AGH University of Science and Technology

Project Report

FrontEnd Electronics(P11)

Author:
Prasoon Ambalathankandy

Supervisor:
Prof.Marek Idzik

January 22, 2013

Report

Prasoon Ambalathankandy

January 22, 2013

Contents

List of Figures

List of Tables

Chapter 1

Test & Characterization of a pipelined ADC

1.1 Introduction

A power scalable 10-bit pipeline ADC developed for the luminosity detector at the future International Linear Collider[3] was thoroughly tested and characterized. The prototype ASIC is fabricated in 0.35μ CMOS technology. ASIC containing six ADC versions, baseline ADC version and slightly different versions (sampling switches, CMFB type, etc.) were tested. After the basic functionality test with a DC input signal, the complete measurements of ADC static & dynamic parameters were performed using a dedicated FPGA based test setup. The ADC static performance is quantified with INL and DNL measurements. The dynamic parameters are quantified with SINAD, THD, SNHR & SFDR measurements.

1.2 Understanding ADC Definitions and Specifications

ADCs convert an analog signal input to a digital output code. ADC measurements deviate from the ideal due to process variations in integrated circuits (ICs) and various sources of inaccuracies in the analog-to-digital conversion process. ADC performance specifications[1] quantify errors that are caused by the ADC itself. It is very essential to understand the specifications hence a gist of useful specification is listed here for quick review. ADC performance specifications are generally categorized in two ways: dc accuracy and dynamic performance. Generally, most applications use ADCs to measure a relatively static, dc-like signal (e.g., a temperature sensor or strain gauge voltage) or a dynamic signal (e.g., processing of a voice signal or tone detection). The application determines which specifications the designer will consider the most important.

1.3 DC Specifications

The DC specifications for the A/D converter tell the designer how the device performs for steady-state analog inputs. These specifications are particularly important in instrumentation applications where the A/D converter is used to measure slowly varying physical events such as temperature, pressure or weight.

1.3.1 ADC Transfer function

The transfer function of an ADC is a plot of the voltage input to the ADC versus the code's output of the ADC.Such a plot is not continuous but is a plot of 2^N codes, where N is the ADC's resolution. If the codes were to be connected by lines (usually at code transition boundaries), the ideal transfer function would be a straight line. A line drawn through the points would begin at the origin of the plot, and the slope of the plot for each of the supplied ADC would be the same as shown in Fig 1.1.

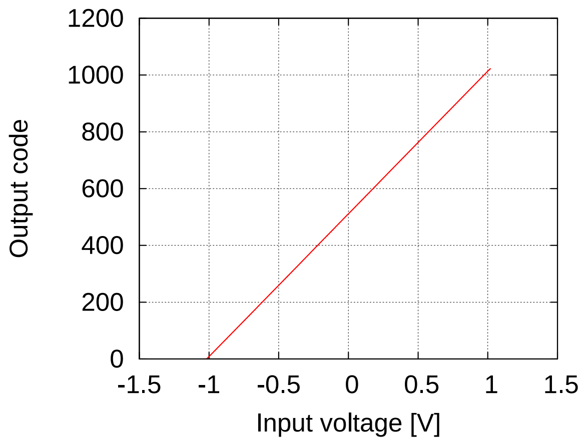


Figure 1.1: Transfer Function

The Transfer function in Fig 1.1 was calculated by ramping input voltage from -1V to 1V.

1.3.2 Differential Non-Linearity

In the ideal A/D converter transfer function, each code has a uniform width. That is, the difference in analog input voltage is constant from one code transition point to the next. Differential non-linearity, or DNL, specifies the deviation of any code in the transfer function from an ideal code width of 1 LSB, The DNL is determined by subtracting the locations of successive code transition points after compensating for any gain and offset errors.

A positive DNL implies that a code is longer than the ideal code width, while a negative DNL implies that a code is shorter than the ideal width. In Fig 1.2 the DNL data obtained from Bootstrap architecture is provided for each code in a graphical format. Here we note that, DNL for any code cannot be less than '-1'. In fact, a DNL value of '-1' implies that a particular code does not exist at all(i.e., a '**missing code**' there is no analog input voltage that will produce a particular code).

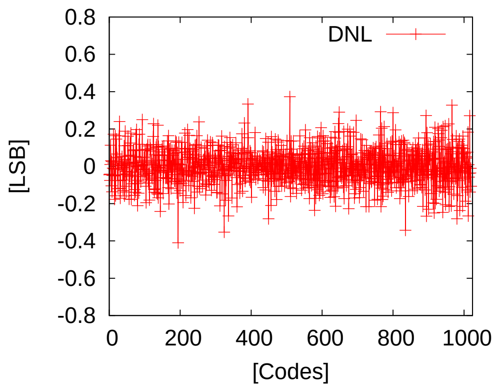


Figure 1.2: DNL

1.3.3 Integral Nonlinearity

Integral nonlinearity or INL, is a result of cumulative DNL errors and specifies how much the overall transfer function deviates from a linear response. INL is sometimes simply referred to as the linearity of the converter. The INL specification tells the designer the best accuracy that the A/D converter will provide after calibrating the system for gain and offset. There are two ways to determine INL.

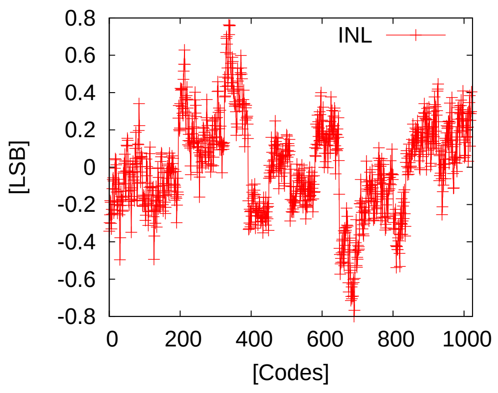


Figure 1.3: INL

- INL End-point method: The locations of the first and last code transitions for the converter are determined and a linear transfer function based on the endpoint is derived. The end point nonlinearity is determined by finding the deviation from the derived linear transfer function at each code location.

- INL Best-fit method: The best-fit response is found by manipulating the gain and offset for the measured transfer function, comparing against a linear transfer function, and balancing the total positive and negative deviations. The maximum positive and negative INL are usually specified for stated operating conditions. Like DNL graphical data, the INL graphical data is used to analyze the quality of the A/D converter. Fig 1.3 shows a graphical example of INL vs digital code for Bootstrap architecture.

1.4 AC Specifications

For applications where the signal is steady-state or has an extremely low frequency compared to the A/D converter sampling frequency, DC error specifications have the most significance. When the signal frequency is increased, however, other measures must be used to determine the performance of the A/D converter. In this case, the performance of the A/D converter in the frequency domain becomes significant to the designer. Imperfections of the A/D converter introduce noise and distortion into the sampled output. In fact, even the ideal A/D converter introduces errors into the sampled AC signal in the form of noise. The AC specifications tell the designer how much noise and distortion has been introduced into the sampled signal and the accuracy of the converter for a given input frequency and sampling rate.

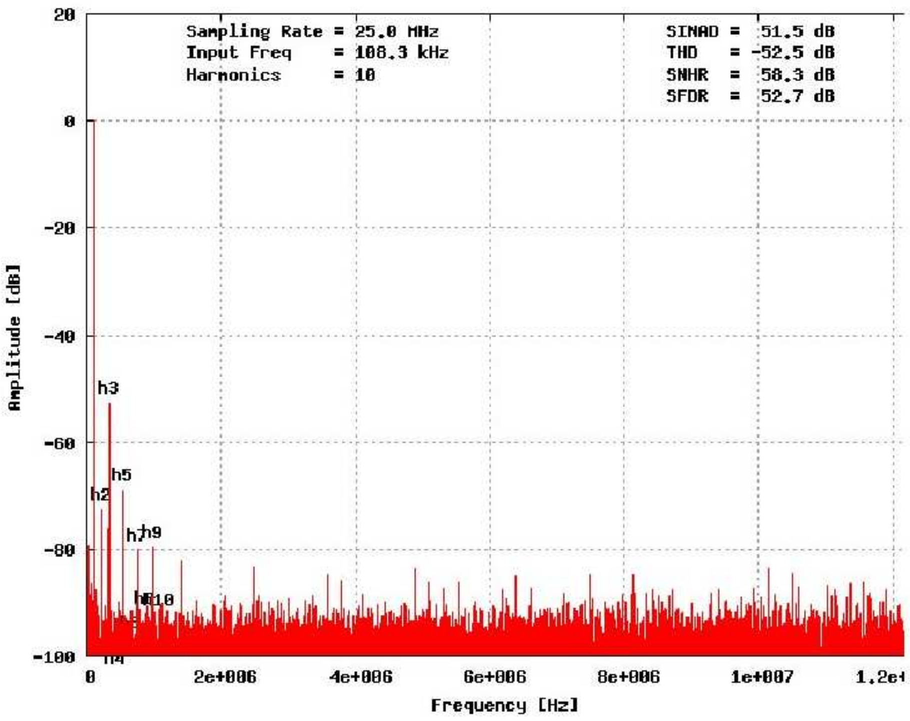


Figure 1.4: ADC Dynamic parameters vs Frequency

1.4.1 SINAD

The signal-to-noise and distortion ratio, the ratio of the signal to the total noise. It is assumed to be the ratio of RMS signal to RMS noise including harmonic distortion, for sine wave input signals.

1.4.2 THD

Total harmonic distortion, for a pure sine wave input of specified amplitude and frequency, the root sum of squares RSS of all the harmonic distortion components including their aliases in the spectral output of the ADC.

1.4.3 SFDR

Spurious free dynamic range, for a pure sine wave input of specified amplitude and frequency, the ratio of the amplitude of the ADC output averaged spectral component at the input frequency, fi to the amplitude of the largest harmonic or spurious spectral component observed over the full Nyquist band.

1.4.4 SNHR

Signal to non-harmonic ratio, for a pure sine wave input of specified amplitude and frequency, the ratio of the RMS amplitude of of the analog to digital converter output signal to the rms amplitude of the output noise which is not harmonic distortion. Plot here shows SNHR v/s sampling frequency (Fig 1.4).

1.5 Test setup

The laboratory has a well designed and fully functional setup[2] to precisely test and characterize ADC prototypes. The main components of this test setup are

- Differential input signal source.
- Reference voltage source.
- FPGA based data acquisition system.

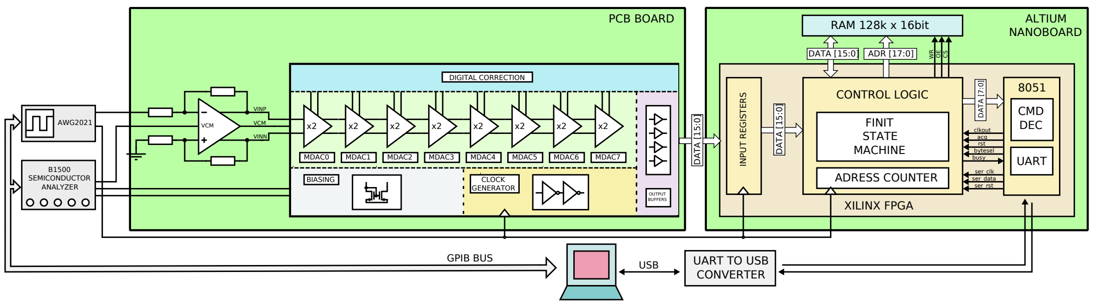


Figure 1.5: Test setup

An illustrative block diagram is given in Fig 1.5

1.6 Measurements and results

1.6.1 Static Measurements

- Static measurements are performed with the input voltage being ramped in the range from -1V to 1V.
- Noise is eliminated by averaging (few hundreds) at the same point.
- Histogram based method is used for the DNL/INL measurement.
- ENOB is computed from the INL curve.

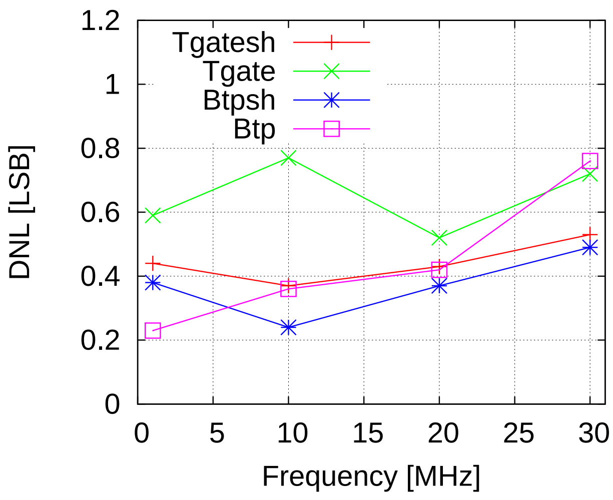


Figure 1.6: DNL vs Frequency

DNL and INL are computed based on histogram method. Both DNL and INL have been studied as a function of frequency as shown in Fig 1.6 and Fig 1.7.

1.6.2 Dynamic Measurements

- Dynamic measurements are performed by feeding ADC with sine wave as the input.
- Fast Fourier Transforms are used to calculate output signal spectrum.
- A typical measured FFT spectrum distribution is shown in Fig 1.4.

1.6.3 Power scaling

Bias currents of amplifiers were iterated through several times with frequency to find optimum bias value which would promise high Signal to Noise Ratio. From the plots in Fig 1.8 we can find that ADC power consumption is scaling linearly with respect to to change in bias currents. Also from the plot we can find that 1.2mW of power is required for every 1MHz.

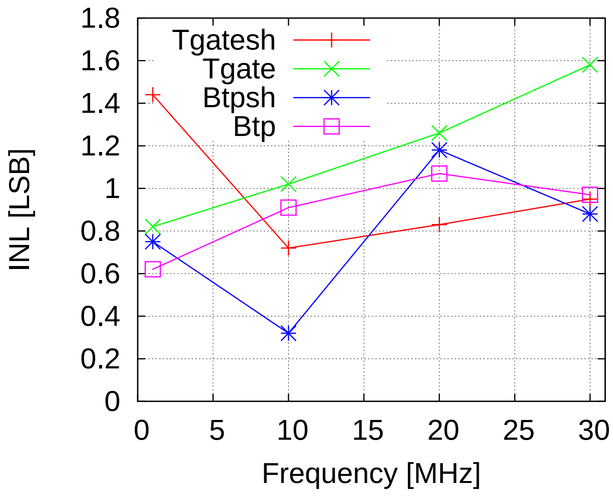


Figure 1.7: INL vs Frequency

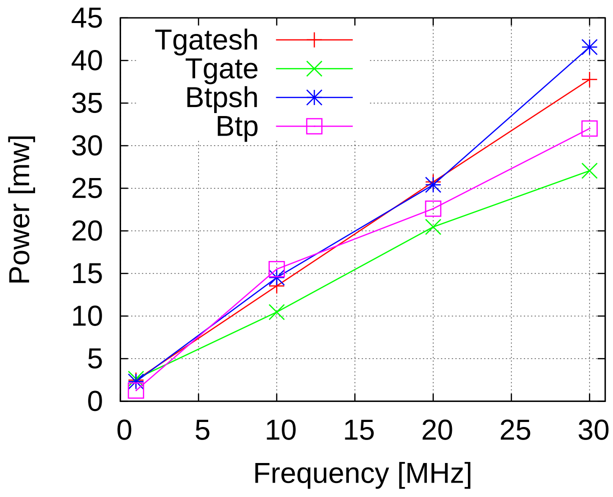


Figure 1.8: Power scaling in ADC

1.7 Conclusion

The ADC test & characterization exercise was completed with following results and findings:

- ADC was found to be functional
- $9.5 < \text{ENOB} < 9.9$
- $0.5 < \text{DNL} < 0.5$
- $1 < \text{INL} < 1$
- Signal to noise ratio was found to be 58.3 dB up to 25 MHz beyond which it starts to deteriorate.

Chapter 2

Command decoder

2.1 Introduction

The aim of this work is to implement a slow interface for LumiMulti ADC chip using a serial SPI(Serial Peripheral Interface[4]) like interface. The LumiMulti ADC is a slave, that works in SPI mode '0'(active posedge). A command decoder is used to operate the multichannel ADC in various modes like Test, Configure, Active and Low power. A Serial Peripheral Interface(SPI) based serial command protocol has been implemented to step ADC prototype and operate in these modes. The serial command has 6-bit header, 2-bit command & 8-bit data set, and the MSB is transmitted first. The two bit command set is used to switch the ADC through above mentioned four operational modes. For verification, self checking testbench and predictive analysis was performed to ensure that the design when fabricated would deliver the intended functionality with a certain accuracy.

2.2 Control/Data Format and Protocol

The serial command protocol defined for the LumiMulti ADC chip is shown below

1	0	1	0	1	1	C1	C0	D7	D6	D5	D4	D3	D2	D1	D0

Table 2.1: Command set

Commands for the LumiMulti ADC are organized in a tree hierarchy and are divided in three main command blocks:

- Header set(6b): MSB first, the header set '101011', one has to always transmit this sequence.
- Command set(2b): The control commands can be broadly grouped into four different groups, hence 2-bit command set would suffice. Table 1 enlists the control commands and corresponding data set.
- Data set(8b): Data set are always 9-bit long and for commands with shorter data the trailing bits are set to zero.

Command type	Code	Data interpretation
config	00	Mode(2b), Test ADC(3b), Low power(2b)
active adc	01	Select ADC(8b)
dac0	10	DAC0 value(9b)
dac1	11	DAC1 value(9b)

Table 2.2: Control commands

2.3 CONFIG Command

The command set described above arrive at the decoder along with different data fields. Based on the data field the **config** commands are divided into three groups, which are used to configure LumiMulti ADC in three modes namely 'Parallel', 'Test' and 'Serial'. The configuration command set is listed in table 2 alongwith the code value.

Mode	Field
Parallel	00
Test	01
Serial	10

Table 2.3: CONFIGURATION Command

Parallel Mode This is the base mode of operation, after every hard reset LumiMulti ADC is set to parallel mode. In this mode each ADC is connected to on LVDS output and sends data serially. Internal ADC clock is 10 times slower than the input clock signal. During transmission MSB are sent out first.

Serial Mode In this mode all ADC sends data through one serial LVDS output. Internal ADC clock is 80 times slower than input clock signal. The transmission pattern is, MSB of ADC7, ADC6, ADC5, ADC4, ADC3, ADC2, ADC1 and ADC0. They are all bit number 9 of those ADC's samples. Next in sequence are bit 8 and so on from all ADCs.

Test In test mode only one ADC reads out data. Consecutive samples are presented on LVDS output 9 to 0 after each clock cycle so the internal ADC clock is exactly the same as input clock signal.

Low Power Selects power mode of LVDS ports and internal buffer controls. The low power field has only two bits: first bit controls the LVDS power mode and the second bit controls both internal buffers.

2.4 Active-ADC Command

The active-adc command permits one choose which ADC is to be turned ON and which to be turned OFF. When an ADC is OFF, the power in analog part is OFF and the clock in the correction logic is stopped. Active-ADC command has 8 bits of data. First bit if set to 1/0 turns ADC7 ON/OFF, similarly second bit controls ADC6, and so on till ADC0.

2.5 DAC0/DAC1 Command

Both dac0/dac1 commands set the DACs in the analog part. DAC number 0 controls the biasing current in the main part of ADC while DAC1 is responsible for current in the sample and hold circuit.

2.6 Finite State Machine

To realize the aforesaid command decoder we conceive it as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition(here, it can be arrival of header/control/data bits), this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition. Below is shown the state transition chart for the command decoder module, the states represented here are for illustration only.

2.7 Command Decoder: RTL Design and Verification

VLSI Design Flow is a term used to describe the various design phases of an IC design. The amount of information that can be provided about this topic is so vast that one can write a complete book about the ASIC Design. However, this report addresses the important two phases(Front-end & Back-end) of an ASIC design so that readers can get an idea of how a chip is developed from concept to silicon.

2.7.1 Frontend Design

Fronend design is the first phase in the ASIC design where the logic for the chip is built using the HDL. The designer would write codes that would represent the function that he/she wants to implement. This logic realization is done using various steps in which the design is coded, verified and mapped to the actual gates using a process called synthesis. Up coming sections addresses to each one of these steps.

2.7.2 Coding

In the industry there are several HDLs being used. However there are two common hardware description languages that are very popular. They are, Verilog and VHDL. Though VHDL is a much older candidate, Verilog became very popular due to its “C” like syntax and constructs. There are plenty of arguments about which one is easier or better. In the recent years support for Verilog has been the prime focus of most of the EDA vendors. But there is always support for the VHDL codes also. No matter which language one uses for coding, the intention here is to express the design in the form of a behavioural description of code. Verilog HDL has been used to describe all of the design blocks for command decoder. The state transition chart in Fig 2.1 comprehends the function of command decoder. The states shown here are simplified for clarity, as there are several internal states at lower level of hierarchy.

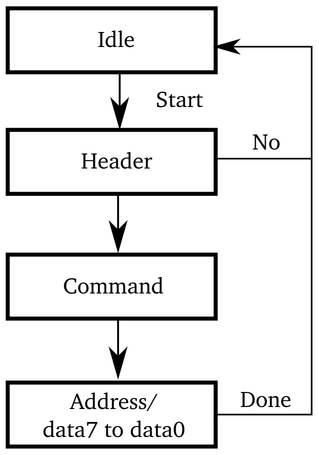


Figure 2.1: State transition flow

2.7.3 Verification

Once the coding is completed it has to be checked if the design is doing its expected function. This is called the functional verification. One can develop a test bench using Verilog or VHDL to apply all possible stimuli at the input and check the output that is generated by the code. For the above design it is not possible to check every possible combinations of the input manually by visual inspection of simulation using NC-Sim. So we have used self-checking test-benches to ensure the functional correctness of the design. A self checking testbench is a intelligent testbench which does some form of output sampling of DUT(design under test) and compares the sampled output with the expected outputs.

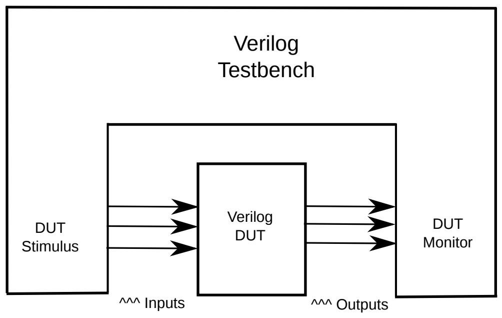


Figure 2.2: A basic testbench

A test bench is comprised of the following set of items (see Fig 2.2)

- A device under test, called a DUT. This is what your testbench is testing.

- A set of stimulus for your DUT. This can be simple, or complex.
- A monitor, which captures or analyzes the output of your DUT.
- You need to connect the inputs of the DUT to the testbench.
- You need to connect the outputs of the DUT to the testbench.

To ensure the functional correctness of the behavioral model designed in above step is validated with a set of test vectors. For visual inspection one such simulation is included here in Fig 2.3.

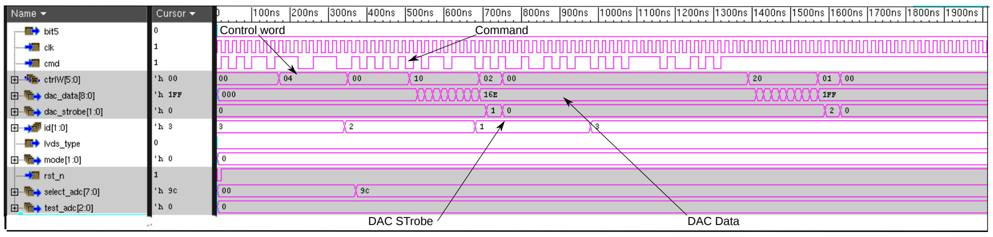


Figure 2.3: Functional Simulation

Once the designer is sure that the code functions as expected, he will take the code through the synthesis process to convert it into gates.

2.7.4 Code coverage

Verification by simulation continues to be the main strategy for validating the functionality of most complex ASICs. Test suites are created to verify the presence of desired functionality and absence of undesired bugs. A challenge for testing a design is knowing how to ensure that the test effort being exercised on a design gives maximum test coverage. As the design implements protocol that must be verified extensively to validates its correct operation. Exhaustive testing of all combinations is not possible due to time and computing resource constraints. Thus, we borrow means of testing mechanisms exercised in software industry "***Code Coverage***". Code coverage is a measure used in Software Testing. It describes the degree to which the source code of a program has been tested. Code coverage data can be used to determine the quality of test suite by identifying components of a design that are not exercised during simulation. This information is used to guide designer in determining the areas to focus test development effort. The use of various coverage metrics to make judgements on the quality of the test programs that have been used to verify the design.

Coverage metrics: **Code coverage** in ICC is classified as:
Code coverage, which includes:

- Block Coverage: Identifies the lines of code that get executed during a simulation run. It helps us in determining if the various test-benches exercise the statements in a block.

- Branch coverage: Yields more precise coverage details than block coverage by obtaining coverage results for various branches individually. With branch coverage, a piece of code is considered 100% covered when each branch of a conditional statement has been executed at least once.

- Expression coverage: Provides information on why a conditional piece of code was executed. It provides statistics for all expressions in the HDL code.

FSM coverage- Interprets the synthesis semantics of the HDL design and monitors the coverage of the FSM representation of control logic blocks in the design. With the FSM coverage, we can identify what states were executed and which transitions were taken.

- With toggle coverage, more information about the change of signals and ports, during a simulation run can be obtained. Thus it measures activity in the design, such as information on unused signals, signals that remain constant or signals that have too few value changes.

Functional coverage is performed on user-defined functional coverage points, specified using cover group statements. These coverage points specify scenarios, error cases, corner cases, and protocols to be covered and also specifies analysis to be done on different values of a variable. Functional coverage is of following types:

- Control-oriented functional coverage - Is an extension of assertion-based verification and identifies interesting functions directly. In ICC, control-oriented functional coverage points are specified using cover directives.

- Data-oriented functional coverage - Focuses on tracking data values. It includes coverage of variable values, binning, specification of sampling, and cross products. It helps designers to identify untested data values or sub-ranges. In ICC, data oriented functional coverage is specified using System Verilog constructs.

Command Decoder's top level module(design hierarchy) is exercised for code coverage and results are discussed here. The simple state machine shown in Fig2.4 has 100% coverage, as all the states has been visited during the exercise, and there are no unused states. Results of the code coverage analysis has been summarized below for the entire design in table 2.4

Type	Coverage	Passing ratio
Module/Unit	97.00%	398/410
Instance	97.00%	398/410
State	100	49/49
Arc	97.00%	58/60

Table 2.4: Code Coverage Analysis

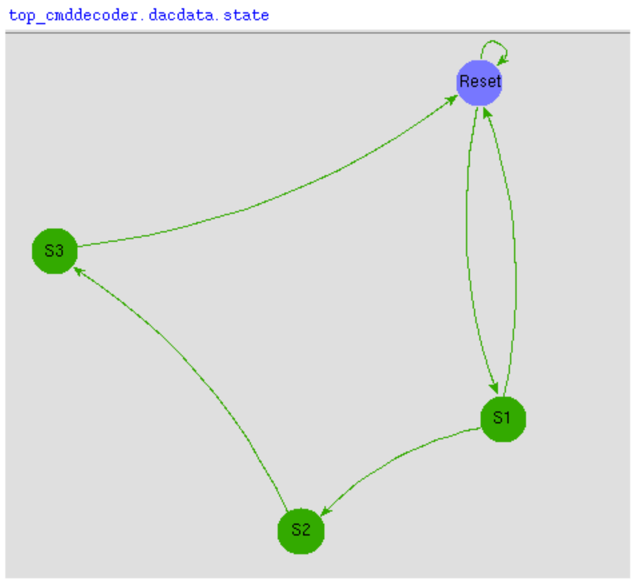


Figure 2.4: FSM Coverage chart

2.7.5 Synthesis

This is the next step in the frontend design. Using one of the various synthesis tools available, the designer will target the design into equivalent gates and flops. The output from the synthesis phase is often referred as the **netlist**, which represents the connectivity of the cells used to realise the logic. These netlists can be in the Verilog or VHDL format. There are other interchangeable formats that are used in the industry too. The tool will read the code and map the logic functions into the relevant gates/flops and provide the connectivity also. For this the tool would need a target library from which it can take the cells. The target libraries are provided by the silicon vendor, which depends on the type of technology the designer intended to use for the chip. The top level design, with sub-modules(hierarchy) and primary inputs-outputs are shown here for clarity(ref Fig2.5).

2.7.6 Gate-level Simulation

Since this netlist also represents the design, one can run the functional verification tests again with this netlist also. What is more important is the Static Timing Analysis (STA) on the netlist. STA reveals any potential timing problems like a setup or hold violation in the design. If there are some timing errors, those have to be fixed. To perform gate level simulation checks we need:

- Gate Level Netlist.
- Gate Level Simulation Library.

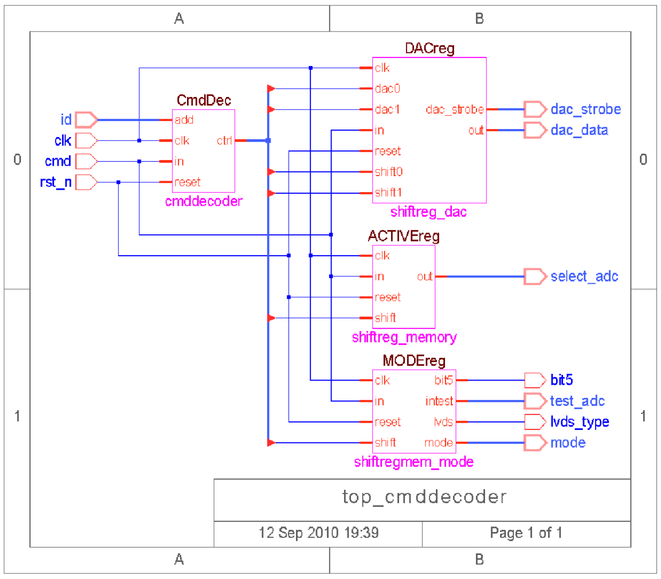


Figure 2.5: Synthesized Top-Level Module

- SDF (Standard Delay Format).
- Test bench For Gate Level Simulation.(The self-checking test-bench was reused)

The advantage of performing netlist simulation are:

- It checks if the RTL written is **synthesizable**.
- It checks if the netlist passes all test cases.
- If there are any un-initialized outputs.
- It checks if the model works at the target frequency.
- It also checks for timing violations and estimates power.

2.7.7 Types of simulation

There are two types of simulations Zero-delay (Without SDF) simulation and Simulation with SDF. The command decoder design model was successfully simulated with SDF as it has several advantages, a quick list is given below:

- SDF File Includes All The Delay For Gates.
- Check Functionality At Given Frequency.
- Identify Timing Issues.
- Check Multicycle Path.
- Check False Path.

2.7.8 Conclusion

The command decoder module designed and verified here is a low level block in the hierarchy that describes the complete ADC. As the design was correctly specified with clear boundaries (inputs, outputs) and functionality; it can now be seamlessly integrated into a modular architecture to realize the SPI like protocol command decoder.

Chapter 3

Slow control for ADC

3.1 Motivation

To realise various functionalities in the multichannel ADC, several on-chip registers will have to be updated and monitored periodically. A simple two wire I^2C compatible protocol was designed and an ASIC prototype was fabricated in 0.35μ CMOS technology. The I^2C bus can support standard and fast modes, 100KHz and 400KHz respectively. External pull-up resistors are required to support these standard and fast modes. With I^2C bus protocol it will be possible to write and modify control and command on-chip registers, and to read and monitor status register periodically. The advantage of using I^2C compatible protocol is the ability to have several devices on the same bus by following some arbitration logic[6]. An automatic verification scheme using a soft-core processor from Xilinx is being developed to test the mini-ASIC. Once completed this module can be used to interface any I^2C compatible ASIC for functional verification.

3.2 Introduction

The Interconnect Integrated Circuit or I^2C interface was originally developed by Philips Semiconductors Company[5] [6]for data transfer among ICs at the Printed Circuit Board (PCB) level in early 1980s. All I^2C -bus compatible devices have an interface allowing them to communicate directly with each other via the I^2C -bus. The concept provides an excellent solution for problems in many interfacing in digital design. I^2C is now broadly adopted by many leading chip design companies like Intel, Texas Instrument, Analog Devices. In I^2C, only two bi-directional lines are required to carry information between devices, a Serial Data (SDA) line and a Serial Clock (SCL) line. Each device can be recognized by a unique 7 or 10 bits address. The device that initiates a communication is called Master, and at that time, all the other devices on the bus are considered Slaves. Normally, Masters are Micro-controllers. The I^2C bus is a multi-Master bus, but only one Master can initiate a data transfer at any time.

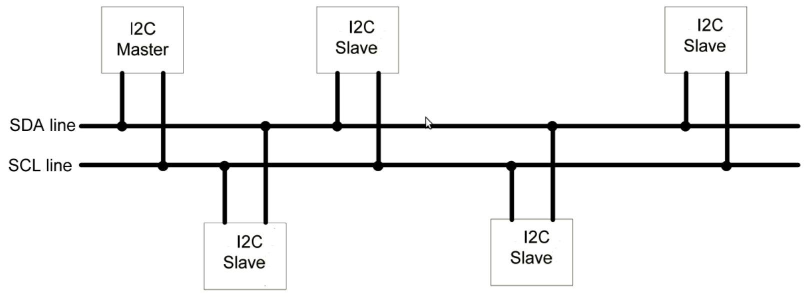


Figure 3.1: A I^2C bus configuration

3.3 I^2C Protocol

In Fig 3.1, there is one Master; the other devices are all Slaves. When a Master wants to initiate a communication, it issues a START condition. At that time, all devices, including the other Masters, have to listen to the bus for incoming data. After the START is issued, the Master sends the ADDRESS of the Slave that it wishes to communicate with along with a bit to indicate the direction of the data transfer (either read or write). All Slaves will then compare their addresses with the address received on the bus.

If the addresses are identical, the Slave with the matching address will send an ACKNOWLEDGEMENT (ACK) to the Master. Slaves whose addresses do not match will not send an ACK. Once communication is established, the two lines are busy. No other device is allowed to control the lines except the Master and the Slave which was selected. When the Master wants to terminate communication, it will issue a STOP signal. After that, both SCL line and SDA line are released and free.

3.3.1 Wired-AND

Wired-AND is a circuit technique which allows multiple devices to communicate bi-directionally on a single wire. Open-collector/open-drain devices sink (flow) current in their low voltage active (logic 0) state, or are high impedance (no current flows) in their high voltage non-active (logic 1) state. These devices usually operate with an external pull-up resistor that holds the signal line high until a device on the wire sinks enough current to pull the line low. One such configuration is shown in Fig 3.2. Many devices can be attached to the signal wire. If all devices attached to the wire are in their non-active state, the pull-up will hold the wire at a high voltage. If one or more devices are in the active state, the signal wire voltage will be low.

3.3.2 Start & Stop bit

When a Master wants to initiate a data transfer, it issues a Start condition and when it wants to terminate the transfer, a Stop condition will be initiated. There can be multiple Starts during one transaction called a repeated Start. The Master can then release the Stop condition whenever it wants to. As we can see from Fig 3.3, a Start is issued by bringing the SDA line low while the

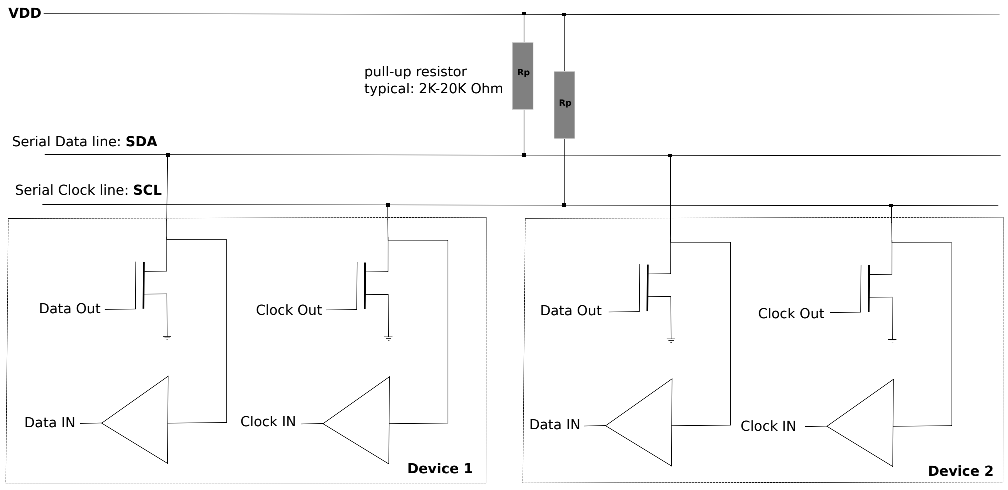


Figure 3.2: Wired-AND logic

SCL line is high. After that the Master controls the SCL line and can generate clock signals. A Stop condition is implemented by transitioning the SDA line high while the SCL line is high. Start & Stop are a special sequence, as the transition on SDA line happens when SCL is High. All other data transition on SDA line should happen when SCL is low (Fig 3.4).

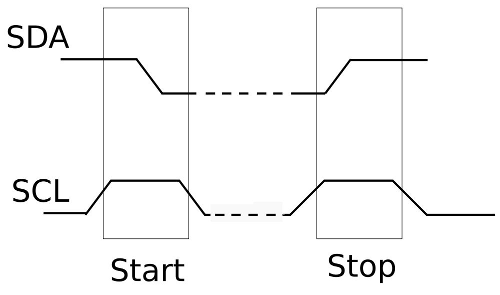


Figure 3.3: I^2C Start and Stop bits

3.3.3 First byte

The I^2C bus is a byte-oriented protocol. After signaling Slaves by the Start condition, the Master sends starting bytes to the Slave. There are two components that make us the starting bytes: Slave address and data direction (Read or Write). The Master sends the MSB (Most Significant Bit) first and the LSB (Least Significant Bit) last. There are two addressing modes in the I^2C protocol: the 7-bit and 10-bit address modes. We will only consider the 7-bit addressing mode. After the START condition (S), a Slave address is sent. This address is the first 7 bits, the eighth bit is a data direction bit (RnW). If the direction bit is 0, it indicates a transmission (or WRITE). If the bit is 1, it indicates a request for data (or READ)(see Fig 3.5). A data transfer is always terminated by a STOP condition (P) generated by the Master. However, a Master can

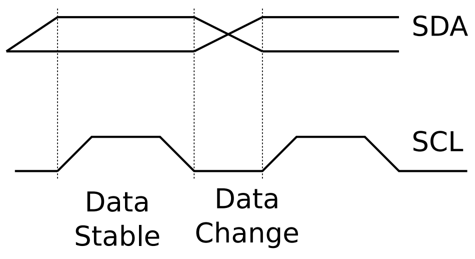


Figure 3.4: Data change allowed

generate a repeated START condition (Sr) and address another Slave without first generating a STOP condition.

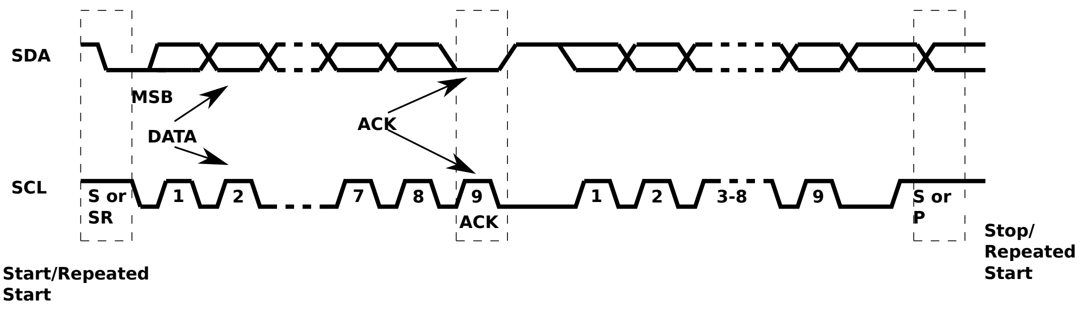


Figure 3.5: A complete data set

3.3.4 Acknowledgement

Acknowledgement is obligatory in order to inform the transmitter that data has been successfully transmitted. The Master generates the acknowledge-related clock pulse and the transmitter releases the SDA line (HIGH) during the acknowledge clock pulse so that the receiver can take control of the SDA line. If the receiver does not acknowledge, leaving the SDA line high, the transfer must be aborted. If it acknowledges by pulling the SDA line low, the transmitter knows that data has been successfully received, so it keeps sending data to the receiver.

3.3.5 A Complete Data Transfer

All of the major aspects of I^2C bus discussed so far are combined to create a complete data transfer from the transmitter to the receiver. The SCL signal, Start and Stop signals, and the first byte must be generated by the Master. The Acknowledgement of the first byte must be generated by the Slave when it recognizes its address on the bus. The other Acknowledgements are generated by the receiver.

3.4 I^2C Master

There are thousands of I^2C peripherals in the market today, ranging from data converters to video processors. The I^2C bus is a good choice for designs that need to communicate with low speed peripherals due its simplicity and low cost. This reported design, can be constructed and utilized in a Xilinx FPGA device demonstrating how a fast and flexible design can be prototyped using reconfigurable FPGA devices. This I^2C master can be used as a general purpose I^2C bus controller, also we can further customize the Hardware Description Language (HDL) to meet specific requirements.

3.4.1 Features

- I^2C bus speeds of 100kbits/sec and 400kbits/sec.
- I^2C 7-bit addressing.
- Interrupt mode or polling mode.
- Automated verification using OpenCores I^2C slave module.
- FPGA Prototype with MCP23008 and STCN75.

3.5 Design & Sub-blocks

An I^2C Master must have the ability to create the I^2C Serial Clock, START and STOP signals. It also has to keep track of the number of transferred bytes to determine an appropriate time to stop a data transfer.

Based on the above characteristics, the master circuit was broken down into:

- Finite state machine
- Logic blocks
- Slave
- Clock generators
- Multiplexers

3.5.1 Behavioral model

Control signal and data are loaded at reset. Slave address, R/W control bit are loaded to the Start Byte. After being reset, the FSM (Fig 3.7) creates a START signal by suitably controlling two multiplexers through a logic control block. The state machine ensures that, once START is signaled other sequence of control signal are stepped through till data direction bit. ACK signal, is also monitored by state machine for the correct execution of protocol. During data transmission sequence, a Master writes byte by byte and Slave acknowledges every 9th clock pulse indicating successful receipt of data. And during read mode, Master behaves similar to a Slave (in receive mode). The Master signals the end of read by not acknowledging (NACK) the last byte of data. Immediately after

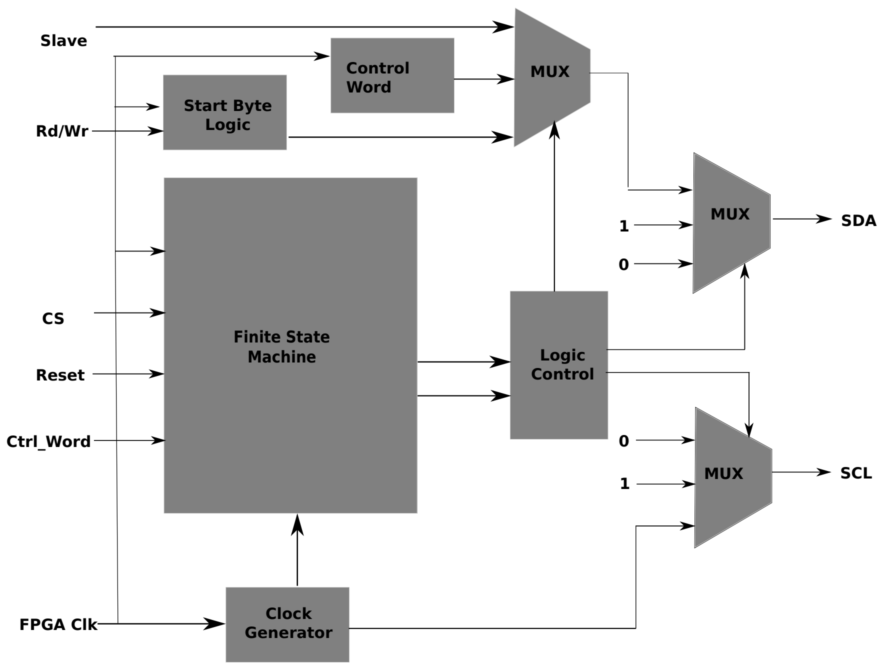


Figure 3.6: Top schematic of I^2C Master

NACK, STOP follows. The STOP bit required to end a data transmission is generated by the state machine in very similar way as START is generated. Verilog HDL is used to capture the complete behavioral model of I^2C Master, the design steps are very similar as described for the command decoder. A top level schematic of the Master module is shown in Fig 3.6, which highlights the interconnection of above listed components. As the design is inteneded for FPGA application the differences in design flow are specified and explained in next section of this chapter. FPGA design flow is shown in Fig 3.8.

3.5.2 Verification

I^2C Master circuit has been verified thoroughly by performing detailed predictive analysis using OpenCores I^2C slave module and code coverage tools from Cadence. FPGA prototypes were built, which were then interfaced with real world I^2C devices like digital temperature sensor-STCN75 and 8-bit data I/O expander MCP23008(see photograph 3.9).

3.6 Prototype

For designing digital systems using programmable logic like FPGA devices, computer aided design (CAD) software packages are used. These software packages assist the designer through all the stages of the design process. Therefore, most of the CAD packages for programmable logic devices provide the following main functions:

- Description of the digital system

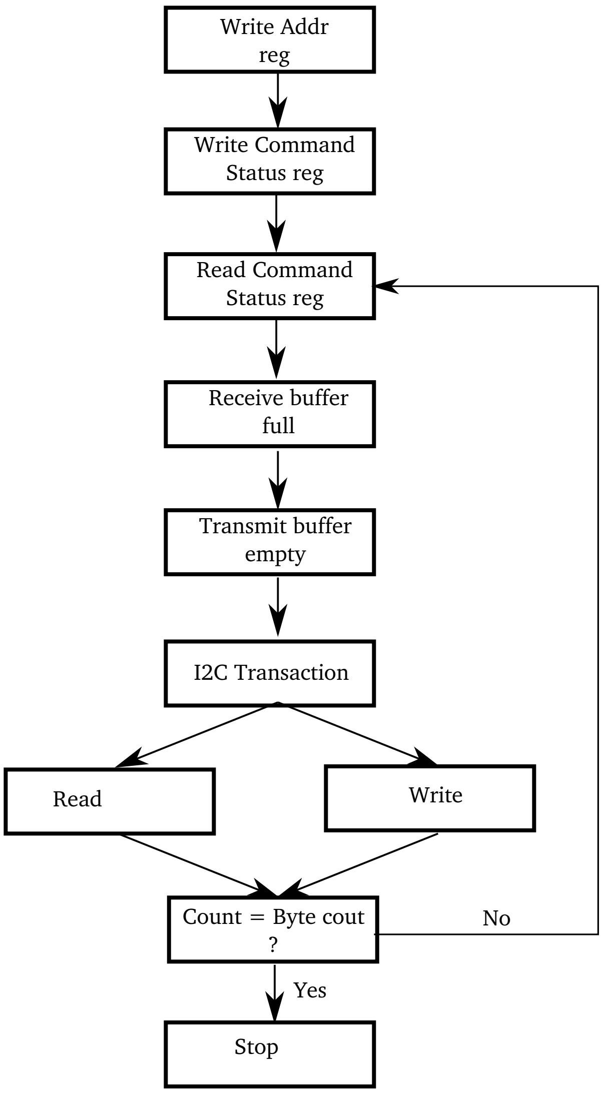


Figure 3.7: Programming Flow for the I^2C Master

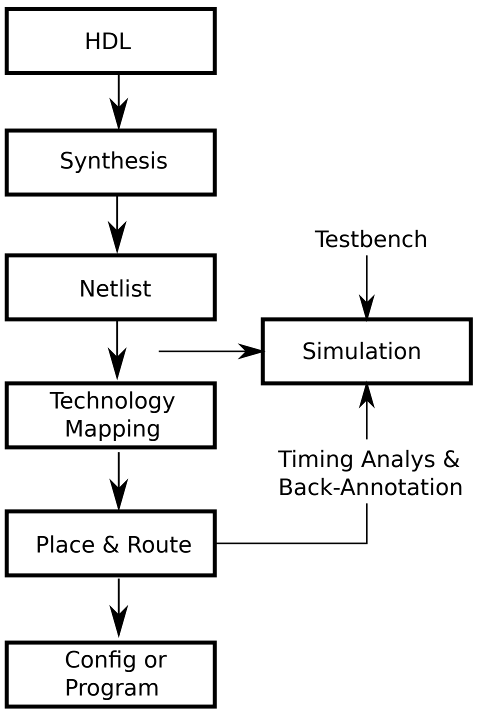


Figure 3.8: FPGA Deisgn Flow

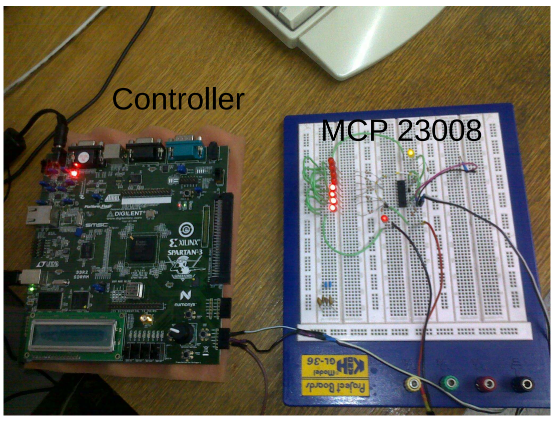


Figure 3.9: I^2C Master(FPGA) interfaced with MCP23008(breadboard)

- Synthesis of the description, which means transforming the description into a netlist containing basic logic gates and their interconnections
- Functional simulation of the system based on the netlist obtained, before the implementation in a specific device;
- Implementation of the system in a device, by adapting the netlist for an efficient usage of the device's available resources;
- Configuration (programming) of the device in order to perform the desired function.

3.6.1 System description

As stated earlier Verilog hardware description language is used to describe the I^2C Master. The language is widely used today, and preferred for the description of systems with high complexity, due to the following advantages

- The capability of a functional description of systems, this being a higher-level description, without detailing the structure at the basic gate level. Therefore, the time required for describing complex systems is significantly reduced.
- The independence of the HDL description among different types of devices. While the logic diagrams are made with library components specific to a certain device family, HDL descriptions are completely independent of a specific device, so that the same description can be used for implementing the system in a certain FPGA device, but also in another type of programmable logic device, for instance, in programmable logic array.
- The possibility to modify the HDL description of a system in a simpler manner, due to the fact that such a description also represents a documentation of the system.

3.6.2 Synthesis

After design entry and optional simulation, the next step in the design flow is the synthesis. The Xilinx ISE Design Suite environment includes the Xilinx Synthesis Technology (XST) software, which synthesizes Verilog designs to create Xilinx-specific netlist files, known as NGC files. Unlike outputs from other tools, which consist of an EDIF file with an associated NCF constraints file, NGC files contain both logical design data and constraints. The NGC file is then accepted as input to the translate step of the design implementation process. In addition to NGC files, the XST software also generates the following outputs: a synthesis report, an NGR file with the RTL (Register Transfer Level) schematic, and a technology schematic. The synthesis report contains the results from the synthesis run, including an area and timing estimation. The RTL schematic is a representation of the design before the optimization using generic symbols, such as adders, multiplexers, counters, gates. This schematic is generated after the HDL synthesis phase of the synthesis process. The technology schematic is a representation of an NGC file using logic elements optimized to the target architecture or technology. This schematic is generated after the optimization phase of the synthesis process.

First, a parsing of the HDL code is performed, when XST checks whether the HDL code is correct and reports syntactic errors. If there are no syntax errors, the HDL synthesis is performed. XST analyzes the HDL code and attempts to infer specific design building blocks or macros (such as multiplexers, memories, adders, subtractors) for which it can create efficient technology implementations. To reduce the amount of inferred macros, XST performs a resource sharing check that leads to a reduction of the area and an increase in the clock frequency. At this step, the XST software recognizes the Finite State Machines (FSMs) independent of the modeling style used. To create the most efficient implementation, XST uses the optimization criterion that has been specified (area or speed) to determine the FSM encoding algorithm that will be used. The last step of the synthesis is the low-level optimization. XST transforms inferred macros and general logic into a technology-specific implementation.

3.6.3 Simulation

Xilinx ISim is a Hardware Description Language (HDL) simulator that enables a designer to perform functional and timing simulations for VHDL/Verilog designs. Simulation Libraries -The Xilinx simulation device libraries are precompiled, and are updated automatically when updates are installed.

The files needed to run a simulation(see Fig 3.10) in ISim are the following:

- design files, including stimulus file
- any user libraries
- any other miscellaneous data files

A Stimulus File -an HDL-based test bench is the stimulus file. For I2C Master a detailed test bench was created using the text editor. But there are few other options too:

- Language Templates - Use a template to populate the file correctly, such as those available with the ISE software.
- Third-party tool - Create or edit an HDL test bench in any vendor-provided tool.

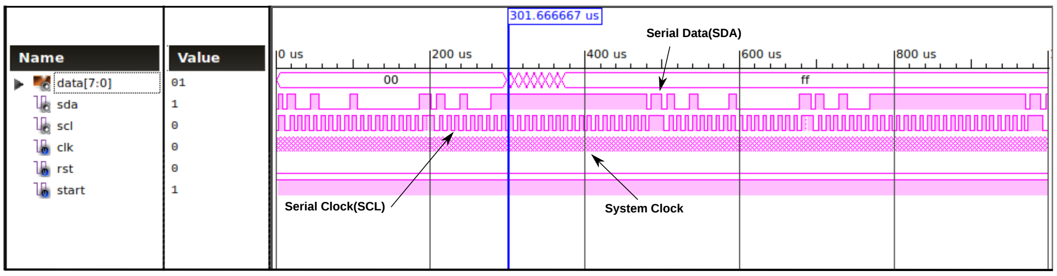


Figure 3.10: Functional Simulation

3.6.4 Design Constraints

The design constraints are instructions given to the FPGA implementation tools to direct the design processes and help improving the performance. Constraints specify placement, implementation, signal direction, and timing considerations for timing analysis and for design implementation. The design constraints are usually placed in the User Constraints File (UCF), but they may exist in the HDL code or in a synthesis constraints file.

3.6.5 Types of constraints

Each type of constraint serves a different purpose and is recommended in different situations. The most commonly used types of constraints are the following:

- Timing constraints
- Placement constraints
- Synthesis constraints

Timing constraints describe the timing performance requirements of the design. Tim-ing constraints are usually specified globally, but can also be specified for individual paths. Global constraints include period constraints for each clock signal (PERIOD), setup times for each input (OFFSET_IN), and time specifications from the clock edge at the input pin until data becomes valid at the output pin (OFFSET_OUT). Timing constraints can be entered us-ing the Create Timing Constraints process in the Project Navigator graphical interface.

Placement constraints control the mapping and placement of symbols defined in the netlist to the resources of the device used for implementation. These constraints can be de-fined for each type of logic element of the FPGA device. Individual logic gates are mapped into CLB (Configurable Logic Block) function generators before the constraints are read, and therefore cannot be constrained. In a constraints file, each placement constraint acts upon one or more symbols. Every symbol in a design carries a unique name, which is defined in the input file.

Synthesis constraints instruct the synthesis tool to perform specific operations. Syn-thesis constraints control how the synthesis tool processes and implements FPGA resources, such as state machines, multiplexers, and multipliers, during the HDL synthesis and low-level optimization steps. Synthesis constraints also allow control of register duplication and fanout control during global timing optimization.

3.6.6 Implementation

In this process, netlists (*.edf) are translated, followed by mapping, placement, and routing. Finally, the corresponding hardware configuration bit stream (*.bit) for programming the FPGA is generated. Design implementation is extremely important to the success of your project.

After synthesis, you run design implementation, which comprises the following steps:

- Translate - Translate is the first step in the implementation process. The Translate process merges all of the input netlists and design constraints information and outputs a Xilinx Native Generic Database (NGD) file. The output NGD file can then be mapped to the targeted device family. All processes necessary to successfully complete the Translate process are run automatically. The Translate process generates an NGD file.

- Map - After translation, you can run the Map process on your design. The Map process takes the Xilinx Native Generic Database (NGD) file created during translation, runs a design rule check, and maps the logic design to a Xilinx FPGA. The results are output to a Native Circuit Description (NCD) file, which is used for placing and routing. All processes necessary to successfully complete the Map process are run automatically. The Map process runs a design rule check on the design, and the logic design is mapped to a Xilinx FPGA. The resulting NCD file is output to the project directory.

- Place and Route - The Place and Route process takes a mapped Native Circuit Description (NCD) file, places and routes the design, and produces an NCD file to be used by the programming file generator, BitGen. All processes necessary to successfully complete the Place and Route process are run automatically. The Place and Route process outputs an NCD file that the programming file generator, BitGen, uses to create a BIT file.

- Programming -The Generate Programming File process runs BitGen, the Xilinx bitstream generation program, to produce a bitstream (BIT or ISC file) for Xilinx device configuration. The programming file is saved in your project directory.

Finally, a prototype of the Master was built on the Xilinx Spartan 3AN[8] and real I^2C devices like MCP23008[9] and STCN75[10] were used to WRITE and READ data thus conforming to the complete protocol.(see photograph 3.9)

3.7 I^2C Slave

This design implements an I^2C slave module targeting AMS 0.35u technology. It follows the I^2C protocol specified in section 3.3 to provide device addressing, read/write operation and acknowledgement mechanism.

It adds an instant I^2C compatible interface to any component in the system.

3.7.1 Features

- Compatible with I^2C bus specification.
- Supports I^2C 7-bit addressing mode.
- Supports random read/write.
- Start/Stop/Repeated Start detection.
- Supports sequential read/write.
- Uses ACK/NACK to communicate with I^2C Master.

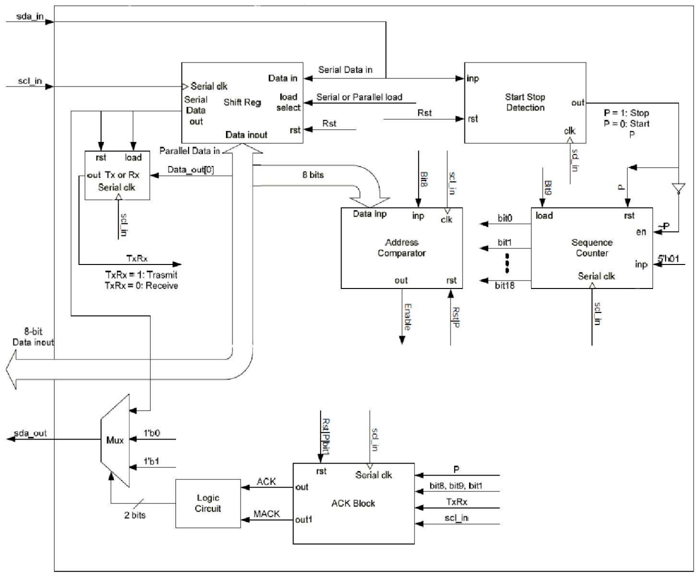


Figure 3.11: Top Schematic of I^2C Slave

3.7.2 Slave Interface ASIC Implementation

The design described here, consists of an I^2C slave device, an 8 byte register array and 8-to-1 multiplexer. This additional register array, is used to write data into slave and read it back. Also the multiplexer can be utilized to asynchronously read data from the register array. To reduce the number of I/O pins, the asynchronous data output from register array is split into nibble. We can select upper or lower nibble by using a control signal.

3.7.3 State Machine

The I^2C slave design consists of a state machine which steps through the I^2C operation mechanism. The start and stop conditions are detected asynchronously to initiate or terminate the I^2C operations. The ACK is generated when the slave address matching the address of the slave device and when a byte of data is successfully received. The state machine supports sequential/continuous read and write by returning to the ACK state. The programming flow for the I^2C slave model is shown in Fig **??**.

3.7.4 Frontend design

As descibed earlier there are two phases in ASIC design. Following the detailed description of Frontend design for the design and development of command decoder in section 2.7 only a brief description of each step in design flow is mentioned. The second phase of ASIC design flow i.e., back-end design is described in detail in later section of this chapter.

- RTL & Logic Simulation: The RTL code for the slave design consists of a hierarchy of several sub-modules, and a top-level module that binds

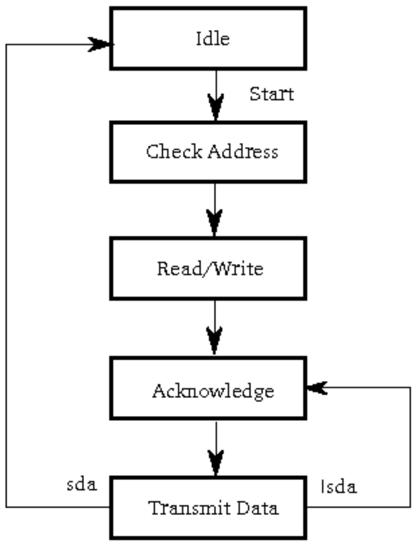


Figure 3.12: Programming flow I^2C Slave

them together. Several logic simulations were performed, to ensure the functional correctness of the design. One such simulation (Fig 3.13 is presented here for visual inspection, where data is being written-to & read-from register array.

- Logic Synthesis: Encounter RTL Compiler was used to turn the abstract RTL code into a design implementation in terms of logic gates, to target the creation of ASICs.

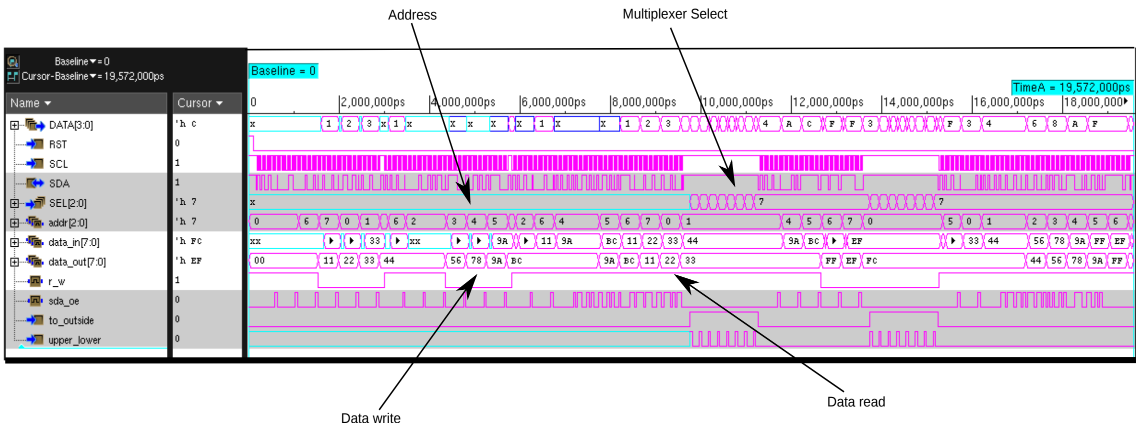


Figure 3.13: Functional simulation of I^2C Slave with Reg Array

3.7.5 Backend Design

The second major phase in the ASIC design is commonly known as the backend where the cells in the netlist are placed on the die and then routed.

3.7.6 Placement

The netlist, which has no violations is read into the placement tool and placed within the die area according to the guidelines given to the tool. Placement of cells depends on the connectivity and also on the distance between the cells so as not to cause any timing violation.

- Cells that are placed far apart will have a lengthy wire to connect them, and this will cause additional delay.
- Another reason for more delay is, the fanout. When the fanout of a cell increases, this increases the capacitance load on the driving cell. This will also cause output signal to propagate slowly from one point to another.

The placement tool will be given a set of constraints that will indicate the timing requirements of the design. Hence the tool will try to place the cells in such a way that there are no timing deteriorations. In addition to the cells, the tool will place the IO pads also along the periphery of the die so that the pins can be connected to the pads using metal bonding at a later stage.

3.7.7 Routing

Once the cells and IO pads are placed satisfactorily, the design is given to the routing tool for routing. This tool will connect the cells according to the information in the netlist. When the routing is completed the designer can extract the actual delay contributed by the cells and the wires. These timing numbers are to be fed to the STA tool once again to determine the conditions of violations based on the final routing. If there are any violations, they need to be fixed by performing some netlist changes or by improving the routing/placement or by doing both. This will take few iterations to converge on the timing.

3.7.8 Netlist & Post-layout Simulations

This phase of verification usually requires re-simulating the RTL test suite on the netlist with the back-annotated timing. Gate-level simulation verifies the output of the synthesis and place & route tools and incorporates timing to bring the DUT closer to real implementation. From inspection of the IRUN log files we report a successful netlist simulation(see Appendix).

When the backend phase is completed with no violations, the design is converted into a database that will be used to build the mask for the semiconductor processing. These data will be given to the processing units for manufacturing the chip.

3.7.9 Post-silicon validation

Post-silicon validation is the last step in the development of a semiconductor integrated circuit. As we cannot rely on pre-silicon design verification alone

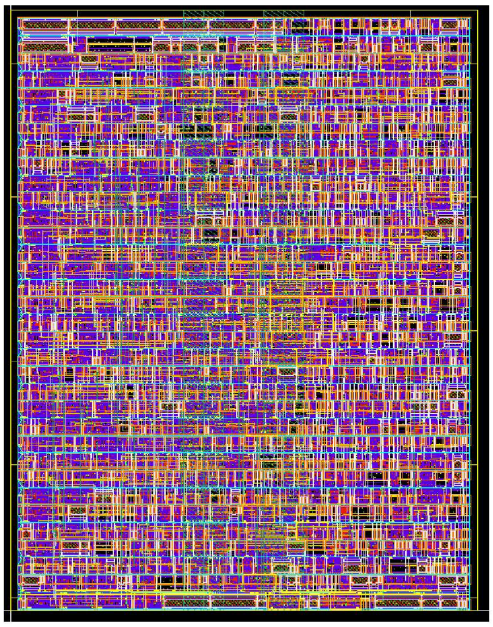


Figure 3.14: Layout

to detect all design bugs, the simulations done during design phase are several orders of magnitude slower than actual silicon. Post-silicon validation involves operating a manufactured chip in actual application environments to validate correct behaviors over specified operating conditions.

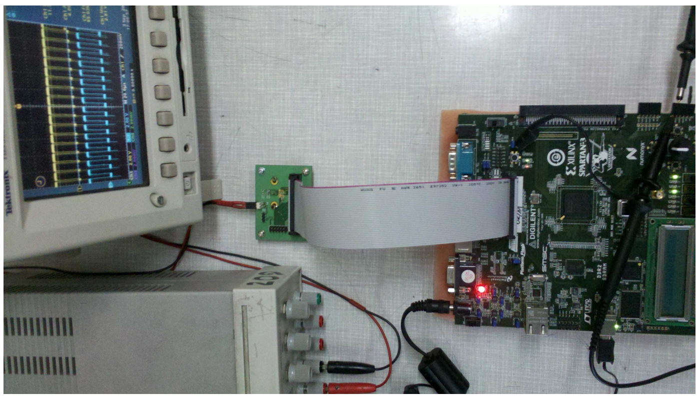


Figure 3.15: The Assembly

3.7.10 PCB design & printed circuit assembly

A dedicated printed circuit(layout Fig 3.17) board is designed and fabricated to mechanically support and electrically connect the manufactured chip. The printed circuit assembly operates as a dedicated slave which is controlled by I^2C Master configured on FPGA. The printed circuit assembly has adequate interfaces such that some of the critical internal signals have been routed to these interfaces, so that they can be probed using test & measuring instruments. The Assembly photograph 3.15 shows how the Master and Slave are connected for testing the chip.

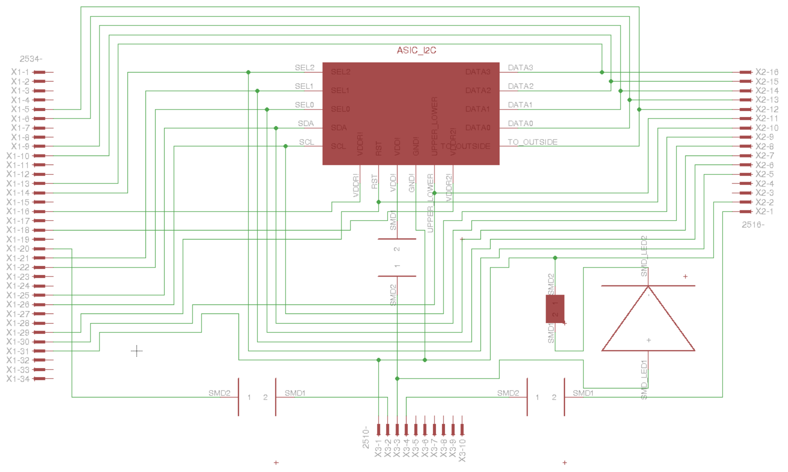


Figure 3.16: PCB Schematic

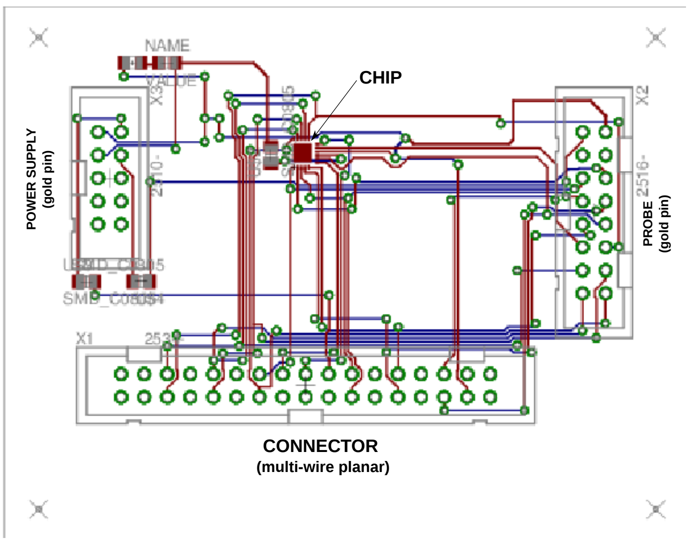


Figure 3.17: PCB Layout

3.7.11 IC Testing

The I^2C Master programmed on FPGA was sending out set of data which would correctly address the Slave on the Assembly. In response to the command received by the Slave it would write a set of data on to the register array and which are read back by the Master and displayed on the LED. Visual inspection of these 8 set of data, ensured the functional correctness of the Slave along with the register array which had held the data. See Appendix A.

Synthesizable I^2C Master A synthesizable and reusable I^2C Master was designed solely for IC testing purpose. The main objective behind this design was to test the chip by configuring the making the slave card to talk with Master which has been programmed on the FPGA. The abundantly available I/O ports on the FPGA spartan-3 board were used to configure and connect the slave card.

I^2C Master Internal Blocks The RTL design can be comprehended to the following the block and signal flow diagram 3.18.

Port Description:

- ADDR [2 : 0]: Internal register address bus.
- $DATA\_I$ [7 : 0]: Data input bus.
- $DATA\_O$ [7 : 0]: Data output bus.
- WR: Write Signal.
- RD: Read Signal.
- CS: Chip Select
- INT: Interrupt ($H \uparrow$ Input data buffer: Full/ Output data buffer: Empty)

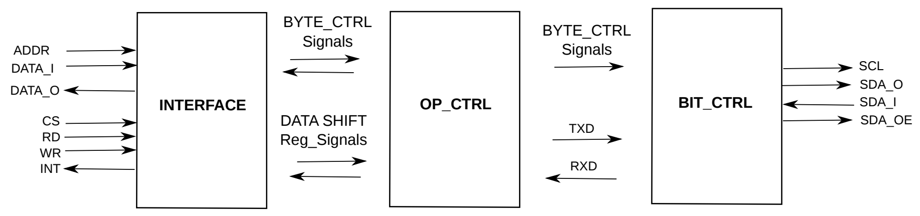


Figure 3.18: I^2C Master Block & Signal Diagram

Characteristics:

- SCL frequency is 1/4 of the output frequency.
- 7-bit address mode.(with little modification- 10 bit addressing mode)
- Continuous reading and writing.
- An interrupt signal has been added.
- Transmit buffer empty flag.
- Receive buffer full flag.
- Single master mode.(Extendible to multi-master with Bus arbitration logic possible)

Name	Address	Access	Details
DEVICE	0	W	Device Address
TARGET	1	W	Target Address
OP_NUM	2	W	Number of bytes of data transmission
DATA_IN	3	W	Output data buffer
DATA_OUT	3	R	Input data buffer
STU_REG	4	W / R	Status Register
CTRL_REG	5	W / R	Control Register

Table 3.1: Internal Register

Bit	Access	Details
7	-	NA
6 to 0	W	7-bit Device Address

Table 3.2: Device Address

Bit	Access	Details
7 to 0	W	Target Address: Access to a peripheral(register) address [Reset to 0]

Table 3.3: Target Address

Bit	Access	Details
7 to 0	W	The number of bytes of data transmission [Reset to 1]

Table 3.4: Byte transmission

Bit	Access	Details
7 to 0	W	Data Output buffer [Reset to 0]

Table 3.5: Output data buffer

Bit	Access	Details
7 to 0	R	Data Input buffer [Reset to 0]

Table 3.6: Input data buffer

Bit	Access	Details
7 to 4	-	NA
3	R	BUSY: 0 Idle (No operation)
2	W / R	ASK_IN: Write operation again, i.e., peripheral return (clear manually)
1	R	FULL: Receive buffer full, i.e., a data has been read from the peripheral. Host to read back data from the receive buffer. (Auto cleared)
0	R	EMPTY: Transmit buffer empty, data ready to send. As $!OP\_NUM$, continue write operation.

Table 3.7: Status Register

Bit	Access	Details
7 to 2	-	NA
1	W	DIR: Data transfer direction. (Write:1 and Read:0)
0	W / R	REQ: Initiate a data transfer. (Auto cleared)

Table 3.8: Control Register

3.7.12 Conclusion & Future work

We report successful implementation of an I^2C Master on FPGA and I^2C Slave on mini ASIC. We could modify the protocol to enhance the number of devices that could be addressed and increase the data transfer rate to 3.4Mbps. Further, with some arbitration logic this two-wire system bus can be operated as a multi-master communication channel between several devices.

Appendix A

Post-silicon validation

Testing a new ASIC, by writing software to verify the design of a silicon device is quite different from other types of embedded software development. Here I discuss validation techniques to successfully plan, develop, and execute software used to verify the a new silicon device. Although my experience in validating the chip(ref:3.7.2) has been unsuccessful, but there has been immense learning opportunities and the same learning has been indispensable in my soon to begin graduate studies. The techniques presented here apply to any type of device with WISHBONE (ref:A.2) interfaces to be controlled by micro-controllers/ soft-core processors.

There are two main categories of design testing: pre-silicon verification and post-silicon validation. In pre-silicon verification, design analysis tools are used to simulate the design and the test environment before an actual silicon device is created. Since the environment is simulated, there is flexibility in setting up test cases at the block and gate level. Inputs can be injected and outputs probed and logged from virtually anywhere in the design. A powerful, low-level test and debug environment is the result. This is equivalent to white-box unit testing in the software world.

Most modern chip designs are developed in a hardware description language such as Verilog or VHDL. The language used for the design in most cases is also used to create the pre-silicon test cases. A problem with design simulation and pre-silicon testing is that they take a long time to execute when compared to actual silicon. The subject of pre-silicon design verification was discussed in section 2.7.3

To my knowledge, post-silicon validation is a less documented area of testing and there are no methodological approaches to it. It is performed on as a case to case basis on the design to be tested once the fabricated device is delivered. The tests use programming languages such as C and assembly and are run on a reference or validation board containing the target silicon device. Testing can be done at speed and now involves interaction with other hardware and peripherals.

One difficulty is that visibility inside the chip is limited; internal signals cannot be probed as they can in a simulation environment. Test and debug is a challenge at this stage. The work is done at the device's register interfaces using software and at its external signals using tools such as logic analyzers and oscilloscopes.

A.1 PicoBlaze Processor as a test & debug processor

A.1.1 The processor

PicoBlaze is an efficient 8-bit microcontroller architecture which can be synthesized in FPGAs. PicoBlaze is similar to many microcontroller architectures but it is specifically designed and optimized for Xilinx FPGAs. There are also larger microprocessors that can be synthesized into FPGAs such as the 32-bit MicroBlaze microprocessor. PicoBlaze and MicroBlaze are typically referred to as soft processor cores since they are synthesized from an HDL and use the programmable logic and routing resources of an FPGA for their implementation, as opposed to a dedicated processor hard core such as the PowerPC that is incorporated in some Virtex II and Virtex-4 FPGAs.

PicoBlaze (see Fig A.1) consists of two parts:

- The processor core- Constant (k) Coded Programmable State Machine (KCPSM).
- Program memory- ROM

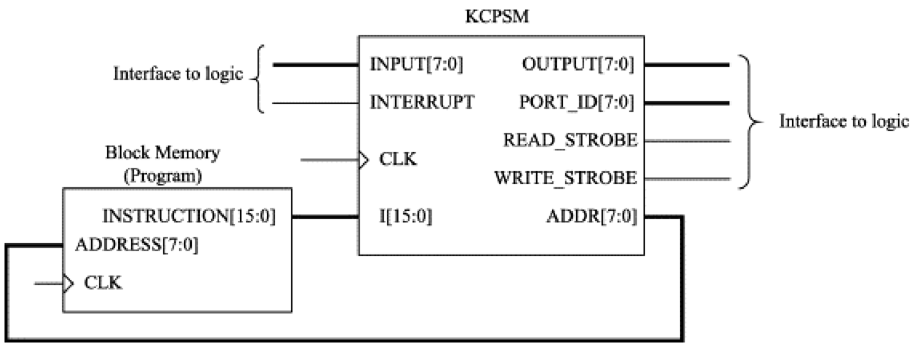
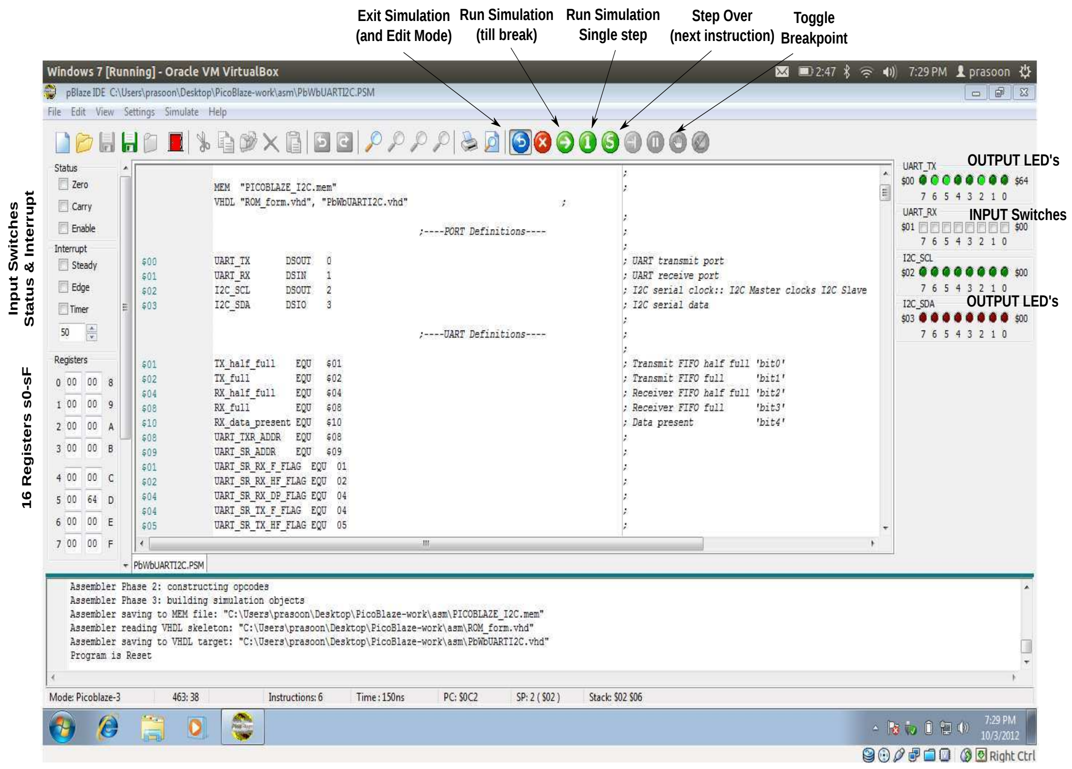


Figure A.1: Basic Components of PicoBlaze

The program memory from which instructions are fetched and executed by the processor core. (Note that the program memory is referred to as an instruction ROM or program ROM in PicoBlaze documentation since the processor core cannot write to the program memory.) As a result, there are also two Verilog HDL files that are used to construct the complete PicoBlaze with program. The KCPSM3.v file is optimized for Spartan 3 by calling design primitives specific to Spartan 3 (also for Virtex II and Virtex-4) and, as a result, this Verilog HDL file should not be modified by the user. The KCPSM3 require approximately 96 slices in a Spartan 3. The program memory, on the other hand, is a VHDL file specific to the user's desired program to be executed by the PicoBlaze core and is generated automatically by the assembler (KCPSM3.exe) from your assembly language program, name.psm. (Note that the prefix for the .psm file must be 8 characters or less.) The program memory is implemented in a single Block RAM in the FPGA configured to function as a $1K \times 18$-bit ROM. The program to be executed is typically initialized in the Block RAM during the download of the overall design (including PicoBlaze and other user logic) and, as a result, is normally assembled prior to synthesis.

A.1.2 pBlazIDE Simulator

One method of verifying the assembly language program prior synthesis is to use the PicoBlaze simulator, pBlazIDE.exe[11]. Open pBlazIDE and under Settings select PicoBlaze version. Under File select Import and then select the assembly language file that you have written with any text editor. This will import the assembly program and convert some of the PicoBlaze assembly instructions to simulator specific instructions (for example, the INPUT instruction is converted to IN). It is very useful to note the difference between KCPSM3 Assembler and pBlazIDE, tableA.1. When imported file is opened with pBlazIDE assembler a screen similar to that of FigA.2 can be seen. Input values can be set by the check boxes for the *switches* and output values can be observed on the *LEDs* to the right of the screen. Also note that the contents of the 16 registers **(s0-sF)** can be observed to the left of the screen along with the status bits and flags. Operation of the simulation is primarily controlled by the 9 buttons shown in the orange oval. From left to right, these buttons control: exit simulation and return to edit mode (blue button), reset the simulation (button with an X), run the simulation (continuously or until a break point is reached) (button with the arrow), single step the simulation (button with a 1), step over the next instruction (button with an S), simulate to cursor, stop or pause the simulation (button with parallel vertical lines), toggle breakpoint (button with hand),and clear all breakpoints.


Exit Simulation (and Edit Mode)
Run Simulation (till break)
Run Simulation Single step
Step Over (next instruction)
Toggle Breakpoint
Input Switches Status & Interrupt
16 Registers s0-sF
OUTPUT LED's
INPUT Switches
OUTPUT LED's
Windows 7 [Running] - Oracle VM VirtualBox
pBlaze IDE C:\Users\prasoon\Desktop\PicoBlaze-work\asm\PbWbUART12C.PSM
File Edit View Settings Simulate Help
Status
Zero
Carry
Enable
Interrupt
Steady
Edge
Timer
Registers
UART_TX
UART_RX
I2C_SCL
I2C_SDA
PbWbUARTI2C.PSM

```
             MEM  "PICOBLAZE_I2C.mem"                                    ;
             VHDL "ROM_form.vhd", "PbWbUARTI2C.vhd"              ;       ;
                                                                         ;
                              ;----PORT Definitions----                  ;
                                                                         ;
$00          UART_TX       DSOUT   0                                     ; UART transmit port
$01          UART_RX       DSIN    1                                     ; UART receive port
$02          I2C_SCL       DSOUT   2                                     ; I2C serial clock:: I2C Master clocks I2C Slave
$03          I2C_SDA       DSIO    3                                     ; I2C serial data
                                                                         ;
                              ;----UART Definitions----                  ;
                                                                         ;
$01          TX_half_full    EQU   $01                                   ; Transmit FIFO half full 'bit0'
$02          TX_full         EQU   $02                                   ; Transmit FIFO full      'bit1'
$04          RX_half_full    EQU   $04                                   ; Receiver FIFO half full 'bit2'
$08          RX_full         EQU   $08                                   ; Receiver FIFO full      'bit3'
$10          RX_data_present EQU   $10                                   ; Data present            'bit4'
$08          UART_TXR_ADDR   EQU   $08                                   ;
$09          UART_SR_ADDR    EQU   $09                                   ;
$01          UART_SR_RX_F_FLAG  EQU  01                                  ;
$02          UART_SR_RX_HF_FLAG EQU  02                                  ;
$04          UART_SR_RX_DP_FLAG EQU  04                                  ;
$04          UART_SR_TX_F_FLAG  EQU  04                                  ;
$05          UART_SR_TX_HF_FLAG EQU  05                                  ;
```

```
Assembler Phase 2: constructing opcodes
Assembler Phase 3: building simulation objects
Assembler saving to MEM file: "C:\Users\prasoon\Desktop\PicoBlaze-work\asm\PICOBLAZE_I2C.mem"
Assembler reading VHDL skeleton: "C:\Users\prasoon\Desktop\PicoBlaze-work\asm\ROM_form.vhd"
Assembler saving to VHDL target: "C:\Users\prasoon\Desktop\PicoBlaze-work\asm\PbWbUARTI2C.vhd"
Program is Reset
```

Mode: Picoblaze-3 | 463: 38 | Instructions: 6 | Time : 150ns | PC: $0C2 | SP: 2 ($02) | Stack: $02 $06

Registers:

0	00	00	8
1	00	00	9
2	00	00	A
3	00	00	B
4	00	00	C
5	00	64	D
6	00	00	E
7	00	00	F

Figure A.2: pBlazIDE Simulator

KCPSM3	**pBlazIDE Instruction**
RETURN	RET
RETURN C	RET C
RETURN NC	RET NC
RETURN Z	RET Z
RETURN NZ	RET NZ
RETURNI ENABLE	RETI ENABLE
RETURNI DISABLE	RETI DISABLE
ADDCY	ADDC
SUBCY	SUBC
INPUT sX, (sY)	IN sX, sY(no parentheses)
INPUT sX, kk	IN sX, kk
OUTPUT sX, (sY)	OUT sX, sY (no parentheses)
OUTPUT sX, kk	OUT sX, kk
ENABLE INTERRUPT	EINT
DISABLE INTERRUPT	DINT
COMPARE	COMP
STORE sX, (sY)	STORE sX, sY(no parentheses)
FETCH sX, (sY)	FETCH sX, sY(no parentheses)

Table A.1: Instruction difference between KCPSM3 and pBlazIDE

A.2 Wishbone bus

The WISHBONE specification document [13] defines the WISHBONE bus as the System-on-Chip (SoC) architecture which is a portable interface for use with semiconductor IP cores. It is intended to be used as an internal bus for SoC applications with the aim of alleviating SoC integration problems by fostering design reuse. This objective is achieved by creating a common interface between IP cores [16]. It improves the portability, reliability of the system, and results in faster time-to-market for the end user [14]. However, the cores can be integrated more quickly and easily by the end user if a standard interconnection scheme is adopted. The WISHBONE bus helps the end user to accomplish all these objectives at one platform. Intended as a general purpose interface, WISHBONE bus defines a standard set of signals and bus cycles to be used between IP core modules making no attempt to regulate the application specific functions of the IP core.

A.2.1 Definition of Wishbone Bus Architecture(WBA) Interface

The issue of the physical structure of the Wishbone Bus Architecture is defined under the WBA Interface. There are two types of interfaces under the WISHBONE bus. These are called MASTER and SLAVE interfaces. Cores that help to generate bus cycles are identified as MASTER interfaces and the cores capable of receiving bus cycles are designated as SLAVE interfaces. As these interfaces range from single to complex architectures, there are a number of ways to connect between the MASTER and the SLAVE interfaces in a WISHBONE bus. These include Point-to-point interconnection, Data flow interconnection, Shared bus interconnection and Crossbar switch interconnection.

A.2.2 Selection and configuration of WISHBONE Bus Protocols

WISHBONE Bus protocols specify the manner in which transactions occur. These protocols include the implementation of arbitration mechanism in centralized or distributed bus arbiters. The issue of bus contention during the selection and configuration of WISHBONE bus protocol is settled or decided with the help of Handshaking protocol; deployment of different arbitration schemes such as Round Robin, TDMA, CDMA, Static Priority, Token Passing etc. All these strategies are application specific in WISHBONE Bus. These issues find a detailed elaboration under sub section 2.3 of this report.

A.2.3 Signal Mapping

Signal Mapping under WISHBONE bus is a process of associating Master and Slave devices in the bus Architecture. It also includes Bus Cycles. Although WISHBONE allows combining of all its signals between the MASTER and SLAVE interfaces yet each can do it at its own expense. WISHBONE signals have been grouped into three categories: Common Signals, Data Signals, and Bus Cycle Signals. The issues of WISHBONE signal types and WISHBONE bus cycles have been dealt with relevant details in subsection A.2 of this report.

A.2.4 Interfaces

Bus interfacing involves an electronic circuit that is responsible for driving or receiving data or power from a bus. So far as the interfacing in the WISHBONE bus is concerned, the on-chip WISHBONE bus architectures can be classified as: *Point-to-point interconnection, Shared bus interconnection, Crossbar switch interconnection, Data flow interconnection, and Off-chip interconnection.* As the Point-to-point interconnection is of interest to us, it is explained next.

Point to point interconnection A Point-to-Point interconnection supports direct connection of two participants that transfer data according to some handshake protocol. It implies that a single master has a direct connection to a single slave. This is the simplest way of connecting two IP cores and the traffic is controlled by the handshaking signals. As the Point-to-point INTERCON only supports connection of a single master interface and a single slave interface, its limitations do not make it suitable for SoC multi-device inter-connection.

A.2.5 WISHBONE Bus Cycles

WISHBONE supports three types of bus cycles; Single Read/Write cycles, Block Read/Write cycles and Read-Modify-Write cycles.

Single Read/Write Cycle A single read/write cycle means that only one data transfer is made each time. For example when a master wants to make a single read operation it presents a valid address on its address out port. Then it negates the write enable signal to show that a read operation is to be done. After that it asserts its cyclic out and strobe out signals to tell the slave that the transfer is ready to start. When the slave has noticed the assertion of the strobe and cyclic signals it places the right data on the data out port and asserts its acknowledged out signal. At the next clock edge the master will read the data and pull down its strobe out and cyclic out signal, which leads to that the slave negates its acknowledged out signal and thereby the transmission is complete.

Block Read/Write Cycle Block Read/Write cycles are used when a master wants to read or write multiple data arrays and works in a similar way as the single read/write cycle. The main difference is that the negation of the cyclic signal from the master does not occur until all data is transferred, instead the strobe and acknowledge signals control the flow of data arrays between the master and the slave. The master can put in wait states by pulling down its strobe signal, whereas the slave can put in wait states by pulling down it's acknowledge signal.

Read-Modify-Write (RMW) Cycle RMW cycle is used to avoid the possibility of two or more masters gaining access to the same slave. Such a possibility may occur in systems having multiple processors that share memories. It becomes important to ensure that they don't access the same memory at the same time. To prevent such a happening, a slave under use must be blocked. This is often done by assertion of a semaphore bit. If a master reads that a semaphore bit is asserted it knows that the slave is accessed by another master. The master has to use a RMW cycle in such a situation. First the master reads the semaphore bit and if it is cleared the master will assert it by writing something to it during the same transfer cycle. If the same procedure would be done by using single read and write operations another master could try to access the slave between the read and write operation, leading to that both masters would get access to the slave at the same time. In other words a RMW cycle gives a master the opportunity to both do a read and a write operation before any other master may use the bus and thereby avoiding a system crash.

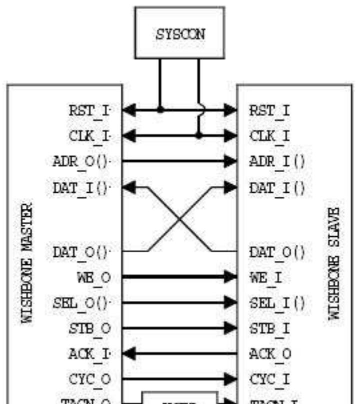


Figure A.3: Point to point interconnection with WISHBONE interface signals

Signal	**Name**	**Optional**	**Description**
Acknowledged Out	ACK_O	No	Acknowledged signal from the Master to Slave
Address Out/In	ADR_O/I()	No	Address Array
Clock In	CLK_I	No	System Clock for Wishbone Interface.
Error In/Out	ERR_I/O	Yes	Indicates an abnormal cycle termination occurred.
Data In/Out	DAT_I/O	No	Data input/output array, used to send/receive data.
Write Enable In	WE_I	No	Read or Write Signals. If asserted it is Write signal otherwise Read Signal.
Strobe In	STB_I	No	Indicates that the slave is selected The slave asserts either ACK_O, ERR_O
Strobe Out	STB_O	No	Handshaking Signal.
Address tag Out	TGA_O()	Yes	Contains Information about the address array.
Cycle tag type Out	TGC_O()	Yes	Contains Information about the transfer cycle.
Write Enable Out	WE_O	No	Shows if the transfer cycle is a Read or Write cycle.
Reset	RST_I	No	Reset signal

Table A.2: WISHBONE Signals

A.3 Design Validation

To validate the fabricated chip, most of the effort in this last stage of IC design & development is dedicated to validating electrical aspects of the design, or diagnosing systematic manufacturing defects, also a portion of the effort focuses on functional system validation. This trend is for the most part due to the increasing complexity of digital systems, which limits the verification coverage provided by traditional pre-silicon methodologies[17]. As a result, a number of functional bugs survive into manufactured silicon, and it is the job of post-silicon validation to detect and diagnose them so that they do not escape into the higher versions. The bugs in this category are often system-level bugs and rare corner-case situations buried deep in the design state space: since these problems encompass many design modules, they are difficult to identify with pre- silicon tools, characterized by limited scalability and performance.

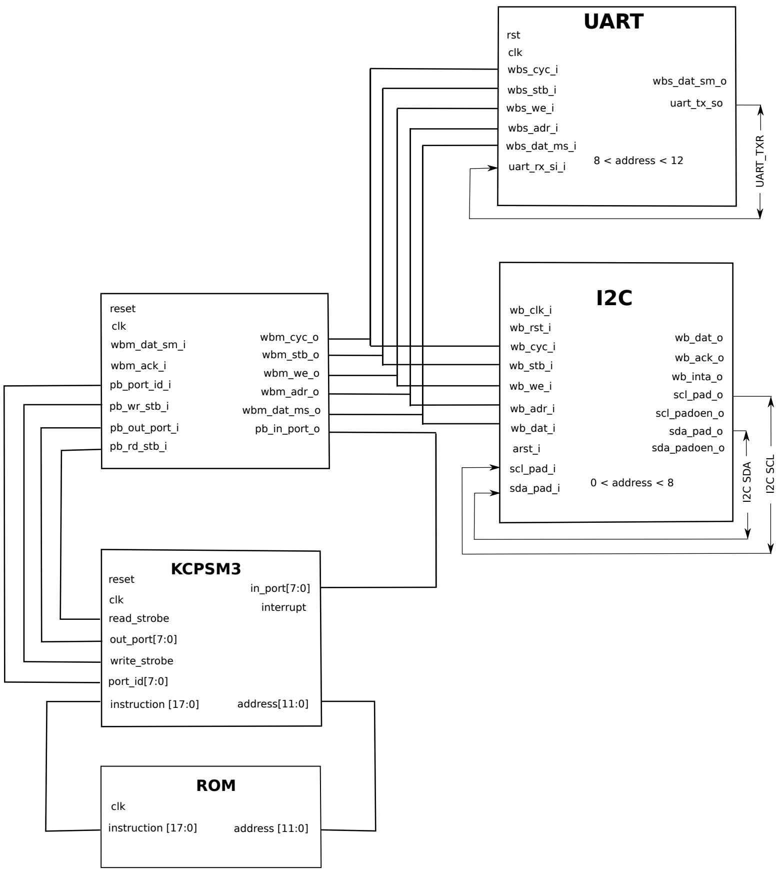


Figure A.4: PicoBlaze Wishbone configuration

Appendix B

Assembly Program

Wishbone slave (UART) module: Settings & subroutines:

- Set baud rate in initialize subroutine.
- Baud rate configuration:
  Baud limit = round(system clock frequency / (16 $\times$ *baudrate*)) - 1
  i. e. 9600 baud at 50 MHz system clock =>
  Baud limit = round(50.0E6 / (16 $\times$ 9600)) - 1 = 325 = 0x0145
- Use subroutines to read and write from UART slave.

Wishbone Read:

- Input cycle to setup wishbone address and control signals from PORT_ID & READ_STROBE.
- Input cycles to poll Wishbone peripheral acknowledgement using IN port.
- The very next INPUT cycle after acknowledgement contains valid Wishbone data from IN port.

Wishbone Write:

- Output cycle to setup Wishbone address, data and control signals from PORT_ID, OUT port and WRITE_STROBE.
- INPUT cycle to poll Wishbone peripheral acknowledgement using IN port. It is useful to note, atleast one output and one input cycle for a write.

The flow chart given in FigB.1 represents step by step process carried out in the assembly program which can move data between computer, processor and DUT.

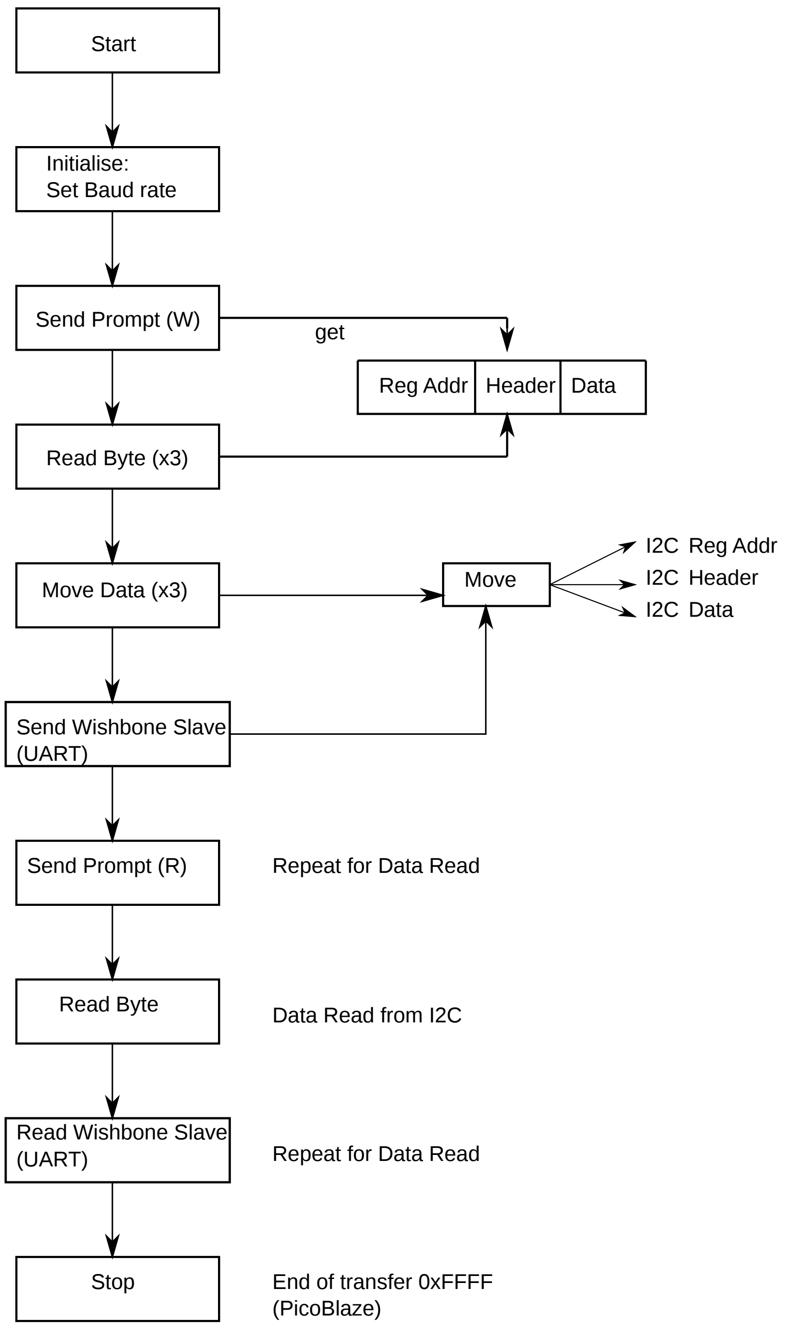


Figure B.1: Algorithm for Assembly program

```
MEM "PICOBLAZE_I2C.mem"
VHDL "ROM_form.vhd", "interface_pb_wb_uart_i2c.vhd"

;—-PORT Definitions—-

UART_TX DSOUT 0
;UART transmit port
UART_RX DSIN 1
;UART receive port
I2C_SCL DSIN 2
;I2C serial clock
I2C_SDA DSIO 3
;I2C serial data

;—-UART Definitions—-

TX_half_full EQU $01
;Transmit FIFO half full 'bit0'
TX_full EQU $02
;Transmit FIFO full 'bit1'
RX_half_full EQU $04
;Receiver FIFO half full 'bit2'
RX_full EQU $08
;Receiver FIFO full 'bit3'
RX_data_present EQU $10
;Data present 'bit4'
UART_TXR_ADDR EQU $00
UART_SR_ADDR EQU $01
UART_SR_RX_F_FLAG EQU $01
UART_SR_RX_HF_FLAG EQU $02
UART_SR_RX_DP_FLAG EQU $04
UART_SR_TX_F_FLAG EQU $10
UART_SR_TX_HF_FLAG EQU $20


;——-UART Variables——

UART_data EQU sF

;—-WISHBONE Variables—-

I2C_PRER_LO_ADDR EQU $00
I2C_PRER_HI_ADDR EQU $01
I2C_PRER_LO_VALUE EQU $63
I2C_PRER_HI_VALUE EQU $00
UART_BAUD_LO_ADDR EQU $02
UART_BAUD_HI_ADDR EQU $03
UART_BAUD_LO_9600_VALUE EQU $45
UART_BAUD_HI_9600_VALUE EQU $01
WB_addr EQU sE
```

```
WB_data EQU sD

WB_ACK_Flag EQU $1

I2C_addr EQU sC
I2C_data EQU sB
I2CSlAddr EQU sA
I2CRegAddr EQU s9
I2CRegData EQU s8

I2CM_Tran_Reg EQU 3
;I2C Transmit Register Address
I2C_TIP_Flag EQU 0
I2CM_Cmd_Reg EQU 4
;I2C Command Register
I2C_RxACK EQU 0
;—-Address origin—-

ORG 000


DINT INTERRUPT
;Interrupt disabled and i <- '0'

CALL I2C_INIT
;Configure WBS_I2C Master
CALL UART_INIT
;Configure WBS_UART

    I2C_INIT:
;Set Prescale register values- SCL
LOAD WB_addr, I2C_PRER_LO_ADDR
LOAD WB_data, I2C_PRER_LO_VALUE
CALL WB_Wr
LOAD WB_addr, I2C_PRER_HI_ADDR
LOAD WB_data, I2C_PRER_HI_VALUE
CALL WB_Wr
RET

UART_INIT:
LOAD WB_addr, UART_BAUD_LO_ADDR
LOAD WB_data, UART_BAUD_LO_9600_VALUE
CALL WB_Wr
LOAD WB_addr, UART_BAUD_HI_ADDR
LOAD WB_data, UART_BAUD_HI_9600_VALUE
CALL WB_Wr

CALL UART_Clr_Buff
RET
```

```
    Main:
; copy header, register address and data from PC-RS232-UART
CALL SendPrompt_Wr
; Prompt for DUT Write
CALL UART_Rd_Byte

LOAD I2CSlAddr, UART_data
; Read BYTE and Move to dedicated registers
CALL UART_Rd_Byte
LOAD I2CRegAddr, UART_data
; Read BYTE and Move to dedicated registers
CALL UART_Rd_Byte
LOAD I2CRegData, UART_data
; Read BYTE and Move to dedicated registers

; send header, reg addr and data to I2C DUT-

; Put (Header+Wr), Reg Addr, Data in transmit Register and

; set STA & WR bit in Command register

LOAD I2C_addr, I2CM_Tran_Reg
;**** Transmit Register Address ****

LOAD I2C_data, I2CSlAddr
; Transfer Slave Address + Wr (Bit)
CALL I2C_Wr_Byte
; Write BYTE: Data(I2C_data) will be written into specified address(I2C_addr)
LOAD I2C_addr, I2CM_Cmd_Reg

;**** Command Register Address **** LOAD I2C_data, $90

;**** Verify 0x90 **** Set STA & WR bits in Command Register
CALL I2C_Wr_Byte
; Write BYTE
CALL I2C_Wait_On_TIP
; Wait for TIP
; Read RxACK from Slave DUT in STATUS Register RxACK -> 0
CALL I2C_Read_ACK
; ACK from DUT(I2C)
LOAD I2C_addr, I2CM_Tran_Reg
LOAD I2C_data, I2CRegAddr
; Transfer Register Address
CALL I2C_Wr_Byte
; Write BYTE
CALL I2C_Wait_On_TIP
; Wait for TIP
LOAD I2C_addr, I2CM_Cmd_Reg
LOAD I2C_data, $90
```

```
; Set STA & WR bits in Command Register
CALL I2C_Wr_Byte
; Write BYTE
; Read RxACK from Slave DUT in STATUS Register RxACK -> 0
CALL I2C_Read_ACK
; ACK from DUT(I2C)
LOAD I2C_addr, I2CM_Tran_Reg
LOAD I2C_data, I2CRegData
;Transfer Register Data
CALL I2C_Wr_Byte
; Write BYTE
LOAD I2C_addr, I2CM_Cmd_Reg
LOAD I2C_data, $50
; Set STO & WR bits in Command Register
CALL I2C_Wr_Byte
; Write BYTE
CALL I2C_Wait_On_TIP
; Wait for TIP
; Read RxACK from Slave DUT in STATUS Register RxACK -> 0
CALL I2C_Read_ACK
; ACK from DUT(I2C)
CALL SendPrompt_Rd
; Prompt for DUT Read
LOAD I2C_addr, I2CM_Tran_Reg
LOAD I2C_data, I2CSladdr
; Transfer Slave Address + Wr (Bit)
CALL I2C_wr_byte
; Write BYTE
LOAD I2C_addr, I2CM_Cmd_Reg
LOAD I2C_data, $90
; Set STA & WR bits in Command Register
CALL I2C_Wr_Byte
; Write BYTE
CALL I2C_Wait_On_TIP
; Wait for TIP

; Read RxACK from Slave DUT in STATUS Register RxACK -> 0
CALL I2C_Read_ACK
; ACK from DUT(I2C)
LOAD I2C_addr, I2CM_Tran_Reg
LOAD I2C_data, I2CRegAddr
; Transfer Register Address
CALL I2C_Wr_Byte
; Write BYTE
LOAD I2C_addr, I2CM_Cmd_Reg
LOAD I2C_data, $90
; Set STA & WR bits in Command Register
CALL I2C_Wr_Byte
; Write BYTE
CALL I2C_Wait_On_TIP
```

```
; Wait for TIP
CALL I2C_Read_ACK
; ACK from DUT(I2C)
LOAD I2C_addr, I2CM_Tran_reg
LOAD I2C_data, I2CSladdr
;Transfer Slave Address + Rd (Bit)
CALL I2C_Wr_BYTE
; Write BYTE
LOAD I2C_addr, I2CM_Cmd_Reg
LOAD I2C_data, $90
; Set STA & WR bits in Command Register
CALL I2C_Wr_BYTE
CALL I2C_Wait_On_TIP
; Wait for TIP
; Write NACK from Master in Command Register ACK -> 1 (NACK) Set STO
bit
LOAD I2C_addr, I2CM_Cmd_Reg
LOAD I2C_data, $48
CALL I2C_Wr_BYTE

    JUMP Main
; Loop
;—- Subroutine :: UART/I2C Write —-

    UART_Clr_Buff:
LOAD WB_addr, UART_SR_ADDR
CALL WB_Rd
TEST WB_data, UART_SR_RX_DP_FLAG
RET Z
CALL UART_Rd_Byte
JUMP UART_Clr_Buff
SendPrompt_Wr:
; Subroutine Prompts for DUT: Header + Data direction (Write)
LOAD UART_data, ASCII_CR_CHAR
; Move data -ASCII Character: to be written in UART
CALL UART_Wr_Byte
; Subroutine to Write a BYTE of data
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_E_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_V_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_A_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_R_UC
```

```
CALL UART_Wr_Byte
LOAD UART_data, ASCII_US_CHAR
CALL UART_Wr_Byte
LOAD UART_data, ASCII_W_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_R_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_SP_CHAR
CALL UART_Wr_Byte
LOAD UART_data, ASCII_R_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_E_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_G_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_SP_CHAR
CALL UART_Wr_Byte
LOAD UART_data, ASCII_A_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_R_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_SP_CHAR
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_A_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_T_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_A_UC
CALL UART_Wr_Byte
RET

   SendPrompt_Rd:
; Subroutine Prompts for DUT: Header + Data direction (Read)
LOAD UART_data, ASCII_CR_CHAR
; Move data -ASCII Character: to be written in UART
CALL UART_Wr_Byte
; Subroutine to Write a BYTE of data
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_E_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_V_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_A_UC
```

```
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_R_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_US_CHAR
CALL UART_Wr_Byte
LOAD UART_data, ASCII_R_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_SP_CHAR
CALL UART_Wr_Byte
LOAD UART_data, ASCII_R_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_E_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_G_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_SP_CHAR
CALL UART_Wr_Byte
LOAD UART_data, ASCII_A_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_D_UC
CALL UART_Wr_Byte
LOAD UART_data, ASCII_R_UC
CALL UART_Wr_Byte
RET

UART_Rd_Byte:
; Blocking read byte from UART
LOAD WB_addr, UART_TXR_Addr
CALL WB_Rd
LOAD UART_data, WB_data
RET

    UART_Wr_Byte:
; Blocking write byte to UART
LOAD WB_addr, UART_TXR_Addr
LOAD WB_data, UART_data
CALL WB_Wr

WB_Rd:
; Wishbone read
CALL WB_Wait_On_ACK
IN WB_data, WB_addr
```

```
RET
WB_Wr:
; Wishbone write
OUT WB_data, WB_addr
CALL WB_Wait_On_ACK
RET
WB_Wait_On_ACK:
; Wait for Wishbone cycle to complete
IN WB_data, WB_addr
TEST WB_data, WB_ACK_Flag
; WB_ACK_Flag EQU $01
JUMP Z, WB_Wait_On_ACK
RET
I2C_Wr_byte:
; Write data
LOAD WB_addr, I2C_addr
LOAD WB_data, I2C_data
OUT WB_data, WB_addr
CALL WB_Wait_On_ACK
RET
I2C_Wait_On_TIP:
; Wait for TIP flag assert
IN WB_data, WB_addr
TEST WB_data, I2C_TIP_Flag
JUMP Z, I2C_Wait_On_TIP
RET
I2C_Read_ACK:
; Read Slave ACK
IN WB_data, WB_addr
TEST WB_data, I2C_RxACK
JUMP Z, I2C_Read_ACK
RET


;—- ASCII TABLE —-

   ASCII_NUL_CHAR EQU $00 ;NUL
ASCII_SOH_CHAR EQU $01 ;SOH
ASCII_STX_CHAR EQU $02 ;STX
ASCII_ETX_CHAR EQU $03 ;ETX
ASCII_EOT_CHAR EQU $04 ;EOT
ASCII_ENQ_CHAR EQU $05 ;ENQ
ASCII_ACK_CHAR EQU $06 ;ACK
ASCII_BEL_CHAR EQU $07 ;BEL
ASCII_BS_CHAR EQU $08 ;BS
ASCII_TAB_CHAR EQU $09 ;TAB
ASCII_LF_CHAR EQU $0A ;LF
ASCII_VT_CHAR EQU $0B ;VT
ASCII_FF_CHAR EQU $0C ;FF
ASCII_CR_CHAR EQU $0D ;CR
```

```
ASCII_SO_CHAR EQU $0E ;SO
ASCII_SI_CHAR EQU $0F ;SI
ASCII_DLE_CHAR EQU $10 ;DLE
ASCII_DC1_CHAR EQU $11 ;DC1
ASCII_DC2_CHAR EQU $12 ;DC2
ASCII_DC3_CHAR EQU $13 ;DC3
ASCII_DC4_CHAR EQU $14 ;DC4
ASCII_NAK_CHAR EQU $15 ;NAK
ASCII_SYN_CHAR EQU $16 ;SYN
ASCII_ETB_CHAR EQU $17 ;ETB
ASCII_CAN_CHAR EQU $18 ;CAN
ASCII_EM_CHAR EQU $19 ;EM
ASCII_SUB_CHAR EQU $1A ;SUB
ASCII_ESC_CHAR EQU $1B ;ESC
ASCII_FS_CHAR EQU $1C ;FS
ASCII_GS_CHAR EQU $1D ;GS
ASCII_RS_CHAR EQU $1E ;RS
ASCII_US_CHAR EQU $1F ;US
ASCII_SP_CHAR EQU $20 ;SP
ASCII_EXCLAMATION_MARK_SIGN EQU $21 ;!
ASCII_DOUBLE_QUOTE_SIGN EQU $22 ;"
ASCII_NUMBER_SIGN EQU $23 ;#
ASCII_DOLLAR_SIGN EQU $24 ;$
ASCII_PERCENT_SIGN EQU $25 ;%
ASCII_AMPERSAND_SIGN EQU $26 ;&
ASCII_SINGLE_QUOTE_SIGN EQU $27 ;´
ASCII_OPN_PARENTHESIS_SIGN EQU $28 ;(
ASCII_CLS_PARENTHESIS_SIGN EQU $29 ;)
ASCII_ASTERISK_SIGN EQU $2A ;*
ASCII_PLUS_SIGN EQU $2B ;+
ASCII_COMMA_SIGN EQU $2C ;EQU
ASCII_MINUS_SIGN EQU $2D ;-
ASCII_DOT_SIGN EQU $2E ;.
ASCII_SLASH_SIGN EQU $2F ;/
ASCII_0_DIGIT EQU $30 ;0
ASCII_1_DIGIT EQU $31 ;1
ASCII_2_DIGIT EQU $32 ;2
ASCII_3_DIGIT EQU $33 ;3
ASCII_4_DIGIT EQU $34 ;4
ASCII_5_DIGIT EQU $35 ;5
ASCII_6_DIGIT EQU $36 ;6
ASCII_7_DIGIT EQU $37 ;7
ASCII_8_DIGIT EQU $38 ;8
ASCII_9_DIGIT EQU $39 ;9
ASCII_COLON_SIGN EQU $3A ;:
ASCII_SEMICOLON_SIGN EQU $3B ;;
ASCII_LESS_THAN_SIGN EQU $3C ;<
ASCII_EQUAL_SIGN EQU $3D ;=
ASCII_GREATER_THAN_SIGN EQU $3E ;>
ASCII_QUESTION_MARK_SIGN EQU $3F ;?
```

```
ASCII_AT_SIGN EQU $40 ;@
ASCII_A_UC EQU $41 ;A
ASCII_B_UC EQU $42 ;B
ASCII_C_UC EQU $43 ;C
ASCII_D_UC EQU $44 ;D
ASCII_E_UC EQU $45 ;E
ASCII_F_UC EQU $46 ;F
ASCII_G_UC EQU $47 ;G
ASCII_H_UC EQU $48 ;H
ASCII_I_UC EQU $49 ;I
ASCII_J_UC EQU $4A ;J
ASCII_K_UC EQU $4B ;K
ASCII_L_UC EQU $4C ;L
ASCII_M_UC EQU $4D ;M
ASCII_N_UC EQU $4E ;N
ASCII_O_UC EQU $4F ;O
ASCII_P_UC EQU $50 ;P
ASCII_Q_UC EQU $51 ;Q
ASCII_R_UC EQU $52 ;R
ASCII_S_UC EQU $53 ;S
ASCII_T_UC EQU $54 ;T
ASCII_U_UC EQU $55 ;U
ASCII_V_UC EQU $56 ;V
ASCII_W_UC EQU $57 ;W
ASCII_X_UC EQU $58 ;X
ASCII_Y_UC EQU $59 ;Y
ASCII_Z_UC EQU $5A ; Z
ASCII_OPN_BRACKET_SIGN EQU $5B ;[
ASCII_BACKSLASH_SIGN EQU $5C ;
ASCII_CLS_BRACKET_SIGN EQU $5D ;]
ASCII_CARET_SIGN EQU $5E ;ˆ
ASCII_UNDERSCORE_SIGN EQU $5F ;_
ASCII_ACCENT_SIGN EQU $60 ;‘
ASCII_A_LC EQU $61 ;a
ASCII_B_LC EQU $62 ;b
ASCII_C_LC EQU $63 ;c
ASCII_D_LC EQU $64 ;d
ASCII_E_LC EQU $65 ;e
ASCII_F_LC EQU $66 ;f
ASCII_G_LC EQU $67 ;g
ASCII_H_LC EQU $68 ;h
ASCII_I_LC EQU $69 ;i
ASCII_J_LC EQU $6A ;j
ASCII_K_LC EQU $6B ;k
ASCII_L_LC EQU $6C ;l
ASCII_M_LC EQU $6D ;m
ASCII_N_LC EQU $6E ;n
ASCII_O_LC EQU $6F ;o
ASCII_P_LC EQU $70 ;p
ASCII_Q_LC EQU $71 ;q
```

```
ASCII_R_LC EQU $72 ;r
ASCII_S_LC EQU $73 ;s
ASCII_T_LC EQU $74 ;t
ASCII_U_LC EQU $75 ;u
ASCII_V_LC EQU $76 ;v
ASCII_W_LC EQU $77 ;w
ASCII_X_LC EQU $78 ;x
ASCII_Y_LC EQU $79 ;y
ASCII_Z_LC EQU $7A ;z
ASCII_OPN_BRACE_SIGN EQU $7B ;
ASCII_VERTICAL_BAR_SIGN EQU $7C ;|
ASCII_CLS_BRACE_SIGN EQU $7D ;
ASCII_TILDE_SIGN EQU $7E ;˜
ASCII_DEL_CHAR EQU $7F ;DEL
```

Appendix C

Results- Test run

C.1 The prototype

With the trouble in getting the automatic verification to run without errors, and pressure of turn around time I switched to using FPGA as a way to verify ASIC. FPGAS can emulate a design at speeds close to their actual operating frequencies, enabling kinds of verification that would otherwise not be possible, such as in-circuit emulation, or runs that go deep into the state space of the design that would take hours or even days to reach in simulation.

FPGA prototyping usually involves a board with a number of FPGA mounted on it with a fixed configuration and may also include other pre-implemented hardware that can be connected in as well. There are two possibilities with these prototypes, either create a custom prototyping board that may match your application domain, or buy a commercial offering that can allow arbitrary systems to be built up from FPGA modules, or from other specialized modules such as memory, processor or peripheral modules. Here I have implemented a I^2C Master core which controls the data transfer between the devices. The entire exercise was worth exploring as you face many problems. For example, how many clocks does your design need? What about the number of registers or amount of memory. All of the resources are limited within an FPGA and this can present serious problems with the mapping. Many of these issues will be lessened with a commercial emulator, but not removed completely.

The complete assembly of Master controller and Slave(DUT) is shown in figC.1 A sample run of serial data and serial clock, between controller and slave as probed with an oscilloscope is is shown with the screen capture in figC.2.

C.2 Prototype- Test runs

Several test runs were run on the Master-Slave assembly.

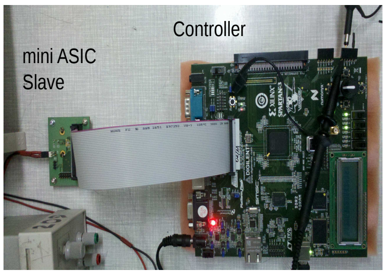


Figure C.1: Controller & Slave

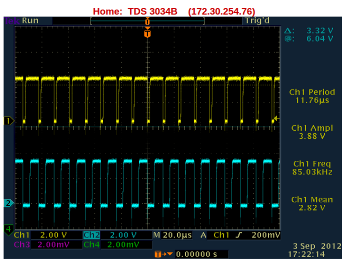


Figure C.2: SDA & SCL

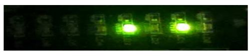


Figure C.3: Test Run1: Data = 1010[U]

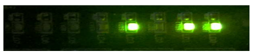


Figure C.4: Test Run2: Data = 1011[L]

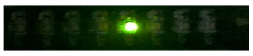


Figure C.5: Test Run3: Data = 1000[U]

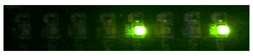


Figure C.6: Test Run4: Data = 1001[L]

Appendix D

MCPAD Activities

D.1 Network Events

The MC-PAD network events that I have attended are the following:

- The first network training event on electronics- 17th - 19th September 09 at AGH UST, Cracow.
- Detector Simulation and Data Analysis (Geant4, ROOT)- 28th - 30th January 2010 organized by University of Hamburg and DESY.
- MC-PAD network training event on Processing and Radiation Hardness of Solid State Detectors- 27th to 29th September 2010 at Jozef-Stefan-Institute in Ljubljana.
  Note- This training event was combined with the MC-PAD midterm review.
- MC-PAD training event on Gaseous and Photodetecors- 16th to 18th March 2011 at CERN, Geneva
- MC-PAD training on complementary aspects, such as science in industry, job interviews etc- 8th to 10th November 2011 at Paul Scherrer Institute, Villigen.
- MC-PAD training on Calorimetry in high energy physics and spin-off in the medical field- 19th to 21st March 2012 at GSI, Darmstadt.
- MC-PAD closing meeting at Laboratori Nazionali di Frascati from 20th - 22nd September 2012.
  Note- Closing term review.

D.2 FCAL Collaboration

The FCAL Collaboration meetings and workshops that I have attended are the following:

- 18th FCAL Workshop Predeal, Romania [30 May-1 June 2011]

- FCAL workshop on High precision measurements of luminosity at future linear colliders and polarization of lepton beams TAU, Tel Aviv [3-5 Oct 2010]
- FCAL Collaboration Meeting at IFJ PAN, Cracow [12-13 April 2010]
- FCAL Collaboration Meeting at CERN, Geneva [21-22 October 2009]

D.3 IDESA

I had attended IDESA training programs for enhancing IC design skills. The didactic training events that I attended are:

- Advanced analog implementation flow:

The 5-day analog implementation flow course covered modeling issues, hand calculation versus simulation accuracy, transistor level and behavioral level design, analog cell trimming using digital functions, mixed mode simulation, mismatch and yield modeling and analysis, and analog modeling and circuit optimization.

- Advanced digital physical implementation flow:

The 5-day digital physical implementation flow will start by introducing the challenges for 90 nm SoC design and the design environment and too chain. The course will proceed with digital synthesis, leakage-aware design, design planning and floorplanning, library analysis and management. It will focus upon low power design flow covering techniques to minimise dynamic and static power consumption, multiple clock tree synthesis, test and multimode and multicorner optimisation. IR-drop analysis, dynamic power analysis sign-off and design finishing and layout verification will be covered. Extensive hands-on labs are part of the course.

D.4 Publications

- **Infrastructure for Detector Research and Development towards the ILC.**
  2012 EUDET

- **A power scalable 10-bit pipeline ADC for Luminosity Detector at ILC.**
  2011 JINST 6 P01004

- **Forward instrumentation for ILC detectors.**
  2010 JINST 5 P12002

- **A Slow Control Interface for ADC ASIC**
  Conference Proceedings of 18th FCAL Collaboration Workshop, Predeal, Romania (2011)

Appendix E

Acknowledgement

This MC-PAD Marie Curie Fellowship has been a great experience for working on very stimulating topics, challenging problems, and for me perhaps the most important to meet and collaborate with extraordinary people, learn new language, experience new cultures and learn to live in a different climate. For three years I have lived in Krakow but indeed there is much more to explore.

First and foremost, many thanks go to Prof. Marek IDZIK and Dr. Krzystof ŚWIENTEK for supervising my project and teaching me a lot of new stuff, for guidance and support, for all the fruitful discussions, and for the company during the conference trips. I am grateful to them for letting me pursue my research interests with sufficient freedom, while being there to guide me all the same. Also, I am grateful to Dr. Andrzej SKOCZEN for his kind support during my stay at WFiIS, AGH University of Science and Technology. I am also grateful to Dr. Christian JORAM and Ms Veronique WEDLAKE, CERN for administrative and other assistance provided during the fellowship period. My greetings go to all other supervisors and colleagues in MCPAD and FCAL collaboration network for their company provided during network meetings and workshops. I would like to thank my officemates Imran, Jonathan, Michal and Szymon for the good times we had.

Thanks, folks!

Prasoon Ambalathankandy

Bibliography

[1] *IEEE Standard for Terminology and Test Methods for Analog-to-Digital Converters*
IEEE Std 1241-2000

[2] M. Idzik, K. Świentek, Sz. Kulis *Development of pipeline ADC for the luminosity detector at ILC* MIXDES 2008

[3] M. Idzik, K. Świentek, T. Fiutowski, S. Kulis, P. Ambalathankandy *A power scalable 10-bit pipeline ADC for luminosity detector at ILC* JINST 2011

[4] *SPI Block Guide* V03.06, Freescale Semiconductor

[5] *Philips I^2C Application notes, NXP(2008)* NXP Semiconductors

[6] *Philips I^2C handbook(1999)* NXP Semiconductors

[7] *Xilinx OPB-IIC bus interface product specification* XILINX

[8] *Xilinx Spartan3AN user guide* XILINX

[9] *MCP23008/MCP23S08 Data Sheet* Microchip Technology

[10] *STCN75 Data Sheet* STMicroelectronics

[11] Mediatronix, *PicoBlaze IDE* `http://www.mediatronix.com/pBlazeIDE.htm`

[12] *IEEE Standard The official standard for Verilog 2001* IEEE Std 1364-2001

[13] Richard Herveille, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, rev. version: B4, 2010* By Open Cores Organization, p.7, 2010.

[14] Richard Herveille, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores* Open Cores Organization, rev. version: B4, 2010.

[15] Richard Herveille, *Combining WISHBONE Interfaces Signals: Application note* Rev. 0.2 Preliminary, p. 2, April 18, 2001

[16] *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores* rev. version: B4, p.92, 2010.

[17] Miron Abramovici and Paul Bradley, Dafca *A New Approach to In-System Silicon Validation and Debug*

[18] IDESA, `http://www.idesa-training.org/`