_logic_vector(7 downto 0);
           Divisor_low : out std_logic_vector(7 downto 0);
           Divisor_high : out std_logic_vector(7 downto 0);
           TX_data_in : out std_logic_vector(7 downto 0);
           TX_data_write : out std_logic;
           TX_buffer_data_present : in std_logic;
           TX_buffer_half_full : in std_logic;
           TX_buffer_full : in std_logic;
           RX_data_out : in std_logic_vector(7 downto 0);
           RX_data_read : out std_logic;
           RX_buffer_data_present : in std_logic;
           RX_buffer_half_full : in std_logic;
           RX_buffer_full : in std_logic);

end UART_registers;

architecture Behavioral of UART_registers is

begin

--Maps the registers onto the corresponding data wires in the
RS232 UART

  TX_data_in <= Transmitter_holding_reg;
  TX_data_write <= Transmitter_holding_reg_write;
  RX_data_read <= Receiver_buffer_reg_read;
  Receiver_buffer_reg <= RX_data_out;
  Divisor_low <= Divisor_low_byte;
  Divisor_high <= Divisor_high_byte;
  Fifo_status_reg(7) <= '0';
  Fifo_status_reg(6) <= RX_buffer_full;
  Fifo_status_reg(5) <= RX_buffer_half_full;
  Fifo_status_reg(4) <= RX_buffer_data_present;
  Fifo_status_reg(3) <= '0';
  Fifo_status_reg(2) <= TX_buffer_full;
  Fifo_status_reg(1) <= TX_buffer_half_full;
  Fifo_status_reg(0) <= TX_buffer_data_present;

end Behavioral;
```

B.20 Unused_Z_outputs.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Master Node
-- Module Name:    unused_Z_outputs - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Sets the termination to the input wires to
--                 high impedence = ('Z').
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity unused_Z_outputs is
    Port ( SA2 : out std_logic_vector(9 downto 0);
           NOWS : out std_logic);
end unused_Z_outputs;

architecture Behavioral of unused_Z_outputs is

begin
  SA2 <= (others => 'Z');
  NOWS <= 'Z';
end Behavioral;
```

B.21 I2C_controller.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    I2C_controller
-- Module Name:    I2C_controller - Behavioral
-- Project Name:   Distributed ISA - I2C
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    This is the main I2C state machine that
--                 handles the data flow between I2C modules.
--
-- Revision:       14
-- Revision date:  19 July 2005
----------------------------------------------------------------
        if SIPO_data(7 downto 1) = "1010101" then --I2C device
                                                  --address
          ackn <= '0';
          if SIPO_data(0) = '0' then   -- SIPO_data(0)==
                                       -- Read1/write0
            state <= SIPO_internal_address_S2;
          else --read
            state <= Read_data_S4;
            PISO_load_sig <= '1';
            case int_reg_address is
              when X"00" => PISO_data <= reg0_in;
              when X"01" => PISO_data <= reg1_in;
              when X"02" => PISO_data <= reg2_in;
              when X"03" => PISO_data <= reg3_in;
              when X"04" => PISO_data <= reg4_in;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity I2C_controller is
    Port ( sclk : in std_logic;
           reset : in std_logic;
           ackn : out std_logic;
           start_of_block : in std_logic;
           reset_start_signal : out std_logic;
           SIPO_is_full : in std_logic;
           SIPO_reset : out std_logic;
           SIPO_restart : out std_logic;
           SIPO_data : in std_logic_vector(7 downto 0);
           PISO_all_is_out : in std_logic;
           PISO_data : out std_logic_vector(7 downto 0);
           PISO_load : out std_logic;
           PISO_reset : out std_logic;
           reg0_in : in std_logic_vector(7 downto 0);
           reg1_in : in std_logic_vector(7 downto 0);
           reg2_in : in std_logic_vector(7 downto 0);
           reg3_in : in std_logic_vector(7 downto 0);
           reg4_in : in std_logic_vector(7 downto 0);
           reg5_out : out std_logic_vector(7 downto 0);
           reg6_out : out std_logic_vector(7 downto 0);
           reg7_out : out std_logic_vector(7 downto 0);
           reg8_out : out std_logic_vector(7 downto 0);
           reg9_out : out std_logic_vector(7 downto 0));
end I2C_controller;

architecture Behavioral of I2C_controller is

--declares the different states in the state machine
type state_type is (Start_of_package_S0, SIPO_I2C_address_S1,
SIPO_internal_address_S2, Restart_or_write_data_S3,
Read_data_S4);
--declares internal signal wires
signal State : state_type;
signal PISO_load_sig : std_logic;
signal int_reg_address : std_logic_vector(7 downto 0);
signal reg5_out_sig, reg6_out_sig, reg7_out_sig, reg8_out_sig,
reg9_out_sig : std_logic_vector(7 downto 0);
begin

reg5_out <= reg5_out_sig;
reg6_out <= reg6_out_sig;
reg7_out <= reg7_out_sig;
reg8_out <= reg8_out_sig;
reg9_out <= reg9_out_sig;
PISO_load <= PISO_load_sig;

Mealy_syncout:process(sclk,reset)  -- Mealy Statemachine with
--                                    synchronous outputs
begin
  if reset='1' then
    State <= Start_of_package_S0;
    reg5_out_sig <= (others => '0');
    reg6_out_sig <= (others => '0');
    reg7_out_sig <= (others => '0');
    reg8_out_sig <= (others => '0');
    reg9_out_sig <= (others => '0');
    PISO_reset <= '1';
    PISO_load_sig <= '0';
    PISO_data <= (others => '0');
    int_reg_address <= (others => '0');
    SIPO_reset <= '1';
    SIPO_restart <= '0';
    reset_start_signal <= '0';
    ackn <= '1';
  elsif falling_edge(sclk) then
    case State is
    when Start_of_package_S0=>
      if start_of_block = '1' then
        State <= SIPO_I2C_address_S1;
        reset_start_signal <= '1';
        SIPO_reset <= '0';
        SIPO_restart <= '1';
      else
        reset_start_signal <= '0';
        SIPO_reset <= '1';
        SIPO_restart <= '0';
      end if;
      Ackn <= '1';
      PISO_reset <= '0';
      PISO_load_sig <= '0';
    when SIPO_internal_address_S2=>
      if start_of_block = '1' then
        reset_start_signal <= '1';
        SIPO_reset <= '0';
        SIPO_restart <= '1';
        Ackn <= '1';
        State <= SIPO_I2C_address_S1;
      elsif SIPO_is_full = '1' then
        int_reg_address <= SIPO_data;
        ackn <= '0';
        SIPO_reset <= '1';
        SIPO_restart <= '0';
        reset_start_signal <= '0';
        State <= Restart_or_write_data_S3;
      else
        SIPO_reset <= '0';
        SIPO_restart <= '0';
        ackn <= '1';
        reset_start_signal <= '0';
      end if;
    when Restart_or_write_data_S3=>
      if start_of_block = '1' then
        reset_start_signal <= '1';
        SIPO_reset <= '0';
        SIPO_restart <= '1';
        Ackn <= '1';
        State <= SIPO_I2C_address_S1;
      elsif SIPO_is_full = '1' then  --write to address
        Ackn <= '0';
        State <= Start_of_package_S0;
        SIPO_reset <= '1';
        SIPO_restart <= '0';
        reset_start_signal <= '0';
        case int_reg_address is
          when X"05" => reg5_out_sig <= SIPO_data;
          when X"06" => reg6_out_sig <= SIPO_data;
          when X"07" => reg7_out_sig <= SIPO_data;
          when X"08" => reg8_out_sig <= SIPO_data;
          when X"09" => reg9_out_sig <= SIPO_data;
          when others => null;
        end case;
      else
        reset_start_signal <= '0';
        Ackn <= '1';
        SIPO_reset <= '0';
        SIPO_restart <= '0';
      end if;
    when Read_data_S4=>
      Ackn <= '1';
      if start_of_block = '1' then
        reset_start_signal <= '1';
        SIPO_reset <= '0';
        SIPO_restart <= '1';
        State <= SIPO_I2C_address_S1;
      elsif PISO_all_is_out = '1' and PISO_load_sig = '0' then
        State <= Start_of_package_S0;
        PISO_load_sig <= '0';
        SIPO_reset <= '1';
        SIPO_restart <= '0';
        reset_start_signal <= '0';
      else
        SIPO_reset <= '1';
        SIPO_restart <= '0';
        reset_start_signal <= '0';
        PISO_load_sig <= '0';
      end if;
    when others =>
      reset_start_signal <= '1';
      SIPO_reset <= '1';
      SIPO_restart <= '0';
            when X"05" => PISO_data <= reg5_out_sig;
            when X"06" => PISO_data <= reg6_out_sig;
            when X"07" => PISO_data <= reg7_out_sig;
            when X"08" => PISO_data <= reg8_out_sig;
            when X"09" => PISO_data <= reg9_out_sig;
            when others => PISO_data <= X"FF";
          end case;
        end if;
      else -- Wrong I2C device address
        Ackn <= '1';
        state <= Start_of_package_S0;
      end if;
    else
      Ackn <= '1';
      SIPO_reset <= '0';
      SIPO_restart <= '0';
    end if;
```

```
      PISO_data <= (others => '0');
    when SIPO_I2C_address_S1=>
      reset_start_signal <= '1';
      if SIPO_is_full = '1' then
        SIPO_reset <= '1';
        SIPO_restart <= '0';
      PISO_reset <= '1';
      State <= Start_of_package_S0;
  end case;
  end if;
end process;
end Behavioral;
```

B.22 Shift_piso_nbit.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    I2C-bus
-- Module Name:    shift_piso_nbit - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Shift register parallel in serial out
-- Revision:       14
-- Revision date:  20 July 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity I2C_PISO is
    generic (reg_width: integer:= 8);   --reg_width = number of
                                        --bits on the input
    Port ( clk, reset, load : in std_logic;
           d_in : in std_logic_vector(reg_width-1 downto 0);
           shift_out : out std_logic;
           all_is_out : out std_logic);
end I2C_PISO;

architecture Behavioral of I2C_PISO is

signal shift_reg:std_logic_vector(reg_width-1 downto 0);
signal counter:std_logic_vector(2 downto 0);
signal loaded:std_logic:='0';

begin
all_is_out <= '1' when counter = "111" else '0';
shift_out <= shift_reg(shift_reg'high);

process(clk,reset)
begin
  if reset='1' then
    shift_reg <= (others => '1');
    counter <= "111";
    loaded <= '0';
  elsif clk'event and clk='0' then    -- falling edge
    if load='0' then
      loaded<='0';
    end if;
    if load='1' and loaded='0' then
-- waits for the load signal to go low again before reloading
-- the shift register
      loaded<='1';
      shift_reg <= d_in;
      counter <= (others => '0');
    elsif counter < "111" then
      shift_reg(reg_width-1 downto 1) <= shift_reg(reg_width-2
downto 0); --Shifts the register
      shift_reg(0) <= '1';
      counter <= counter + 1;
    else
      shift_reg(shift_reg'high) <= '1';
    end if;
  end if;
end process;
end Behavioral;
```

B.23 I2C_register_selector.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    I2C-bus
-- Module Name:    I2C_register_selector - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Shift register parallel in serial out
-- Revision:       14
-- Revision date:  20 July 2005
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity I2C_register_selector is
    Port ( clk, reset : in std_logic;
           reg_select_register : in std_logic_vector(7 downto
0);
           reg_Write : in std_logic;
           input_register : in std_logic_vector(7 downto 0);
           output_register : out std_logic_vector(7 downto 0);
           reg0 : out std_logic_vector(7 downto 0);
           reg1 : out std_logic_vector(7 downto 0);
           reg2 : out std_logic_vector(7 downto 0);
           reg3 : out std_logic_vector(7 downto 0);
           reg4 : out std_logic_vector(7 downto 0);
           reg5 : in std_logic_vector(7 downto 0);
           reg6 : in std_logic_vector(7 downto 0);

reg0 <= reg0_sig;
reg1 <= reg1_sig;
reg2 <= reg2_sig;
reg3 <= reg3_sig;
reg4 <= reg4_sig;

with reg_select_register select
output_register <= reg0_sig when X"00" ,
                   reg1_sig when X"01" ,
                   reg2_sig when X"02" ,
                   reg3_sig when X"03" ,
                   reg4_sig when X"04" ,
                   reg5 when X"05" ,
                   reg6 when X"06" ,
                   reg7 when X"07" ,
                   reg8 when X"08" ,
                   reg9 when X"09" ,
                   X"FF" when others;

process(clk, reset)
begin
  if reset = '1' then
    reg0_sig <= (others => '0');
    reg1_sig <= (others => '0');
    reg2_sig <= (others => '0');
    reg3_sig <= (others => '0');
    reg4_sig <= (others => '0');
  elsif rising_edge(clk) then
    if reg_Write = '1' then
      case reg_select_register is
        when X"00" => reg0_sig<=input_register;
```

```
          reg7 : in std_logic_vector(7 downto 0);
          reg8 : in std_logic_vector(7 downto 0);
          reg9 : in std_logic_vector(7 downto 0));
end I2C_register_selector;

architecture Behavioral of I2C_register_selector is
signal reg0_sig,reg1_sig, reg2_sig,reg3_sig,reg4_sig :
std_logic_vector(7 downto 0);

begin
```

```
        when X"01" => reg1_sig<=input_register;
        when X"02" => reg2_sig<=input_register;
        when X"03" => reg3_sig<=input_register;
        when X"04" => reg4_sig<=input_register;
        when others => null;
      end case;
    end if;
  end if;
end process;
end Behavioral;
```

B.24 Shift_sipo.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    I2C-bus
-- Module Name:    shift_sipo - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Shift register - serial in, parallel out.
--                 Indicates when a high bit has reached the
--                 highest position.
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity I2C_SIPO is
    generic (reg_width: integer:= 8);
--reg_width = number of bits on the output
    Port ( clk, reset, restart, d_in : in std_logic;
           full_out : out std_logic;
           shift_out : out std_logic_vector(reg_width-1 downto
0));
end I2C_SIPO;
```

```
architecture Behavioral of I2C_SIPO is
signal restarted : std_logic;
signal counter : std_logic_vector(3 downto 0);
signal shift_reg : std_logic_vector(reg_width-1 downto 0);

begin
full_out <= '1' when counter = "1000" else '0';
p0:process(clk,reset)
begin
  if reset = '1' then
    counter <= "0000";
    shift_reg <= (others => '0');
  elsif clk'event and clk='1' then
    if restart = '1' and restarted = '0' then
      restarted <= '1';
      counter <= "0001";
      shift_reg(reg_width-1 downto 1) <= "0000000";
      shift_reg(0) <= d_in;
    elsif counter < "1000" then
      counter <= counter + 1;
      shift_reg(reg_width-1 downto 1) <=  shift_reg(reg_width-2
downto 0);
      shift_reg(0) <= d_in;
    end if;
    if restart = '0' then
      restarted <= '0';
    end if;
  end if;
end process;

shift_out <= shift_reg;

end Behavioral;
```

B.25 I2C_start_signal_detector.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    I2C-bus_Start_signal_detector
-- Module Name:    I2C_start_signal_detector - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Detects a start of package
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity I2C_start_signal_detector is
    Port ( clk25_100mhz : in std_logic;
           reset : in std_logic;
           sclk : in std_logic;
           sdata : in std_logic;
           start_signal : out std_logic);
end I2C_start_signal_detector;
```

```
architecture Behavioral of I2C_start_signal_detector is
signal counter : std_logic_vector(3 downto 0);

begin
process(clk25_100mhz,reset)
begin
  if reset='1' then
    start_signal <= '0';
    counter <= "0000";
  elsif clk25_100mhz'event and clk25_100mhz='1' then    --25 to
100 mhz gives delay of 70 to 290 ns
    if sdata = '1' then
      counter <= "0000";
    elsif sclk = '1' and counter < "1000" then
      counter <= counter +1;
      if counter = "0111" then
        start_signal <= '1';
      end if;
    else
      counter <= "1111";
    end if;
  end if;
end process;
end Behavioral;
```

B.26 Internal_register_manager_SNode.vhd

```
----------------------------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Slave Node
-- Module Name:    Internal_register_manager - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Manages the registers in the FPGA.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Internal_register_manager_SNode is
    Port ( clk, reset : in std_logic;
           Data_in : in std_logic_vector(15 downto 0);
           Data_out : out std_logic_vector(15 downto 0);
           Address_in : in std_logic_vector(9 downto 0);
           Read : in std_logic;
           Write : in std_logic;
           SBHE : in std_logic;
           reg_transmission_ok : out std_logic;

           I2C_reg_select_register : out std_logic_vector(7 downto 0);
           I2C_writedata_register : out std_logic_vector(7 downto 0);
           I2C_writedata_register_write : out std_logic;                --Wire signals if register is written to
           I2C_readdata_register : in std_logic_vector(7 downto 0);

           UART_Transmitter_holding_reg : out std_logic_vector(7 downto 0);
           UART_Transmitter_holding_reg_write : out std_logic;          --Wire signals if register is written to
           UART_Receiver_buffer_reg : in std_logic_vector(7 downto 0);
           UART_Receiver_buffer_reg_read : out std_logic;               --Wire signals if register is read from
           UART_Divisor_low_byte : out std_logic_vector(7 downto 0);
           UART_Divisor_high_byte : out std_logic_vector(7 downto 0);
           Fifo_status_reg : in std_logic_vector(7 downto 0));
end Internal_register_manager_SNode;

architecture Behavioral of Internal_register_manager_SNode is
--  Generates internal signals so that the data on the outputs may be read back again
--  This should have the same effect as if 'buffer' is used instead of 'out'.
--  Because the 'buffer' declaration has sometimes been misinterpreted by the simulator
--  this method is used instead, for safety reson.

signal I2C_reg_select_register_sig : std_logic_vector(7 downto 0);
signal I2C_writedata_register_sig : std_logic_vector(7 downto 0);
signal I2C_writedata_register_write_sig : std_logic;

signal UART_Transmitter_holding_reg_sig : std_logic_vector(7 downto 0);
signal UART_Transmitter_holding_reg_write_sig : std_logic;
signal UART_Receiver_buffer_reg_read_sig : std_logic;
signal UART_Divisor_low_byte_sig : std_logic_vector(7 downto 0);
signal UART_Divisor_high_byte_sig : std_logic_vector(7 downto 0);

signal reg_transmission_ok_sig : std_logic;

signal Scratch_reg_sig : std_logic_vector(7 downto 0);

begin
UART_Transmitter_holding_reg <= UART_Transmitter_holding_reg_sig;
UART_Transmitter_holding_reg_write <= UART_Transmitter_holding_reg_write_sig;
UART_Receiver_buffer_reg_read <= UART_Receiver_buffer_reg_read_sig;

reg_transmission_ok <= reg_transmission_ok_sig;

UART_Divisor_low_byte <= UART_Divisor_low_byte_sig;
UART_Divisor_high_byte <= UART_Divisor_high_byte_sig;
I2C_reg_select_register <= I2C_reg_select_register_sig;
I2C_writedata_register <= I2C_writedata_register_sig;
I2C_writedata_register_write <= I2C_writedata_register_write_sig;

process(clk, reset)
begin
  if reset = '1' then
    UART_Transmitter_holding_reg_sig <= (others => '0');
    UART_Transmitter_holding_reg_write_sig <= '0';
    UART_Receiver_buffer_reg_read_sig <= '0';
    UART_Divisor_low_byte_sig <= (others => '0');
    UART_Divisor_high_byte_sig <= (others => '0');
```

```
    reg_transmission_ok_sig <= '0';
    Scratch_reg_sig <= (others => '0');
    I2C_reg_select_register_sig <= (others => '0');
    I2C_writedata_register_sig <= (others => '0');
    I2C_writedata_register_write_sig <= '0';

  elsif rising_edge(clk) then
    if UART_Transmitter_holding_reg_write_sig = '1' or UART_Receiver_buffer_reg_read_sig = '1'
       or I2C_writedata_register_write_sig = '1' then
      UART_Transmitter_holding_reg_write_sig <= '0';
      UART_Receiver_buffer_reg_read_sig <= '0';
      I2C_writedata_register_write_sig <= '0';
    elsif reg_transmission_ok_sig = '1' then
      if read = '0' and write = '0' then
        reg_transmission_ok_sig <= '0';
      end if;
    elsif read = '1' then ----------------------------Read from registers-----------------------
      reg_transmission_ok_sig <= '1';
      if SBHE = '1' then --SBHE=1 => Signal bus is not high enabled - 8bit
        case address_in is
          when "11" & X"E8" => --1000
            Data_out(7 downto 0) <= UART_Receiver_buffer_reg;
            UART_Receiver_buffer_reg_read_sig <= '1';
          when "11" & X"E9" => --1001
            Data_out(7 downto 0) <= X"FF";
          when "11" & X"EA" => --1002
            Data_out(7 downto 0) <= UART_Divisor_low_byte_sig;
          when "11" & X"EB" => --1003
            Data_out(7 downto 0) <= UART_Divisor_high_byte_sig;
          when "11" & X"EC" => --1004
            Data_out(7 downto 0) <= Fifo_status_reg;
          when "11" & X"ED" => --1005
            Data_out(7 downto 0) <= Scratch_reg_sig;
          when "11" & X"EE" => --1006
            Data_out(7 downto 0) <= I2C_reg_select_register_sig;
          when "11" & X"EF" => --1007
            Data_out(7 downto 0) <= I2C_readdata_register;
          when others =>
            Data_out(7 downto 0) <= X"FF";
        end case;
      else    --SBHE=0 => Signal bus is high enabled - 16bit
        case address_in is
          when "11" & X"E8" => --1000
            Data_out <= UART_Receiver_buffer_reg & X"FF";
            UART_Receiver_buffer_reg_read_sig <= '1';
          when "11" & X"E9" => --1001
            Data_out <= X"FFFF";
          when "11" & X"EA" => --1002
            Data_out(7 downto 0) <= UART_Divisor_low_byte_sig;
            Data_out(15 downto 8) <= UART_Divisor_high_byte_sig;
          when "11" & X"EB" => --1003
            Data_out(15 downto 8) <= UART_Divisor_high_byte_sig;
          when "11" & X"EC" => --1004
            Data_out(7 downto 0) <= Fifo_status_reg;
            Data_out(15 downto 8) <= Scratch_reg_sig;
          when "11" & X"ED" => --1005
            Data_out(15 downto 8) <= Scratch_reg_sig;
          when "11" & X"EE" => --1006
            Data_out(7 downto 0) <= I2C_reg_select_register_sig;
            Data_out(15 downto 8) <= I2C_readdata_register;
          when "11" & X"EF" => --1007
            Data_out(15 downto 8) <= I2C_readdata_register;
          when others =>
            Data_out <= X"FFFF";
        end case;
      end if;
    elsif write = '1' then ----------------------------Write to registers-------------------------
      reg_transmission_ok_sig <= '1';
      if SBHE = '1' then --SBHE=1 => Signal bus is not high enabled - 8bit
        case address_in is
          when "11" & X"E8" => --1000
            UART_Transmitter_holding_reg_sig <= Data_in(7 downto 0);
            UART_Transmitter_holding_reg_write_sig <= '1';
          when "11" & X"EA" => --1002
            UART_Divisor_low_byte_sig <= Data_in(7 downto 0);
          when "11" & X"EB" => --1003
            UART_Divisor_high_byte_sig <= Data_in(7 downto 0);
          when "11" & X"ED" => --1005
            Scratch_reg_sig <= Data_in(7 downto 0);
          when "11" & X"EE" => --1006
            I2C_reg_select_register_sig <= Data_in(7 downto 0);
          when "11" & X"EF" => --1007
            I2C_writedata_register_sig <= Data_in(7 downto 0);
            I2C_writedata_register_write_sig <= '1';
          when others => null;
        end case;
      else    --SBHE=0 => Signal bus is high enabled - 16bit
```

```
        case address_in is
          when "11" & X"E8" => --1000
            UART_Transmitter_holding_reg_sig <= Data_in(7 downto 0);
            UART_Transmitter_holding_reg_write_sig <= '1';
          when "11" & X"EA" => --1002
            UART_Divisor_low_byte_sig <= Data_in(7 downto 0);
            UART_Divisor_high_byte_sig <= Data_in(15 downto 8);
          when "11" & X"EB" => --1003
            UART_Divisor_high_byte_sig <= Data_in(15 downto 8);
          when "11" & X"ED" => --1005
            Scratch_reg_sig <= Data_in(15 downto 8);
          when "11" & X"EE" => --1006
            I2C_reg_select_register_sig <= Data_in(7 downto 0);
            I2C_writedata_register_sig <= Data_in(15 downto 8);
            I2C_writedata_register_write_sig <= '1';
          when "11" & X"EF" => --1007
            I2C_writedata_register_sig <= Data_in(15 downto 8);
            I2C_writedata_register_write_sig <= '1';
          when others => null;
        end case;
      end if;
    end if;
  end if;
end process;
end Behavioral;
```

B.27 IRQ_in.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Slave Node
-- Module Name:    IRQ_in - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Samples the interrupt segnals on the ISA bus
and keeps track
--                     of which interrupt lines that have been
pulled.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity IRQ_in is
    Port ( clk, reset: in std_logic;
           Data_from_lvds : in std_logic_vector(15 downto 0);
           IRQ_read_ok : out std_logic;
           IRQ_out : out std_logic;
           IRQ_count_ready : in std_logic;
           IRQ3, IRQ4, IRQ5, IRQ6, IRQ7, IRQ9, IRQ10, IRQ11,
IRQ12, IRQ13, IRQ14: in std_logic); --These signals are pulluped
end IRQ_in;

architecture Behavioral of IRQ_in is
signal sampling_done : std_logic;
Signal Prev_state_IRQ3, Prev_state_IRQ4, Prev_state_IRQ5,
Prev_state_IRQ6, Prev_state_IRQ7, Prev_state_IRQ9,
Prev_state_IRQ10, Prev_state_IRQ11, Prev_state_IRQ12,
Prev_state_IRQ13, Prev_state_IRQ14 :  std_logic;
--Signals stores the previous value of the IRQ-wire. The
--interrupt signal from 0 to 1 can then be detected by comparing
--with the present value.

Signal Signal_pulled_IRQ3, Signal_pulled_IRQ4,
Signal_pulled_IRQ5, Signal_pulled_IRQ6, Signal_pulled_IRQ7,
       Signal_pulled_IRQ9, Signal_pulled_IRQ10,
Signal_pulled_IRQ11, Signal_pulled_IRQ12, Signal_pulled_IRQ13,
       Signal_pulled_IRQ14 :  std_logic; --Signals stores if the
interruptline has been interupted or not.
begin

process(clk, reset)
begin
  if reset='1' then
    IRQ_read_ok <= '0';
        when "0100" => --IRQ4
          IRQ_out <= Signal_pulled_IRQ4; --IRQ_state sampled
          Signal_pulled_IRQ4 <= '0';
        when "0101" => --IRQ5
          IRQ_out <= Signal_pulled_IRQ5; --IRQ_state sampled
          Signal_pulled_IRQ5 <= '0';
        when "0110" => --IRQ6
          IRQ_out <= Signal_pulled_IRQ6; --IRQ_state sampled
          Signal_pulled_IRQ6 <= '0';
        when "0111" => --IRQ7
          IRQ_out <= Signal_pulled_IRQ7; --IRQ_state sampled
          Signal_pulled_IRQ7 <= '0';
        when "1001" => --IRQ9
          IRQ_out <= Signal_pulled_IRQ9; --IRQ_state sampled
          Signal_pulled_IRQ9 <= '0';
        when "1010" => --IRQ10
          IRQ_out <= Signal_pulled_IRQ10; --IRQ_state sampled
          Signal_pulled_IRQ10 <= '0';
        when "1011" => --IRQ11
          IRQ_out <= Signal_pulled_IRQ11; --IRQ_state sampled
          Signal_pulled_IRQ11 <= '0';
        when "1100" => --IRQ12
          IRQ_out <= Signal_pulled_IRQ12; --IRQ_state sampled
          Signal_pulled_IRQ12 <= '0';
        when "1101" => --IRQ13
          IRQ_out <= Signal_pulled_IRQ13; --IRQ_state sampled
          Signal_pulled_IRQ13 <= '0';
        when "1110" => --IRQ14
          IRQ_out <= Signal_pulled_IRQ14; --IRQ_state sampled
          Signal_pulled_IRQ14 <= '0';
        when others => IRQ_out <= '0';
      end case;
    elsif IRQ_count_ready = '0' then
      sampling_done <= '0';
      IRQ_read_ok <= '0';
      IRQ_out <= '0';
    end if;

    if Prev_state_IRQ3 = '0' and IRQ3 = '1' then
      Signal_pulled_IRQ3 <= '1';
    end if;
    if Prev_state_IRQ4 = '0' and IRQ4 = '1' then
      Signal_pulled_IRQ4 <= '1';
    end if;
    if Prev_state_IRQ5 = '0' and IRQ5 = '1' then
      Signal_pulled_IRQ5 <= '1';
    end if;
    if Prev_state_IRQ6 = '0' and IRQ6 = '1' then
      Signal_pulled_IRQ6 <= '1';
    end if;
    if Prev_state_IRQ7 = '0' and IRQ7 = '1' then
      Signal_pulled_IRQ7 <= '1';
    end if;
    if Prev state IRQ9 = '0' and IRQ9 = '1' then
```

```
    IRQ_out <= '0';

    Prev_state_IRQ3 <= '1';
    Prev_state_IRQ4 <= '1';
    Prev_state_IRQ5 <= '1';
    Prev_state_IRQ6 <= '1';
    Prev_state_IRQ7 <= '1';
    Prev_state_IRQ9 <= '1';
    Prev_state_IRQ10 <= '1';
    Prev_state_IRQ11 <= '1';
    Prev_state_IRQ12 <= '1';
    Prev_state_IRQ13 <= '1';
    Prev_state_IRQ14 <= '1';

    Signal_pulled_IRQ3 <= '0';
    Signal_pulled_IRQ4 <= '0';
    Signal_pulled_IRQ5 <= '0';
    Signal_pulled_IRQ6 <= '0';
    Signal_pulled_IRQ7 <= '0';
    Signal_pulled_IRQ9 <= '0';
    Signal_pulled_IRQ10 <= '0';
    Signal_pulled_IRQ11 <= '0';
    Signal_pulled_IRQ12 <= '0';
    Signal_pulled_IRQ13 <= '0';
    Signal_pulled_IRQ14 <= '0';

  elsif rising_edge(clk) then
    if IRQ_count_ready = '1' and sampling_done = '0' then
      sampling_done <= '1';
      IRQ_read_ok <= '1';
      case Data_from_lvds(3 downto 0) is --IRQ_state to sample
        when "0011" => --IRQ3
          IRQ_out <= Signal_pulled_IRQ3; --IRQ_state sampled
          Signal_pulled_IRQ3 <= '0';
          Signal_pulled_IRQ9 <= '1';
        end if;
        if Prev_state_IRQ10 = '0' and IRQ10 = '1' then
          Signal_pulled_IRQ10 <= '1';
        end if;
        if Prev_state_IRQ11 = '0' and IRQ11 = '1' then
          Signal_pulled_IRQ11 <= '1';
        end if;
        if Prev_state_IRQ12 = '0' and IRQ12 = '1' then
          Signal_pulled_IRQ12 <= '1';
        end if;
        if Prev_state_IRQ13 = '0' and IRQ13 = '1' then
          Signal_pulled_IRQ13 <= '1';
        end if;
        if Prev_state_IRQ14 = '0' and IRQ14 = '1' then
          Signal_pulled_IRQ14 <= '1';
        end if;

        Prev_state_IRQ3 <= IRQ3;
        Prev_state_IRQ4 <= IRQ4;
        Prev_state_IRQ5 <= IRQ5;
        Prev_state_IRQ6 <= IRQ6;
        Prev_state_IRQ7 <= IRQ7;
        Prev_state_IRQ9 <= IRQ9;
        Prev_state_IRQ10 <= IRQ10;
        Prev_state_IRQ11 <= IRQ11;
        Prev_state_IRQ12 <= IRQ12;
        Prev_state_IRQ13 <= IRQ13;
        Prev_state_IRQ14 <= IRQ14;

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

B.28 ISA_master.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Slave Node
-- Module Name:    ISA_master - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Acts as an ISA Master and sends and receives
--                 data from the slaves on the bus.
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ISA_master is
    Port ( SA : out std_logic_vector(9 downto 0);
--Address bus from ISA
           SD : inout std_logic_vector(15 downto 0) := (others => 'Z');
           BCLK_sync833_out : out std_logic;                --BCLK clock out 8.33 Mhz
           AEN : out std_logic;                             --AEN = '0' => no DMA
           SBHE_out : out std_logic;                        --ISA SBHE# out
           IORC : out std_logic;
           IOWC : out std_logic;
           BALE : out std_logic;
           IO16 : in std_logic;
           CHRDY : in std_logic;
           NOWS : in std_logic;

           clk : in std_logic;
           reset : in std_logic;
           address_in : in std_logic_vector(9 downto 0);    --Address bus from 'master_node_state_machine'
           data_in : in std_logic_vector(15 downto 0);      --Data bus from 'master_node_state_machine'
           data_out : out std_logic_vector(15 downto 0);    --Data bus to 'master_node_state_machine'
           read_data : in std_logic;                        --Read data command
           write_data : in std_logic;                       --Write data command
           SBHE_in : in std_logic;
           transmission_ok : out std_logic);
--signals that transmission to ISA device is completed
```

```
end ISA_master;

architecture Behavioral of ISA_master is
type state_type is (Groundstate_SRW0,  --Ground state
                    State_SW1, State_BALE_SW2, State_SW3, State_IOWC_SW4,
                    State_SW5, State_CHRDYnNOWS_SW6, State_SW7, State_delay8bit1_SW8, State_SW9,
                    State_delay8bit2_SW10, State_SW11, State_delay8n16bit3_SW12, State_Endfirst8bitcycle_SW13,
                    State_Beginnext8bitcycle_SW14, State_Endwritecycle_SW15, State_holddataonbus_SW16,
                    --Writestates above and Read states below
                    State_SR1, State_BALE_SR2, State_SR3, State_IORC_SR4,
                    State_SR5, State_CHRDYnNOWS_SR6, State_SR7, State_delay8bit1_SR8, State_SR9,
                    State_delay8bit2_SR10, State_SR11, State_delay8n16bit3_SR12, State_Endfirst8bitcycle_SR13,
                    State_Beginnext8bitcycle_SR14, State_Endreadcycle_SR15,
                    State_Waitforcommandreset_SRW20); --End state

signal state : state_type:= Groundstate_SRW0;
signal Data_in_latch : std_logic_vector(15 downto 0);
signal Address_in_latch : std_logic_vector(9 downto 0);
signal IO16_latched, SBHE_out_sig : std_logic;
signal Count_last_byte : std_logic;
signal ISA_timer_count : std_logic_vector(2 downto 0):= "000";
signal BCLK_833_in : std_logic:= '0';
begin
AEN <= '0';

ISA_timer: process(CLK, reset)
begin
  if reset = '1' then
    ISA_timer_count <= "000";
    BCLK_833_in <= '0';
  elsif CLK'event and CLK = '1' then
    if ISA_timer_count = "010" then  -- BCLK_833_in == CLK (50MHz) * 6 (= 8.3333 MHz)
      BCLK_833_in <= not BCLK_833_in;
      ISA_timer_count <= "000";
    else
      ISA_timer_count <= ISA_timer_count +1;
    end if;
  end if;
end process;

p0: process(CLK, reset)
begin
  if reset = '1' then
    SD <= (others => 'Z');
    SA <= (others => '1');
    BALE <= '0';
    IORC <= '1';
    IOWC <= '1';
    state <= Groundstate_SRW0;
    Count_last_byte <= '0';
    BCLK_sync833_out <= '0';
    transmission_ok <= '0';
  elsif CLK'event and CLK = '1' then   -- Sampling clock faster than 16 MHz if BCLK is 8 MHZ
    BCLK_sync833_out <= BCLK_833_in;
    case state is  --ISA-master_State machine
      when Groundstate_SRW0 =>  ---------------- GROUND STATE --------------------------------
        SD <= (others => 'Z');
        BALE <= '0';
        IORC <= '1';
        IOWC <= '1';
        Count_last_byte <= '0';
        Data_in_latch <= data_in;
        Address_in_latch <= address_in;
        SBHE_out_sig <= SBHE_in;
        if read_data = '1' then
          state <= State_SR1;   --Jumps to READ command states
        elsif write_data = '1' then
          state <= State_SW1;   --Jumps to WRITE command states
        end if;
      when State_SW1 =>   ------------------- STATES FOR WRITE COMMAND -----------------------------
        if BCLK_833_in = '1' then
          state <= State_BALE_SW2;
        end if;
      when State_BALE_SW2 =>  -- BALE is asserted as BCLK goes low
        if BCLK_833_in = '0' then
          SD <= Data_in_latch;
          SA <= Address_in_latch;
          SBHE_out <= SBHE_out_sig;
          BALE <= '1';
          state <= State_SW3;
        end if;
      when State_SW3 =>       -- BALE is deasserted to tell devices that the address is latched
        if BCLK_833_in = '1' then
          state <= State_IOWC_SW4;
          BALE <= '0';
        end if;
```

```
      when State_IOWC_SW4 =>  -- IOWC is asseted
        if BCLK_833_in = '0' then
          state <= State_SW5;
          IOWC <= '0';
        end if;
      when State_SW5 =>
        if BCLK_833_in = '1' then
          state <= State_CHRDYnNOWS_SW6;
        end if;
      when State_CHRDYnNOWS_SW6 =>
        if BCLK_833_in = '0' then
          if IO16 = '1' then --(8bit_device)
            if SBHE_out_sig = '0' then --(high address)
              if Count_last_byte = '0' and Address_in_latch(0)='0' then  --Even address & first byte
                if NOWS = '0' and CHRDY = '1' then -- end early
                  state <= State_Endfirst8bitcycle_SW13;
                else
                  state <= State_SW7;
                end if;
              else
                Count_last_byte <= '1';
                SD(7 downto 0) <= Data_in_latch(15 downto 8);   --Odd addressed data in high databus
                if NOWS = '0' and CHRDY = '1' then -- end early
                  state <= State_Endwritecycle_SW15;
                else
                  state <= State_SW7;
                end if;
              end if;
            else --(low address)
              Count_last_byte <= '1';
              if NOWS = '0' and CHRDY = '1' then -- end early
                state <= State_Endwritecycle_SW15;
              else
                state <= State_SW7;
              end if;
            end if;
          else --(16bit_device)
            Count_last_byte <= '1';
            if CHRDY = '1' then -- end early
              state <= State_Endwritecycle_SW15;
            else  --extend cycle
              state <= State_SW11;
            end if;
          end if;
        end if;
      when State_SW7 =>  --delay because 8 bit device transfer
        if BCLK_833_in = '1' then
          state <= State_delay8bit1_SW8;
        end if;
      when State_delay8bit1_SW8 =>
        if BCLK_833_in = '0' then
          if NOWS = '0' and CHRDY = '1' then   --Early end of bus cycle
            if Count_last_byte = '1' then
              state <= State_Endwritecycle_SW15;
            else
              state <= State_Endfirst8bitcycle_SW13;
            end if;
          else
            state <= State_SW9;
          end if;
        end if;
      when State_SW9 =>
        if BCLK_833_in = '1' then
          state <= State_delay8bit2_SW10;
        end if;
      when State_delay8bit2_SW10 =>
        if BCLK_833_in = '0' then
          if NOWS = '0' and CHRDY = '1' then   --Early end of bus cycle
            if Count_last_byte = '1' then
              state <= State_Endwritecycle_SW15;
            else
              state <= State_Endfirst8bitcycle_SW13;
            end if;
          else
            state <= State_SW11;
          end if;
        end if;
      when State_SW11 =>
        if BCLK_833_in = '1' then
          state <= State_delay8n16bit3_SW12;
        end if;
      when State_delay8n16bit3_SW12 =>
        if BCLK_833_in = '0' then
          if CHRDY = '1' then -- avsluta
            if Count_last_byte = '1' then
              state <= State_Endwritecycle_SW15;
            else
```

```
            state <= State_Endfirst8bitcycle_SW13;
          end if;
        else -- bromsa
          state <= State_SW11;
        end if;
      end if;
    when State_Endfirst8bitcycle_SW13 =>
      Count_last_byte <= '1';
      if BCLK_833_in = '1' then
        IOWC <= '1';
        state <= State_Beginnext8bitcycle_SW14;
      end if;
    when State_Beginnext8bitcycle_SW14 =>
      if BCLK_833_in = '0' then
        BALE <= '1';
        SD(7 downto 0) <= Data_in_latch(15 downto 8);
        SA(0) <= '1';                                   -- increase even address to address +1
        state <= State_SW3;
      end if;
    when State_Endwritecycle_SW15 =>
      if BCLK_833_in = '1' then
        IOWC <= '1';
        transmission_ok <= '0';
        state <= State_holddataonbus_SW16;
      end if;
    when State_holddataonbus_SW16 =>
      if BCLK_833_in = '0' then
        transmission_ok <= '1';
        state <= State_Waitforcommandreset_SRW20;
      end if;
    when State_SR1 =>   ------------------ STATES FOR READ COMMAND -----------------------------
      if BCLK_833_in = '1' then
        state <= State_BALE_SR2;
      end if;
    when State_BALE_SR2 =>
      if BCLK_833_in = '0' then
        SA <= Address_in_latch;
        SBHE_out <= SBHE_out_sig;
        BALE <= '1';
        state <= State_SR3;
      end if;
    when State_SR3 =>
      if BCLK_833_in = '1' then
        state <= State_IORC_SR4;
        BALE <= '0';
      end if;
    when State_IORC_SR4 =>
      if BCLK_833_in = '0' then
        state <= State_SR5;
        IORC <= '0';
      end if;
    when State_SR5 =>
      if BCLK_833_in = '1' then
        state <= State_CHRDYnNOWS_SR6;
      end if;
    when State_CHRDYnNOWS_SR6 =>
      if BCLK_833_in = '0' then
        IO16_latched <= IO16;
        if IO16 = '1' then --(8bit_device)
          if SBHE_out_sig = '0' then --(high address)
            if Count_last_byte = '0' and Address_in_latch(0)='0' then  --Even address & first byte
              if NOWS = '0' and CHRDY = '1' then -- end early
                state <= State_Endfirst8bitcycle_SR13;
              else
                state <= State_SR7;
              end if;
            else
              Count_last_byte <= '1';
              if NOWS = '0' and CHRDY = '1' then -- end early
                state <= State_Endreadcycle_SR15;
              else
                state <= State_SR7;
              end if;
            end if;
          else --(low address)
            Count_last_byte <= '1';
            if NOWS = '0' and CHRDY = '1' then -- end early
              state <= State_Endreadcycle_SR15;
            else
              state <= State_SR7;
            end if;
          end if;
        else --(16bit_device)
          Count_last_byte <= '1';
          if CHRDY = '1' then -- end early
            state <= State_Endreadcycle_SR15;
          else  --extend cycle
```

```
            state <= State_SR11;
          end if;
        end if;
      end if;
    when State_SR7 =>
      if BCLK_833_in = '1' then
        state <= State_delay8bit1_SR8;
      end if;
    when State_delay8bit1_SR8 =>
      if BCLK_833_in = '0' then
        if NOWS = '0' and CHRDY = '1' then -- Early end of bus cycle
          if Count_last_byte = '1' then
            state <= State_Endreadcycle_SR15;
          else
            state <= State_Endfirst8bitcycle_SR13;
          end if;
        else
          state <= State_SR9;
        end if;
      end if;
    when State_SR9 =>
      if BCLK_833_in = '1' then
        state <= State_delay8bit2_SR10;
      end if;
    when State_delay8bit2_SR10 =>
      if BCLK_833_in = '0' then
        if NOWS = '0' and CHRDY = '1' then -- Early end of bus cycle
          if Count_last_byte = '1' then
            state <= State_Endreadcycle_SR15;
          else
            state <= State_Endfirst8bitcycle_SR13;
          end if;
        else
          state <= State_SR11;
        end if;
      end if;
    when State_SR11 =>
      if BCLK_833_in = '1' then
        state <= State_delay8n16bit3_SR12;
      end if;
    when State_delay8n16bit3_SR12 =>
      if BCLK_833_in = '0' then
        if CHRDY = '1' then -- avsluta
          if Count_last_byte = '1' then
            state <= State_Endreadcycle_SR15;
          else
            state <= State_Endfirst8bitcycle_SR13;
          end if;
        else -- bromsa
          state <= State_SR11;
        end if;
      end if;
    when State_Endfirst8bitcycle_SR13 =>
      Count_last_byte <= '1';
      if BCLK_833_in = '1' then
        IORC <= '1';
        Data_out(7 downto 0) <= SD(7 downto 0);
        state <= State_Beginnext8bitcycle_SR14;
      end if;
    when State_Beginnext8bitcycle_SR14 =>
      if BCLK_833_in = '0' then
        BALE <= '1';
        SD <= (others => 'Z');
        SA(0) <= '1';                                   -- increase even address to address +1
        state <= State_SR3;
      end if;
    when State_Endreadcycle_SR15 =>
      if BCLK_833_in = '1' then
        IORC <= '1';
        transmission_ok <= '0';
        state <= State_Waitforcommandreset_SRW20;
        if IO16_latched = '1' then --(8bit)
          if SBHE_out_sig = '0' then --(high address)
            if Address_in_latch(0) = '0' then --(was even, SD is now odd (even+1))
              Data_out(15 downto 8) <= SD(7 downto 0);
            else
              Data_out(15 downto 8) <= SD(7 downto 0);
            end if;
          else
            Data_out(7 downto 0) <= SD(7 downto 0);
          end if;
        else --(16bit)
          Data_out <= SD; --whole bus
        end if;
      end if;
    when State_Waitforcommandreset_SRW20 =>   ------------------------ END STATE --------------------------------
      if read_data = '0' and write_data = '0' then
```

```
          Count_last_byte <= '0';
          transmission_ok <= '0';
          state <= Groundstate_SRW0;
        else
          transmission_ok <= '1';
        end if;
      when others =>
        state <= State_Waitforcommandreset_SRW20;  -- Takes care of undefined states
    end case;
  end if;
end process;
end Behavioral;
```

B.29 ISA_master_input_flipflop.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Slave Node
-- Module Name:    ISA_master_input_flipflop - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Generates flip-flops that samples the
--                 asynchronous signals from the ISA bus.
--                 Result is read by ISA_master.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ISA_master_input_flipflop is
    Port ( clk : in std_logic;
           IO16 : in std_logic;
           CHRDY : in std_logic;
           NOWS : in std_logic;
           IO16_out : out std_logic;
           CHRDY_out : out std_logic;
           NOWS_out : out std_logic);

end ISA_master_input_flipflop;

architecture Behavioral of ISA_master_input_flipflop is
begin

Dvippa: process(clk)  --All input signals (that a state jump
--                      depends upon) has to to be syncronized
begin
  if rising_edge(clk) then
    IO16_out <= IO16;
    CHRDY_out <= CHRDY;
    NOWS_out <= NOWS;
  end if;
end process;

end Behavioral;
```

B.30 ISA_master_termination.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Slave Node
-- Module Name:    ISA_master_termination - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Handles the termination of some unuses ISA
bus wires.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity ISA_master_termination is
    Port ( DMA_ack : inout std_logic_vector(7 downto 0);
           SA : inout std_logic_vector(19 downto 10);
           reset_ISA_dev : inout std_logic;
           MRDC : inout std_logic;
           MWTC : inout std_logic);
end ISA_master_termination;

architecture Behavioral of ISA_master_termination is

begin
  DMA_ack  <= (others => 'H');   --use pullUP on io pads in
                                 --constraints file
  reset_ISA_dev <= 'L';          --use pullDOWN on io pads in
                                 --constraints file
  MRDC <= 'H';                   --use pullUP on io pads in
                                 --constraints file
  MWTC <= 'H';
  SA <= (others => 'Z');         --use pullUP on io pads in
                                 --constraints file
end Behavioral;
```

B.31 Slave_node_state_machine.vhd

```
----------------------------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Slave Node
-- Module Name:    Slave_node_state_machine - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    The main state machine in the slave node. Handles the
--                 communication between modules.
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Slave_node_state_machine is
    Port ( clk : in std_logic;
           reset : in std_logic;
           lvds_transmission_bad : in std_logic;
           lvds_transmission_ok : in std_logic;
           dev_transmission_ok : in std_logic;
           reg_transmission_ok : in std_logic;
           IRQ_read_ok : in std_logic;
           IRQ_count_ready : out std_logic;
           IRQ_sampled : in std_logic;
           read1_write0 : in std_logic;
           lvds_out_ready : in std_logic;
           slave_transmission_timeout : in std_logic;
           reset_slave_transmission_timer : out std_logic;
           lvds_transmitt_enable : out std_logic;
           dev_read : out std_logic;
           dev_write : out std_logic;
           reg_read : out std_logic;
           reg_write : out std_logic;
           lvds_data_send : out std_logic;
           lvds_receive_reset : out std_logic;

           read1_write0_out : out std_logic;
           SBHE_out : out std_logic;
           Data_from_ISA_in : in std_logic_vector(15 downto 0);
           Data_from_reg_in : in std_logic_vector(15 downto 0);
           infobits_from_LVDS_in : in std_logic_vector(3 downto 0);
           Data_out : out std_logic_vector(15 downto 0);
           Address_out : out std_logic_vector(9 downto 0);
           Addres_from_lvds_in : in std_logic_vector(9 downto 0);
           info_bits_out : out std_logic_vector(3 downto 0));

end Slave_node_state_machine;

architecture Behavioral of Slave_node_state_machine is
type state_type is (Wait_for_lvds_receive_S0, Wait_for_device_receive_S1, LVDS_send_S2,
                    LVDS_ack_S3, End_of_lvds_cycle_S4, Device_command_reset_S5);
signal State : state_type;
begin
info_bits_out <= "0000";
Address_out <= Addres_from_lvds_in;
read1_write0_out <= '0';
SBHE_out <= '0';

Mealy_syncout:process(clk,reset)  -- Mealy Statemachine with synchronous outputs
begin
  if reset='1' then
    State <= Wait_for_lvds_receive_S0;
    dev_read <= '0';
    dev_write <= '0';
    reg_read <= '0';
    reg_write <= '0';
    reset_slave_transmission_timer <= '1';
    lvds_transmitt_enable <= '0';
    lvds_data_send <= '0';
    lvds_receive_reset <= '1';
    IRQ_count_ready <= '0';
    Data_out <= X"0000";

  elsif rising_edge(clk) then
    case State is
    when Wait_for_lvds_receive_S0=>            --Wait for data and command from lvds module
      reset_slave_transmission_timer <= '1';
      lvds_data_send <= '0';
```

```
      if lvds_transmission_bad ='1' then       --If bad transmission the receive module is reset and a new message is waited for
        lvds_receive_reset <= '1';
      elsif lvds_transmission_ok = '1' then    --Data is received and passed the error check
        if infobits_from_LVDS_in(2) = '1' then --interrupt request bit
          State <= Wait_for_device_receive_S1;
          IRQ_count_ready <= '1';
        elsif Addres_from_lvds_in(9 downto 3) = "1111101" then      --3E8 to 3EF = internal registers
          State <= Wait_for_device_receive_S1;
          if read1_write0 = '1' then
            reg_read <= '1';
          else
            reg_write <= '1';
          end if;
        else       -- read or write command is directed to the isa bus connected to this slave node
          State <= Wait_for_device_receive_S1;
          if read1_write0 = '1' then
            dev_read <= '1';
          else
            dev_write <= '1';
          end if;
        end if;
      else
        lvds_receive_reset <= '0';
      end if;
    when Wait_for_device_receive_S1=>          --Wait for register, device or interupt module to finish transmission
      reset_slave_transmission_timer <= '0';
      if reg_transmission_ok = '1' then        --Register command is executed
        State <= LVDS_send_S2;
        Data_out <= Data_from_reg_in;
        lvds_transmitt_enable <= '1';
      elsif dev_transmission_ok = '1' then     --Device command is executed
        State <= LVDS_send_S2;
        Data_out <= Data_from_ISA_in;
        lvds_transmitt_enable <= '1';
      elsif IRQ_read_ok = '1' then             --IQR command (polling) is executed
        State <= LVDS_send_S2;
        Data_out <= X"000" & "000" & IRQ_sampled;
        lvds_transmitt_enable <= '1';
      elsif slave_transmission_timeout = '1' then    --Timeout - NO command is executed in specified time
        State <= Device_command_reset_S5;
        IRQ_count_ready <= '0';
        reg_read <= '0';
        reg_write <= '0';
        dev_read <= '0';
        dev_write <= '0';
        lvds_receive_reset <= '1';
      end if;
    when LVDS_send_S2=>              --Send reply to master node via lvds
      State <= LVDS_ack_S3;
      lvds_data_send <= '1';
      reset_slave_transmission_timer <= '1';
    when LVDS_ack_S3=>               --lvds module acknowledges the command by setting lvds_out_ready line to '0'
      if lvds_out_ready = '0' then
        State <= End_of_lvds_cycle_S4;
        lvds_data_send <= '0';
        IRQ_count_ready <= '0';
        reg_read <= '0';
        reg_write <= '0';
        dev_read <= '0';
        dev_write <= '0';
      end if;
    when End_of_lvds_cycle_S4=>      --Wait for the lvds module to completing the transmission
      if lvds_out_ready = '1' then
        State <= Device_command_reset_S5;
        lvds_transmitt_enable <= '0';
        lvds_receive_reset <= '1';
      end if;
    when Device_command_reset_S5=>  --Wait for transmission =ok command to reset
      if dev_transmission_ok = '0' and reg_transmission_ok = '0' and IRQ_read_ok ='0' then
        reset_slave_transmission_timer <= '1';
        State <= Wait_for_lvds_receive_S0;
      end if;
    when others => State <= Wait_for_lvds_receive_S0;
  end case;
  end if;
end process;

end Behavioral;
```

B.32 Slave_transm_timer.vhd

```
----------------------------------------------------------------
-- Company: Hectronic AB
-- Engineer: Johan Johansson
--
-- Design Name:    Slave Node
-- Module Name:    slave_transm_timer - Behavioral
-- Project Name:   Distributed ISA
-- Target Device:  Xilinx - Spartan 3
-- Tool versions:  Xilinx - ISE WebPACK 7.1i
-- Description:    Generates a timeout signal when the ISA
--                 device on the slave node has held CHRDY too
--                 long. This will return X"FFFF" to the master
--                 node.
--
-- Revision:       14
-- Revision date:  20 June 2005
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity slave_transm_timer is
    Port ( clk, reset : in std_logic;
           slave_transmission_timout : out std_logic);
end slave_transm_timer;
```

```
architecture Behavioral of slave_transm_timer is
signal counter : std_logic_vector(15 downto 0);
begin
--Timeout at 15 us
--Vid MHZ clk -----Count to
--10               150 = X"0096"
--25               375 = X"0177"
--50               750 = X"02EE"
--100              1500 = X"05DC"
--200              3000 = X"0BB8"

  timer:process(clk, reset)
  begin
    if reset = '1' then
      slave_transmission_timout <= '0';
      counter <= (others => '0');
    elsif clk'event and clk ='1' then
      if counter <= X"0052" then   --X"0019" && clk = 50 MHz =>
                                   --0,54 us
        counter <= counter + 1;    --X"0032" && clk = 50 MHz =>
                                   --1,0 us
      else                         --X"0052" && clk = 50 MHz =>
                                   --1,65 us
        slave_transmission_timout <= '1';
      end if;
    end if;
  end process;
end Behavioral;
```

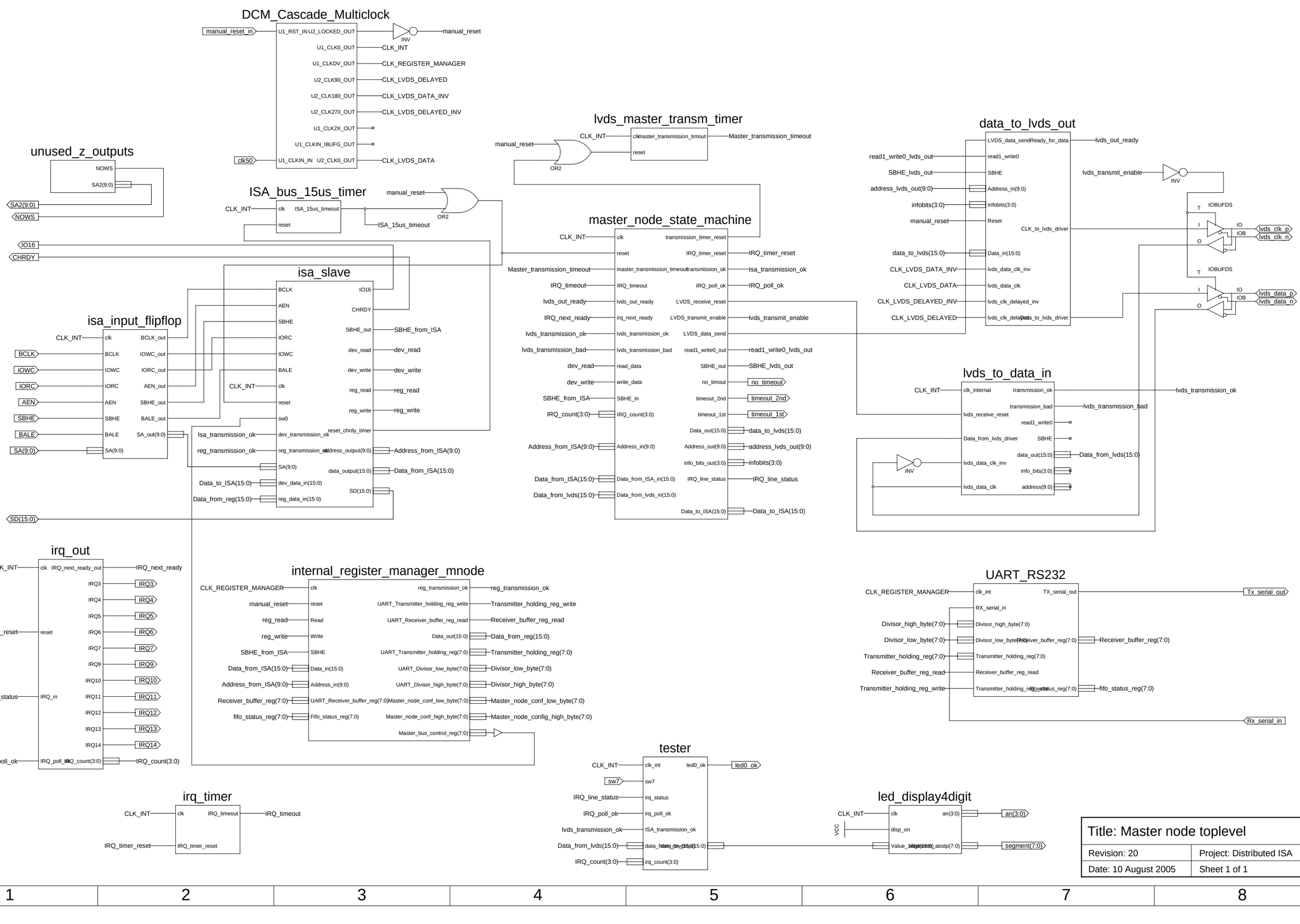
Appendix C – Schematic

C.1 Master_Node_top-level

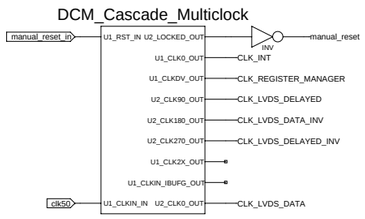
(See the A3 printout on the following page)

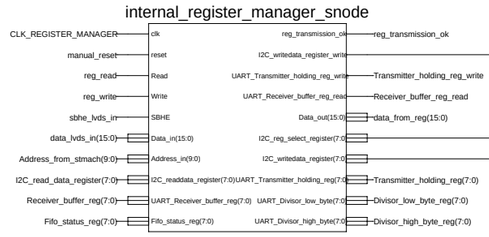
C.2 Slave_Node_top-level

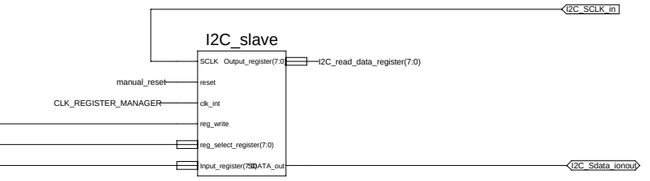
(See the A3 printout on the following page)

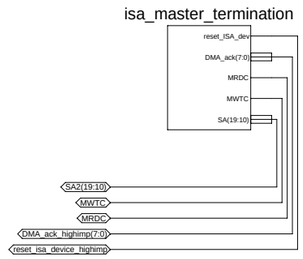


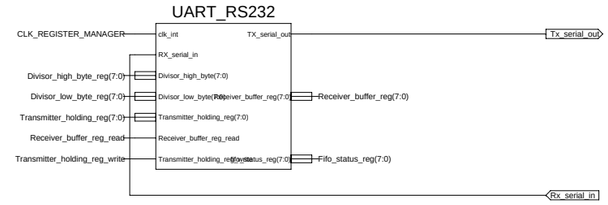
Title: Master node toplevel	
Revision: 20	Project: Distributed ISA
Date: 10 August 2005	Sheet 1 of 1

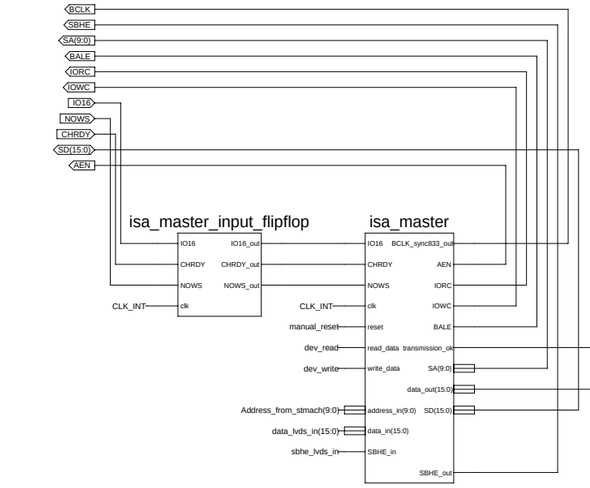


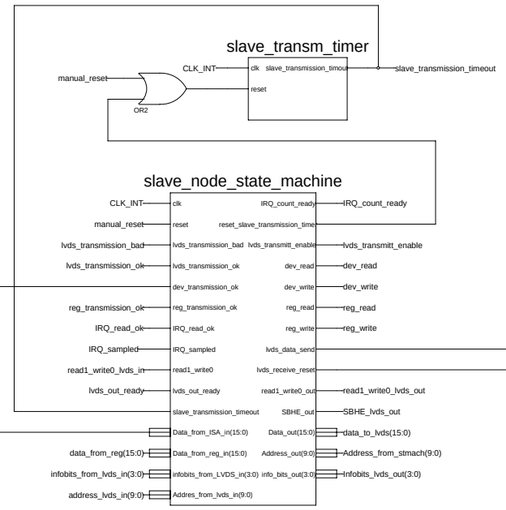


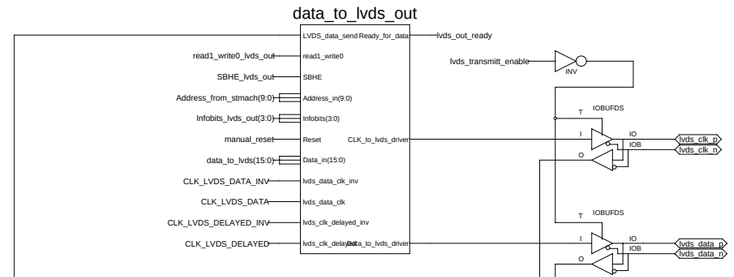


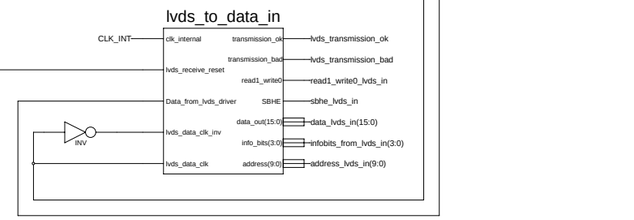


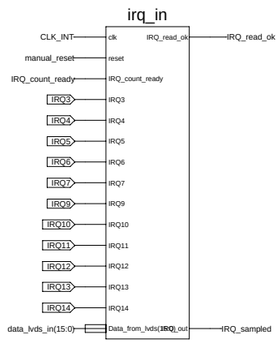


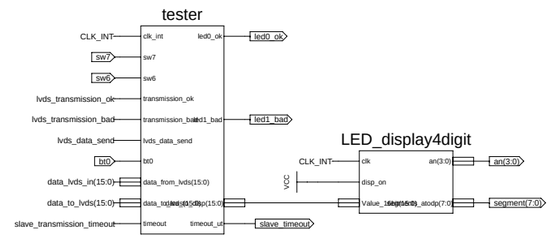






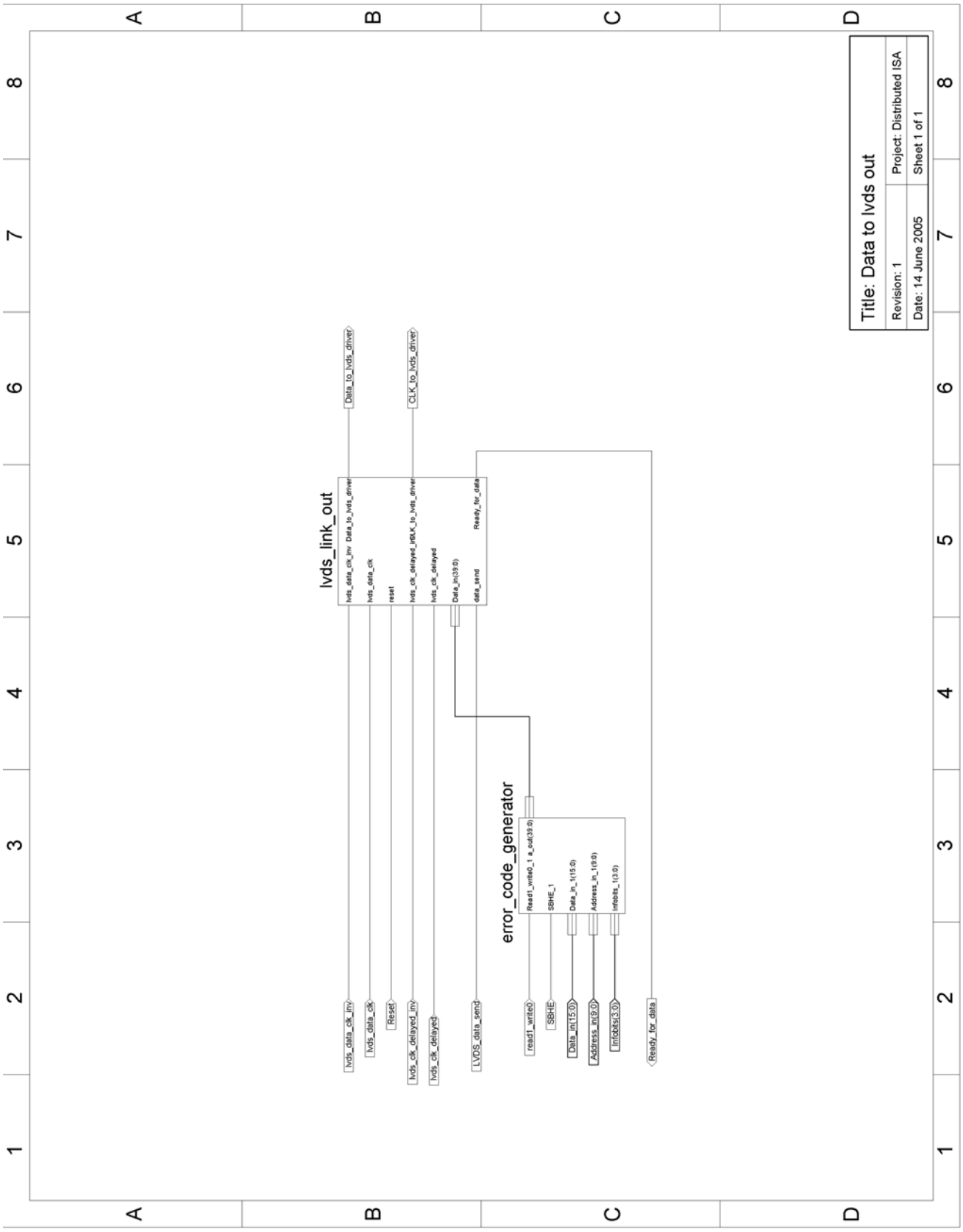




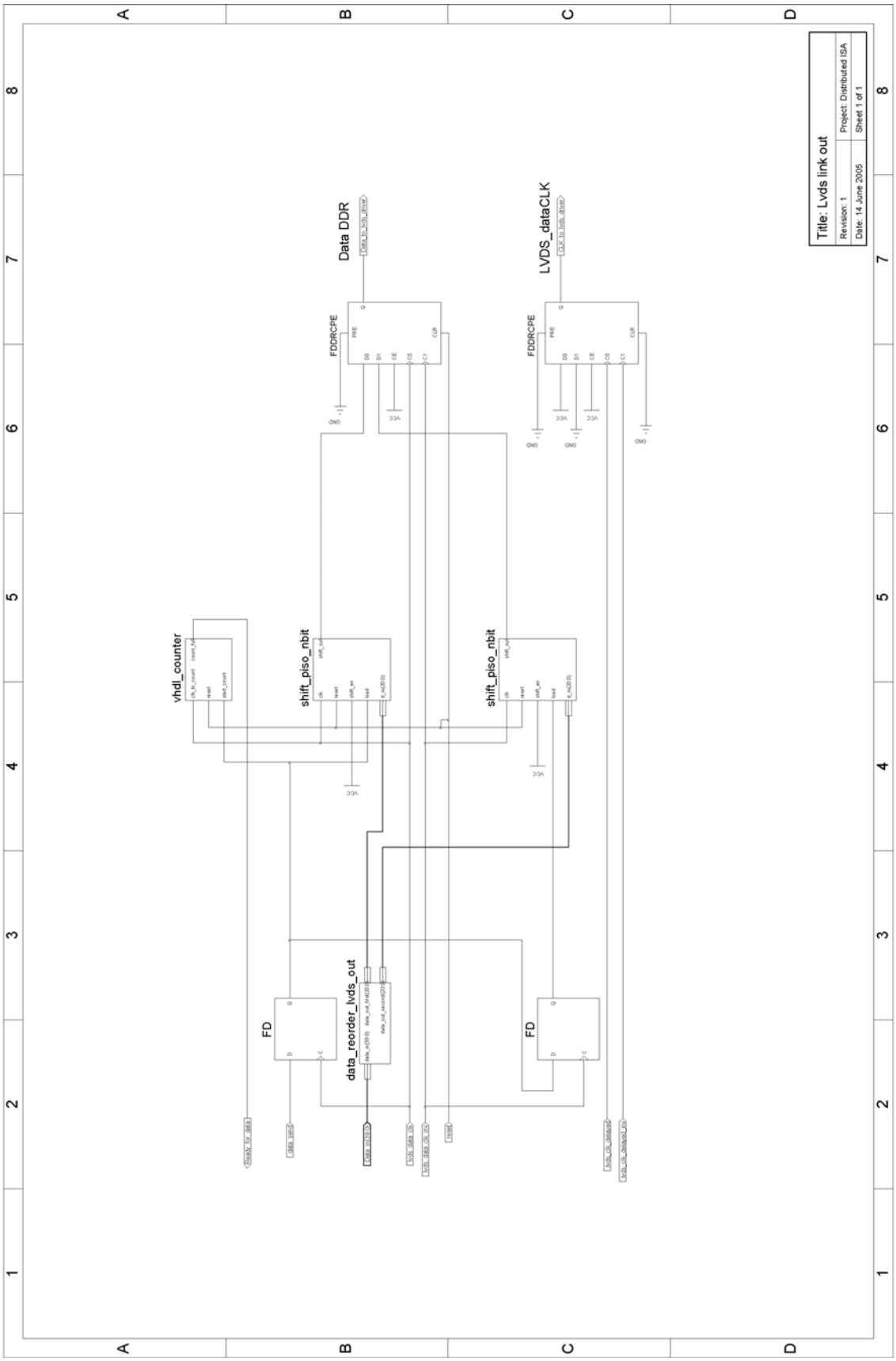


Title: Slave node toplevel	
Revision: 20	Project: Distributed ISA
Date: 10 August 2005	Sheet 1 of 1

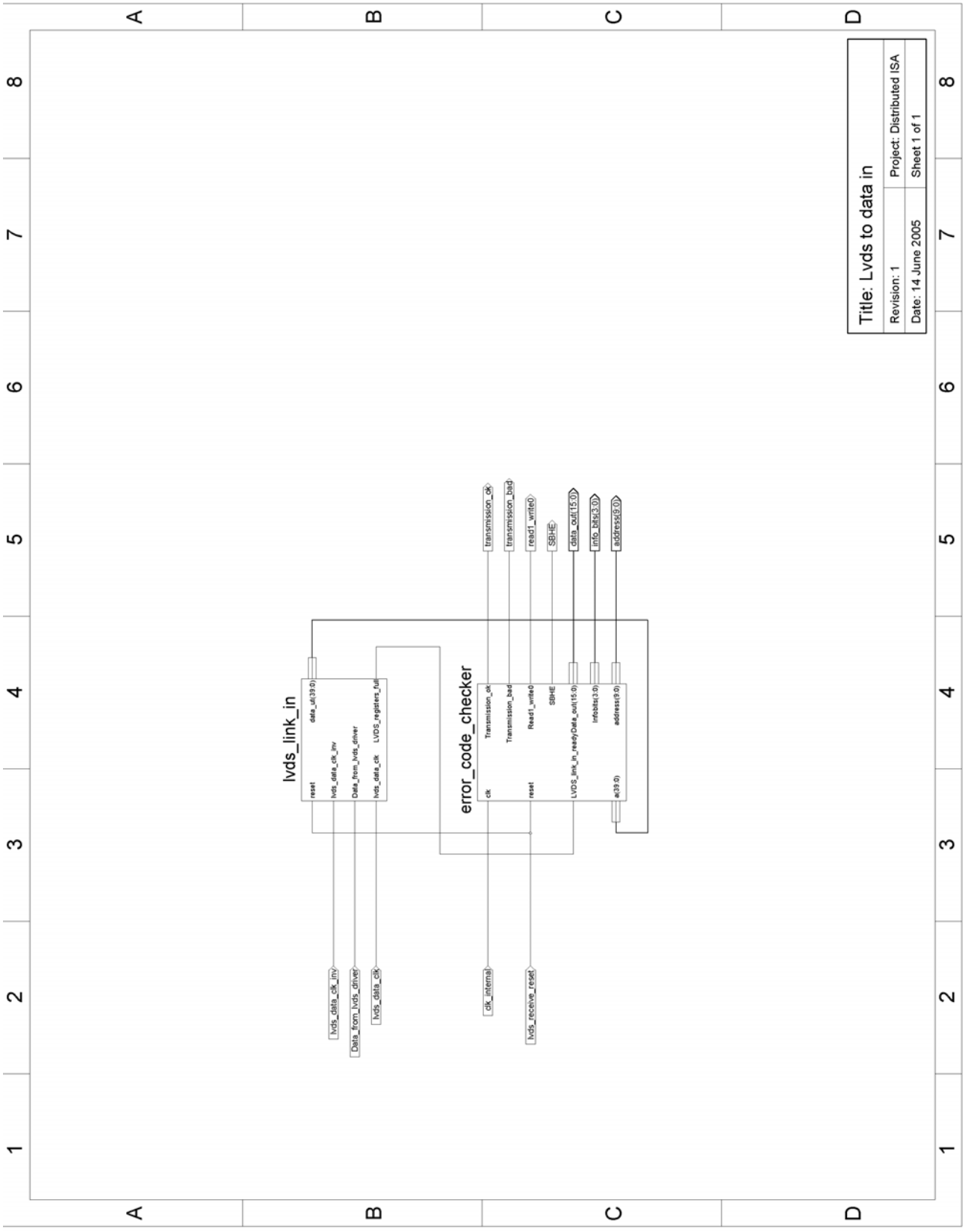
C.3 Data_to_LVDS_out.sch



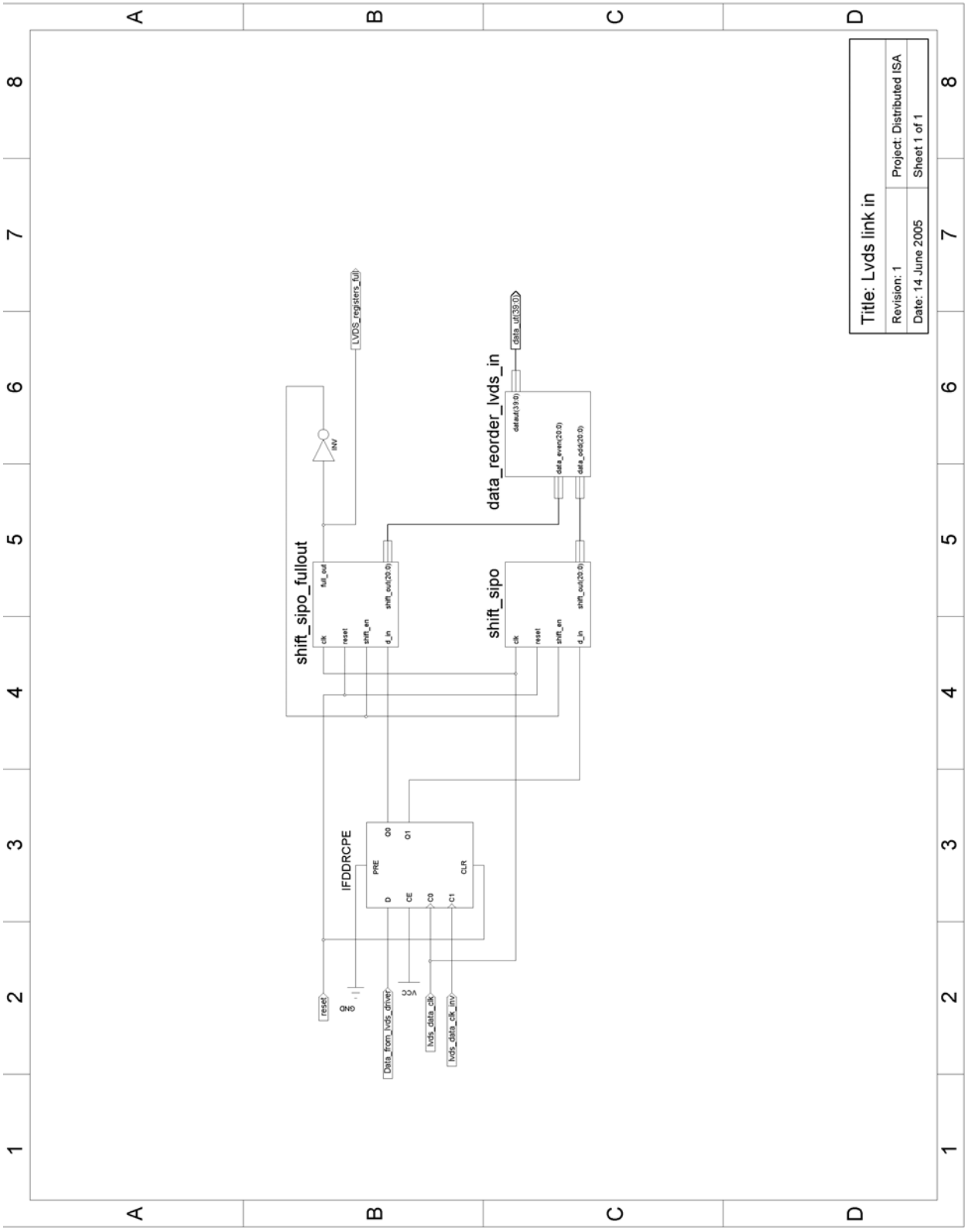
C.4 LVDS_link_out.sch



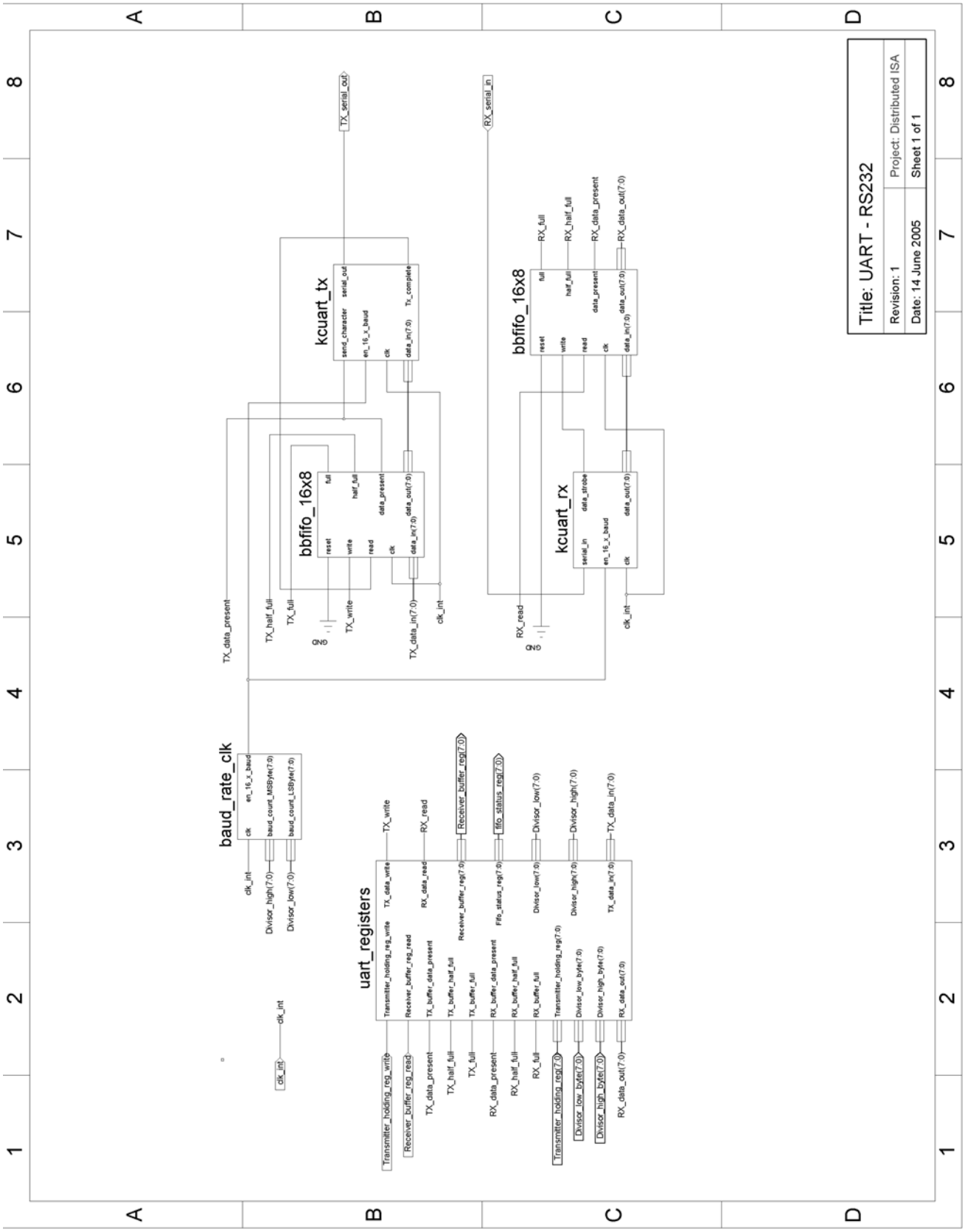
C.5 LVDS_to_data_in.sch



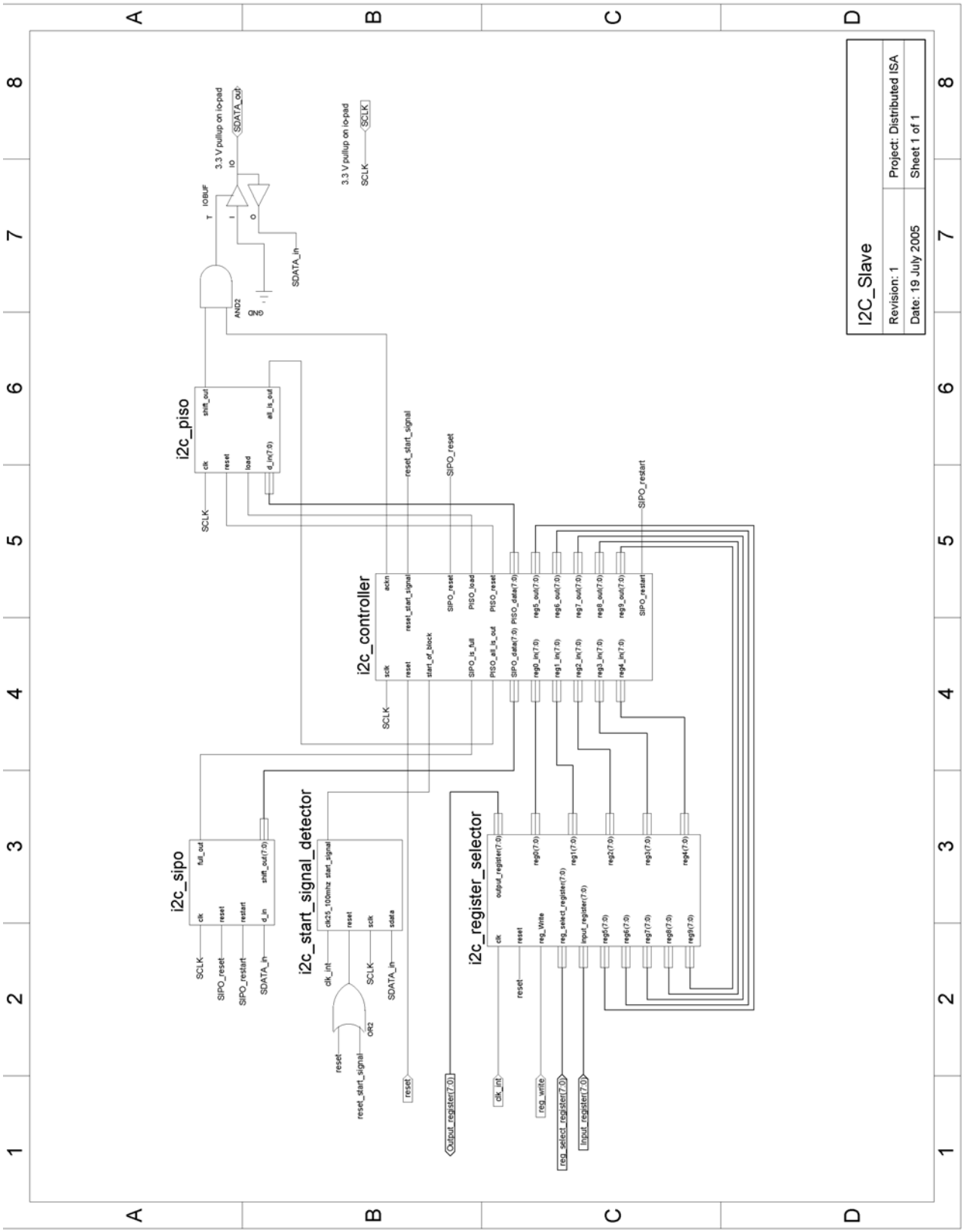
C.6 LVDS_link_in.sch



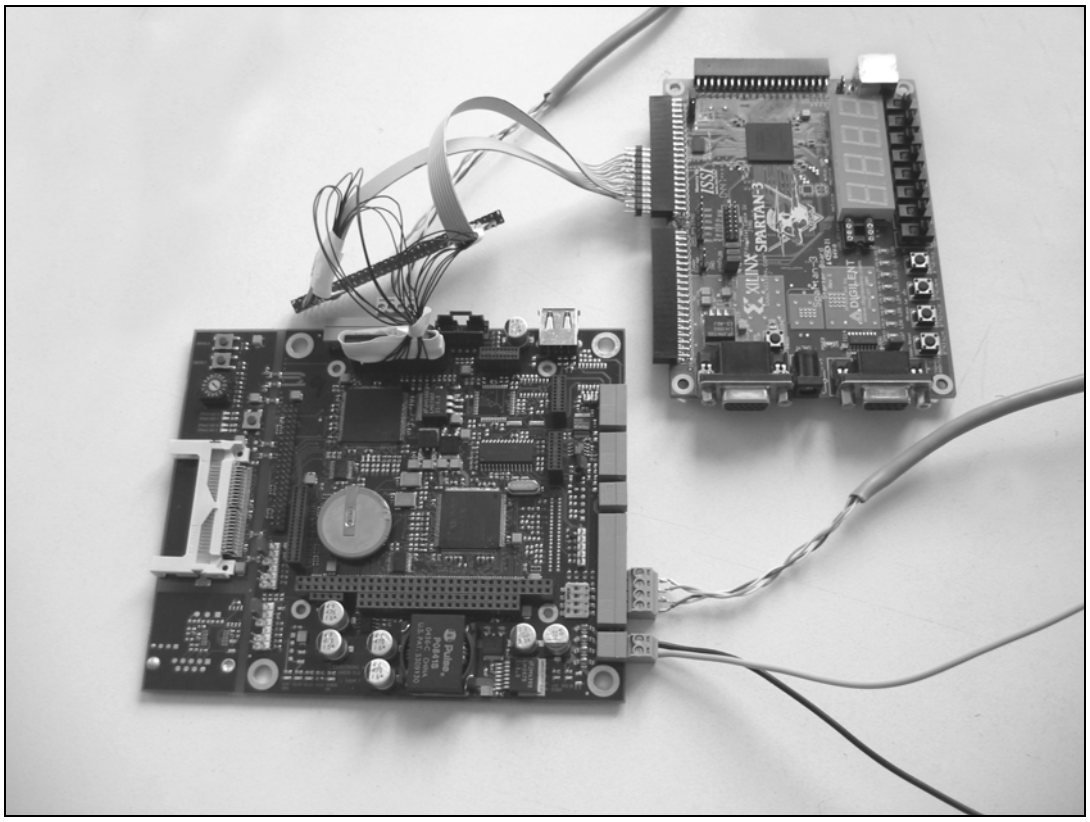
C.7 UART_RS232.sch



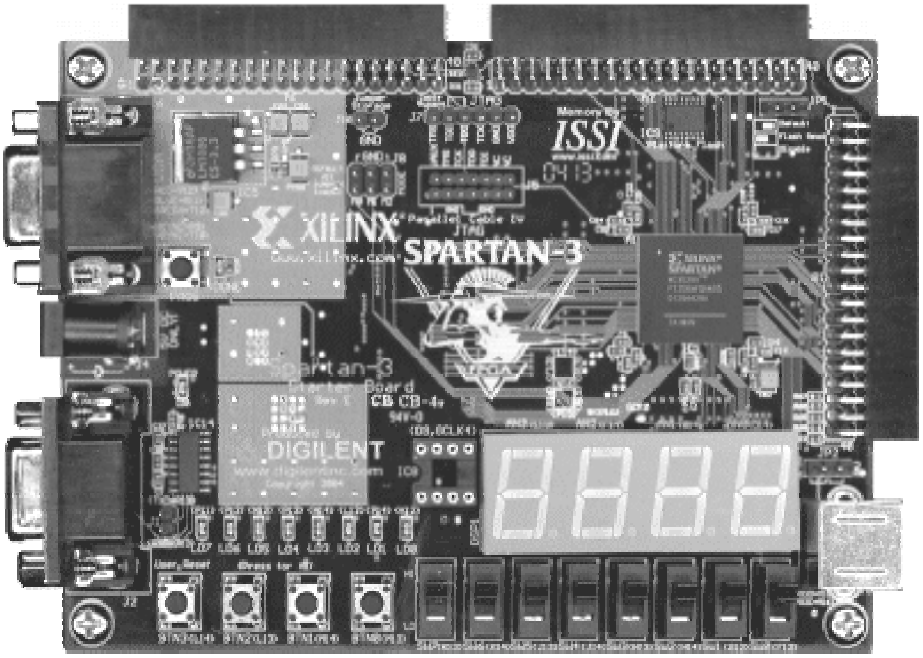
C.8 I2C_slave.sch



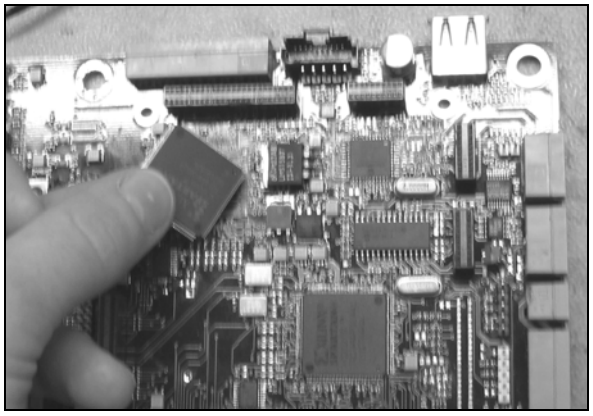
Appendix D – Pictures



A Spartan 3 starter kit board reading and writing to I/O registers connected to one slave node FPGA on the H4070 board. For debugging purpose only.



The Spartan 3 starter kit board.



Left: A 144-pin Spartan 3 FPGA put in place on the H4070 board. The FPGA was later soldered by me as well.
Right: A 144-pin FPGA chip (Spartan 2).

Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
BALE	Bus Address Latch Enable
BCLK	Bus Clock
BLVDS	Bus LVDS
BRAM	Block RAM
CHRDY	Channel Ready
CLB	Configurable Logic Blocks
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DCM	Digital Clock Manager
DMA	Direct Memory Access
ECL	Emitter Coupled Logic
EEPROM	Electronically Erasable Programmable Read-Only Memory
GCLK	Global Clock
FIFO	First In First Out
FPGA	Field Programmable Gate Array
I/O	Input / Output
I2C	Inter IC Communication
IC	Integrated Circuit
IO16	Input Output 16-bit
IORC	Input Output Read Command
IOWC	Input Output Write Command
IRQ	Interrupt Request
ISA	Industry Standard Architecture
JTAG	Programming interface that stands for Joint Test Action Group
SD	System Data
LED	Light Emitting Diode
LUT	Look-Up Table
LVDS	Low Voltage Differential Signalling
M-LVDS	Multipoint LVDS
MUX	Multiplexer
NOWS	No Wait State
OSI	Open System Interconnection
PCI	Peripheral Component Interconnect
PECL	Positive Emitter Coupled Logic
PLA	Programmable Logic Arrays
PLL	Phase Locked Loop
PoE	Power Over Ethernet
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
RST	Reset
RX	Receive
SA	System Address
SBHE	Signal Bus High Enable
TX	Transmit
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit, a type of digital logic circuit.
VLSI	Very-Large-Scale Integration
VME	Versa Module Eurocard

Acknowledgements

I would like to express my dearest gratitude to a few people for their support and assistance while working on this project. First and foremost, I would like to thank my scientific reviewer Leif Gustafsson for his invaluable assistance, insight and expertise in the subject area. I would also like to thank him for his patience and support reviewing this report.
Special thanks goes also to my great supervisors and encouraging engineers at Hectronic AB:

Erik Jansson for supervising this project from the start with enthusiasm, expertise and encouragement.

Lennart Nyström for his superior H4070 board design and never decreasing support.

Mats Arnlund for his expertise and encouragement.

Lars Hägglund for always being at hand giving invaluable unquestionable support whenever needed.

I would also like to thank the rest of the Hectronic staff that with smiles, jokes and debates forms the great working atmosphere at the company.

Index

References

[1] SearchNetworking.com, (2005). OSI (Open Systems Interconnection), http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci212725,00.html, 1 July 2005.
[2] Fairchild Semiconductor (2005), Fairchild Semiconductor Backplane Designer's Guide, http://www.fairchildsemi.com/products/interface/backplane.html, 1 July 2005.
[3] Rhys Haden, Data Network Resource (2002), Data Encoding Techniques, http://www.rhyshaden.com/encoding.htm, 1 July 2005.
[4] Godred Fairhurst (2001), Manchester Encoding, http://www.erg.abdn.ac.uk/users/gorry/course/phy-pages/man.html, Department of Engineering, University of Aberdeen, U.K., 1 July 2005.
[5] Beth Cohen, Debbie Deutsch, Wi-Fi planet (2003), Power over Ethernet - Ready to Power On?, http://www.wi-fiplanet.com/tutorials/article.php/2208781, 1 July 2005.
[6] Oliver Brosch, (2003). Introduction to FPGA Processors, http://www-li5.ti.uni-mannheim.de/fpga/group/intro.html, 1 July 2005.
[7] Xilinx, (2005). Spartan-3 FPGA Family: Complete Data Sheet - DS099. http://www.xilinx.com/bvdocs/publications/ds099.pdf, 1 July 2005.
[8] Xilinx, (2003). Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs - XAPP462 (v1.0). http://www.xilinx.com/bvdocs/appnotes/xapp462.pdf, 1 July 2005.
[9] Leif Gustafsson, (2005). Personal communication, Institutionen för strålningsvetenskap, Uppsala University.
[10] Xilinx, (2005). Spartan-3 FPGA Family: Complete Data Sheet - DS099.
[11] Nick Sawyer, (2002). High-Speed Data Serialization and Deserialization (840 Mb/s LVDS) - XAPP265 (1.3). http://www.xilinx.com/bvdocs/appnotes/xapp265.pdf, 1 July 2005
[12] Xilinx, (2005). Using Dedicated Multiplexers in Spartan-3 Generation FPGAs - XAPP466 (v1.1). http://www.xilinx.com/bvdocs/appnotes/xapp466.pdf, 1 July 2005.
[13] Brian Von Herzen, Jon Brunetti, (2001). Multi-Channel 622 Mb/s LVDS Data Transfer for Virtex-E Devices - XAPP233 (v1.2). http://www.xilinx.com/bvdocs/appnotes/xapp233.pdf, 1 July 2005.
[14] Stefan Sjöholm, Lennart Lindh, (2003). VHDL för konstruktion. Fjärde upplagan, Studentlitteratur, Lund.
[15] Xilinx, (2005). ISE 7.1i Development System Reference Guide, http://toolbox.xilinx.com/docsan/xilinx7/books/docs/dev/dev.pdf, 1 July 2005.
[16] Corelis, (2005). Intuitive and high-performance JTAG tools, www.jtag.org, 1 July 2005
[17] Tom Shanley, Don Andrersson (2001). ISA System Architecture, Third edition, Mindshare, Inc, USA.
[18] John Goldie (2005). The Many Flavors of LVDS, http://www.national.com/nationaledge/feb02/flavors.html, 1 July 2005.
[19] Fairchild semiconductors (2005), LVDS fundamentals, http://www.fairchildsemi.com/an/AN/AN-5017.pdf, 1 July 2005.
[20] Xilinx, (1999). Virtex-E High Performance Differential Solutions: Low Voltage Differential Signalling (LVDS), http://www.xilinx.com/products/virtex/techtopic/lvds.pdf, 1 July 2005.
[21] Leroy Davis (2005). LVDS Bus, http://www.interfacebus.com/Design_Connector_LVDS.html, 1 July 2005.
[22] Xilinx, (1999). Virtex-E High Performance Differential Solutions: Low Voltage Differential Signalling (LVDS), http://www.xilinx.com/products/virtex/techtopic/lvds.pdf, 1 July 2005.
[23] Benchmarq Microelectronics inc. (1998), System management bus specification.
[24] Andreas Berg, (2005), Personal communication, Field Application Engineer, Memec Insight, Sundbyberg, Sweden.