



Norwegian University of  
Science and Technology

# Ultra-low power serial communication for Internet of Things

**Martin Lesund**

Master of Science in Electronics

Submission date: July 2017

Supervisor: Snorre Aunet, IES

Co-supervisor: Sigve Tjora, Disruptive Technologies

Norwegian University of Science and Technology  
Department of Electronic Systems



# Master Assignment

**Candidate Name:**

Martin Lesund

**Assignment Title:**

Ultra-low power serial communication for Internet of Things

**Assignment Description:**

In ultra low power systems, the energy used for inter die communication is an important contributor to power dissipation. Many sensors communicate with the host system over low pin count serial buses like I2C, SPI or single wire buses. Careful analysis and optimized design of building blocks like I2C or SPI masters are needed to minimize the current consumption. The assignment will be:

- Study literature and identify solutions for low power serial communication
- Investigate tradeoffs and compare the identified solutions
- Select best approach and design key building blocks, e.g. for an ultra-low power I2C master

The assignment could be done as a purely digital assignment or as a small mixed signal assignment. The design should target implementation in an ASIC.

**Assignment Proposer/Co-Supervisor:**

Sigve Tjora - Disruptive Technologies AS

**Supervisor:**

Snorre Aunet





## Abstract

This master thesis is about finding a suitable bus protocol to use on an ultra low power IoT device. The bus protocols I<sup>2</sup>C, 1-Wire, SPI, and MBus is compared by different parameters and previous power measurement experiments. Both Mbus and SPI uses substantially less power compared to the other interfaces. The SPI protocol is suggested as a good communication solution, and for this reason an SPI master design is made from scratch. The RTL code is written in VHDL and different energy techniques is used to make it more suitable for ultra low power devices. The SPI master design is implemented on an FPGA on the "Zedboard development kit" which is connected to an external temperature sensor called "LM74". Then a custom test application runs on the "Zedboard" to evaluate the functionality of the SPI master. The data read from the sensor is outputted on a serial terminal to the computer. The SPI master design functioned as expected and got the correct data from the temperature sensor.

This thesis contains material that sometimes can be overlooked in the design process, namely how the communication is done between integrated circuits. There are several solutions that can do this, and some communication solutions is better suited than others. A good communication solution can increase the battery life of an IoT device. This is because the energy used for inter die communication is an important contributor to power dissipation, which is important to keep as low as possible when it comes to ultra low power systems.

This thesis is a collection of some of the most common bus protocols in serial communication, and makes arguments why the SPI interface can be the most suitable for your design. To further prove this argument an SPI master design is made and tested on an FPGA with a temperature sensor to verify the functionality of the design.



## Sammendrag

(Abstract in Norwegian)

Denne masteroppgaven handler om å finne en egnet bus protokoll til bruk for en ultra lav effekt IoT enhet. Bus protokollene I<sup>2</sup>C, 1-Wire, SPI og Mbus blir sammenlignet med hverandre fra forskjellige parametre og tidligere effektmålings eksperiment. SPI protokollen blir foreslått som en god kommunikasjons metode, og av den grunn blir det designet en egen SPI master. RTL koden er skrevet i VHDL og forskjellige energisparende teknikker er brukt for å gjøre modulen tilpasset for en ultra lav effekt enhet. SPI masteren blir implementert på en FPGA på "Zedboard development kit", som er koblet til en ekstern temperatur sensor navngitt "LM74". Deretter testes funksjonaliteten av designet for SPI masteren med et eget test program, og temperaturen leses fra sensoren og blir sendt til en seriell terminal på datamaskinen. Designet av SPI masteren fungerte som forventet og ga ut riktig temperatur fra sensoren.

Denne masteroppgaven omhandler et tema som noen ganger kan bli oversett i design prosessen, nemlig hvordan kommunikasjonen burde gjøres mellom enhetene i et system. Det er mange forskjellige måter å gjøre dette på, og noen løsninger er bedre egnet enn andre. Forbedret kommunikasjon mellom enhetene kan for eksempel føre til en lengre batterilevetid for en IoT enhet. Dette er fordi energien som er brukt i kommunikasjonen mellom enheter er en viktig bidragsyter til energitap. Det er viktig å holde energitapet så lavt som mulig når det kommer til ultra lave effekt systemer.

Denne masteroppgaven er en samling av noen av de mest brukte bus protokollene for seriell kommunikasjon, og gjør argumenter for at SPI grensesnittet kan være den beste løsningen for ditt design. For å ytterligere prøve å bevise dette, blir en SPI master laget og testet på en FPGA med en temperatur sensor.



## Preface

This is a master thesis in "Design of Digital Systems" with the course code TFE4915, carried out at the Norwegian University of Science and Technology in Trondheim. This is the last project for the masters program in Electronics (program code MIEL). The master thesis was carried out during the spring semester of 2017, and is a continuation of a specialization project under the same name (course code TFE4520). This master was done in cooperation with the company "Disruptive Technologies AS". It was supervised by professor Snorre Aunet at NTNU and Sigve Tjora at Disruptive Technologies. This master is executed and documented by the student Martin Lesund.

Trondheim, 2017-07-13

A handwritten signature in black ink, reading "Martin Lesund". The signature is written in a cursive style with a horizontal line underlining the name.

Martin Lesund



## **Acknowledgment**

I would like to thank the following persons for their great help during this master thesis. I would like to thank my supervisor Snorre Aunet for giving me guidance and support throughout this master. I would also like to thank Sigve Tjora from Disruptive Technologies for helping me when I had questions, and giving me tips during this period. I would also like to thank senior engineer at NTNU Terje Mathiesen, who made a breakout board for the LM74 sensor, which made it usable for testing my design.

M.L.





## Abbreviations

**IoT** Internet of Things

**IC** Integrated Circuit

**FPGA** Field-programmable gate array

**SoC** System on Chip

**LSB** Least Significant Bit

**MSB** Most Significant Bit

**SDK** Software Development Kit

$\mu$ **C** Microcontroller

**GPIO** General Purpose Input/Output

**I/F** Interface

**DAC** Digital to Analog Converter

**ADC** Analog to Digital Converter

**LUT** Look Up Table

**ASIC** Application-Specific Integrated Circuit

**RTL** Register Transfer Level

**FSM** Finite-State Machine



# Contents

Master Assignment . . . . .	i
Abstract . . . . .	iii
Sammendrag . . . . .	v
Preface . . . . .	vii
Acknowledgment . . . . .	ix
Abbreviations . . . . .	xi
List of Figures . . . . .	xvii
List of Tables . . . . .	xxi
<b>1 Introduction</b>	<b>1</b>
1.1 Topic . . . . .	1
1.2 Problem Description . . . . .	2
1.3 Objectives . . . . .	2
1.4 Motivation . . . . .	3
1.5 Thesis Outline . . . . .	4
<b>2 Background Theory</b>	<b>5</b>
2.1 Communication . . . . .	5
2.2 Bus Communication . . . . .	7
2.3 Interconnects . . . . .	9
2.3.1 Power Savings in Interconnects . . . . .	12

2.4	Coding Techniques . . . . .	13
2.5	Serial communication protocols . . . . .	13
2.5.1	I <sup>2</sup> C . . . . .	14
2.5.2	1-Wire . . . . .	15
2.5.3	SPI . . . . .	16
2.5.4	MBus . . . . .	19
<b>3</b>	<b>Comparing the Bus Protocols I<sup>2</sup>C, 1-Wire, SPI and MBus</b>	<b>21</b>
3.1	Comparison of bus protocols . . . . .	21
3.2	Previous Power Measurement Experiments . . . . .	23
3.3	Suggestion for solution . . . . .	24
<b>4</b>	<b>Design Process of an SPI Master</b>	<b>29</b>
4.1	SPI Specification . . . . .	30
4.1.1	Port List . . . . .	30
4.1.2	SPI Architecture . . . . .	31
4.1.3	SPI FSM . . . . .	33
4.1.4	SPI Addressing . . . . .	35
4.2	SPI Simulation . . . . .	35
4.3	SPI Registers . . . . .	39
4.3.1	Implementing the Registers . . . . .	40
4.3.2	AXI lite Connection . . . . .	42
<b>5</b>	<b>Results for the SPI Experiment</b>	<b>45</b>
5.1	SPI with LM74 . . . . .	45
5.2	Power Saving Techniques . . . . .	48
<b>6</b>	<b>Discussion</b>	<b>49</b>
6.1	Power Consumption . . . . .	49

<i>CONTENTS</i>	xv
6.2 SPI with LM74 . . . . .	49
6.3 Power Saving Techniques . . . . .	50
6.4 Limitations of the Findings . . . . .	51
<b>7 Conclusion</b>	<b>53</b>
7.1 Recommendations for Further Work . . . . .	54
<b>References</b>	<b>55</b>
<b>A LM74 Temperature Sensor</b>	<b>61</b>
A.1 LM74 . . . . .	61
A.2 Serial Bus Interface . . . . .	62
A.3 Interfacing with 3-wire SPI . . . . .	63
<b>B Connecting to the Zedboard</b>	<b>65</b>
<b>C Axi Lite core</b>	<b>69</b>
C.1 Creating a new IP . . . . .	76
<b>D SPI Master code</b>	<b>89</b>
D.1 SPI Master . . . . .	89
D.2 Test bench . . . . .	96
<b>E AXI vhdl files</b>	<b>103</b>
E.1 my_spi_v1_0.vhd . . . . .	103
E.2 my_spi_v1_0_S00_AXI.vhd . . . . .	106
E.3 C Code Test Program . . . . .	117



# List of Figures

- 1.1 Growth of IoT in the 2000s. Source: [11] . . . . . 3
- 2.1 Example of Parallel interface . . . . . 6
- 2.2 Example of Serial interface . . . . . 6
- 2.3 Example of bus-based communication architecture. Adapted from source:  
[27] . . . . . 8
- 2.4 An example of bidirectional bus lines. Adapted from Source: [27] . . . . . 8
- 2.5 Single and multiple microstrip wire. Source: [29] . . . . . 10
- 2.6 Using coding on a bus. Adapted from source: [29] . . . . . 12
- 2.7 A simple I<sup>2</sup>C example. Adapted from source:[38] . . . . . 14
- 2.8 A microcontroller with multiple 1-Wire temperature sensors. Adapted  
from source:[28] . . . . . 15
- 2.9 A simple SPI example. Adapted from source:[20] . . . . . 17
- 2.10 Timing diagram of SPI modes 0 and 1 . . . . . 18
- 2.11 Timing diagram of SPI modes 2 and 3 . . . . . 18
- 2.12 MBus physical topology for stepped 3D stacking with wirebonding. Source:  
[26] . . . . . 20
- 3.1 Bus overhead. Adapted from source: [26] . . . . . 25
- 3.2 Total Power Draw. Adapted from source: [26] . . . . . 26

4.1	Flowchart of the design process . . . . .	29
4.2	This is the architecture of the design . . . . .	32
4.3	This is the FSM of the SPI Master . . . . .	33
4.4	FSM VHDL code . . . . .	34
4.5	Simulation of the SPI Master (Mode 2) . . . . .	38
4.6	Simulation of the SPI Master (Mode3) . . . . .	38
4.7	VHDL code of the registers . . . . .	41
4.8	Block diagram of the AXI interconnects with the SPI Master and PS . .	43
4.9	Block diagram of the SPI master . . . . .	44
5.1	Testing environment . . . . .	46
5.2	Read from terminal . . . . .	46
5.3	Validating temperature . . . . .	47
5.4	Temperature increasing . . . . .	47
5.5	Utilization Report of Binary Encoding . . . . .	48
5.6	Utilization Report of Gray Encoding . . . . .	48
A.1	The LM74 chip . . . . .	62
A.2	The LM74 on a breakout board . . . . .	62
A.3	SOIC - Top View. Source [35] . . . . .	62
A.4	LM74 digital input control using $\mu$ C's GPIO. Source [35] . . . . .	63
A.5	The LM74 on a breakout board . . . . .	63
B.1	The Zedboard Connections . . . . .	65
B.2	The Zedboard Connections . . . . .	66
B.3	Pmod schematic. Source [8] . . . . .	67
C.1	Create a new project . . . . .	70
C.2	Pick a suitable project name . . . . .	70



C.3 Choose RTL Project . . . . .	71
C.4 Select what platform you are using . . . . .	71
C.5 GUI of Vivado . . . . .	72
C.6 Creating a block design . . . . .	72
C.7 Add a new IP . . . . .	73
C.8 Find the Zynq IP . . . . .	73
C.9 Zynq added to design . . . . .	73
C.10 Re-Customize IP . . . . .	74
C.11 PS-PL Configuration panel . . . . .	75
C.12 Connect the wire highlighted in the block design . . . . .	75
C.13 Choose AXI4 peripheral . . . . .	76
C.14 Default settings for AXI Lite . . . . .	77
C.15 Choosing the SPImaster.vhd . . . . .	78
C.16 Package IP panel . . . . .	79
C.17 Add your custom IP . . . . .	79
C.18 Updated block diagram . . . . .	80
C.19 Block diagram after connection automation . . . . .	80
C.20 Make inputs/outputs external . . . . .	81
C.21 Generate output products . . . . .	81
C.22 Creating HDL Wrapper . . . . .	82
C.23 Bitstream finished . . . . .	82
C.24 I/O planning section . . . . .	83
C.25 I/O addressing . . . . .	83
C.26 Export Hardware . . . . .	84
C.27 Make a New Project . . . . .	84
C.28 Src folder for the project . . . . .	85
C.29 Copy "ps7_init.h" into your src folder . . . . .	85

C.30 Program FPGA . . . . .	85
C.31 Select test file . . . . .	86
C.32 Serial terminal port options . . . . .	86
C.33 Read data from UART . . . . .	87

# List of Tables

2.1	The different SPI modes . . . . .	17
3.1	Comparison of the serial interfaces. Adapted from source: [26] and [21]	22
3.2	Energy consumption for evaluated serial interfaces [21] . . . . .	24
4.1	SPI signals in the design . . . . .	31
4.2	Binary vs. Gray Encoding . . . . .	34
4.3	SPI Master addressing . . . . .	35
4.4	slv_reg0, D31 to D16 . . . . .	39
4.5	slv_reg0, D15 to D0 . . . . .	39
4.6	spi_to_ps, D31 to D16 . . . . .	40
4.7	spi_to_ps, D15 to D0 . . . . .	40
B.1	Pmod Connection. Adapted from source [3] . . . . .	66



# Chapter 1

## Introduction

This chapter contains information about why bus communication is important for ultra low power devices. It also contains the problem description and the objectives of this master thesis as well the motivation for this research. At the end of the chapter a thesis outline is listed.

### 1.1 Topic

The energy used for inter die communication is an important contributor to power dissipation. This is important to keep as low as possible for ultra low power systems to increase battery life if it's used as a power supply. These systems could include sensors which usually communicate with the host system over serial buses like 1-Wire, I<sup>2</sup>C or SPI buses. Which bus you chose can have an impact on the overall power dissipation. The interconnects between devices are generally responsible for a substantial fraction of the total power consumption [16]. The trend is that systems are getting more complex with additional features, size requirements and power requirements. Therefore, the method of communication can have a great impact for power savings and prolonged battery life. In the past, the serial bus you chose usually wasn't

as much thought through, because of lower requirements for the system. They prioritized the parts that were the most power hungry, not the parts where power dissipation was relatively lower. Today, for ultra low power systems we have to optimize everything to save as much power as possible, and serial buses has an impact in the overall power dissipation.

## 1.2 Problem Description

The difference between the serial buses can have a respective impact on the power dissipation for a system. We must identify what can be done to minimize the current consumption for a bus master. Then we have to compare the most common serial busses for several parameters e.g. difference between complexity, pins used and robustness. After that we use the most suited interface for an ultra low power system, and design the master module for this interface. The serial communication design solution is directed towards a System on Chip (SoC).

## 1.3 Objectives

The objectives for this master thesis is as follows:

- Identify solutions for low power serial communication
- Investigate tradeoffs and compare the identified solutions
- Select the best approach and design key building blocks for an ultra low bus master

## 1.4 Motivation

In the last few years the interest for devices connected to the Internet has rapidly increased, under the term most known as the "Internet of Things" (IoT). We are surrounded by small electronic devices everywhere which usually runs on battery as their power supply. This trend of increasing numbers of IoT devices doesn't seem to stagnate. "Cisco Internet Business Solutions Group" predicted in 2010, that there will be over 50 billion<sup>1</sup> connected devices to the Internet in 2020 [11]. The Figure 1.1 shows the estimated growth of connected IoT devices. This relatively high estimate can also be interpreted to create market efficiency by guiding companies to make choices to for example invest and enter this new era of IoT.

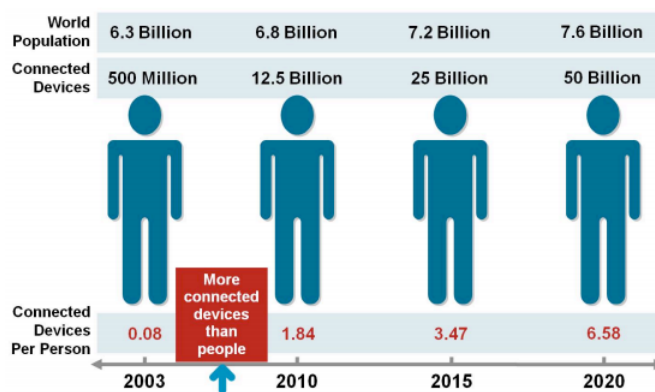


Figure 1.1: Growth of IoT in the 2000s. Source: [11]

The energy required for operations from machines build around embedded systems are increasing [14]. Systems are getting more complex and the energy requirements increases. Reduction of energy consumption in core processors or controllers, peripherals and communication interfaces can increase the lifetime of a battery substantially [39]. This is why choosing the right bus protocol for your system is important and has the potential of reducing the energy consumption considerably.

<sup>1</sup>This number has been reduced to around 20-30 billion by new estimates from Gartner, Ericsson and others [24]

## 1.5 Thesis Outline

The structure of this thesis is as follows:

**Chapter 2 - Background Theory:** This chapter presents relevant information behind basic communication styles, bus communication and interconnects between devices in a system. This chapter also include power saving techniques such as coding techniques, and ends with information about the I<sup>2</sup>C, 1-Wire, SPI and MBus protocols.

**Chapter 3 - Comparing the Bus Protocols I<sup>2</sup>C, 1-Wire, SPI and MBus:** This chapter compares the different bus protocols, investigate tradeoffs and look at advantages and disadvantages. This chapter also includes previous power measurement experiments with a couple of the bus protocols. The chapter ends with a suggestion for a suited bus interface for an ultra low power system.

**Chapter 4 - Design Process of an SPI Master:** This chapter will describe the design process of an SPI bus master. The specifications, including port list, architecture, Finite-State Machine (FSM), Addressing and simulation of the design are described. The end of the chapter describes how the SPI master is implemented on a FPGA with the AXI Lite interface and how the registers are set up.

**Chapter 5 - Results for the SPI Experiment:** This chapter contains the results of testing the design with an LM74 sensor and power saving techniques results.

**Chapter 6 - Discussion:** This chapter contains the discussions of the results for the experiment. It also includes what frequencies worked on the design and why. As well as how the power saving techniques affected the design. The chapter ends with the limitations of the findings.

**Chapter 7 - Conclusion:** This chapter is the conclusion of the whole thesis. It summarizes why the SPI is a suitable protocol for an ultra low power device, as well as the design process and testing. This chapter also includes some recommendations for further work/investigation.



# Chapter 2

## Background Theory

This chapter will give a summary on how communication can be done in Integrated Circuits, how this is done with buses, theory behind interconnects, some power saving techniques and information about the different bus protocols I<sup>2</sup>C, 1-Wire, SPI and MBus.

### 2.1 Communication

Communication in Integrated Circuits (IC) are typically designed in serial or parallel, and it can either be done with synchronous or asynchronous transfer.

Parallel communication are used in for example modern computers, and can have eight data wires (+ ground wires/control wires) in parallel between two components. Parallel communication is a method of transferring blocks e.g. one byte of data at the same time. The Figure 2.1 shows an example on how the interface can look like. These connections have a high throughput and are fast. However, they require more lines which can lead to a high energy consumption and increased overall area cost. They are impractical for longer distances and have a higher probability for "*crosstalk*" between the lines. Crosstalk describes the circumstance when a transmitted signal

creates an undesired effect in another circuit or wire. This can corrupt the transmission and the data has to be sent again.

Serial communication can have a single data (+ ground and/or control) wire, where the bits are sent sequentially (one bit at a time). An example of how the interface can look like is seen in Figure 2.2, where 1 byte (= 8 bits) of data is sent from one device to another. This way of communication is more area cost effective and can therefore be a more favourable choice of communication when it comes to IoT devices, because these devices often need to be small-scaled and have few pins between the ICs.

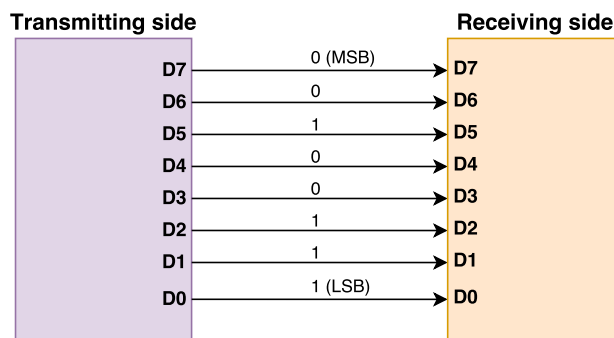


Figure 2.1: Example of Parallel interface

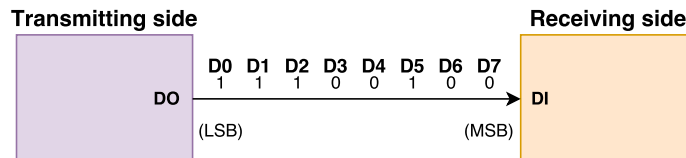


Figure 2.2: Example of Serial interface

Synchronous communication uses a separate clock signal to synchronize transfers between components. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line. Asynchronous communication do not have a separate clock signal line, instead it can use a handshake protocol to synchronize transfer. This often leads to a larger overhead because many of the transmitted bits are used only for control purposes.

## 2.2 Bus Communication

Busess consist of one or several wires where signals are transmitted through for communicating between one or more ICs. The terminology for a bus is defined by [27]:

- **Master** - Initiate and controls data transfer
- **Slave** - Respond to data transfer requests from master
- **Arbiter** - Decides which master grants access to the bus
- **Interface** - Connection between component and bus<sup>1</sup>
- **Bridge** - Connection between buses
- **Decoder** - Decodes the address and selects the correct slave to receive data

The Figure 2.3 shows an example of a bus-based communication architecture with two buses connected via a bridge. For **Bus 1** the Processor is the master of the slave devices Memory 1 and 2. It also contains a Direct Memory Access (DMA) which is often used in systems to relieve the use of the CPU (master) to access the memory. This gives the CPU opportunity to do other tasks or sleep. **Bus 2** has a Digital Signal Processing (DSP) unit which works as the master. **Bus 2** also contains the slave device Memory 3. The components DMA and Memory controller have both master and slave Interface (I/F), which mean they can act as both master and slave. There is also logic components such as decoders, arbiters and bridges in this architecture example. The decoder decodes the destination address of a data transfer initiated by the master. The arbiter determines which master is granted access to the bus. The bridge connects the two buses.

---

<sup>1</sup>buffers, wires, frequency converter etc.

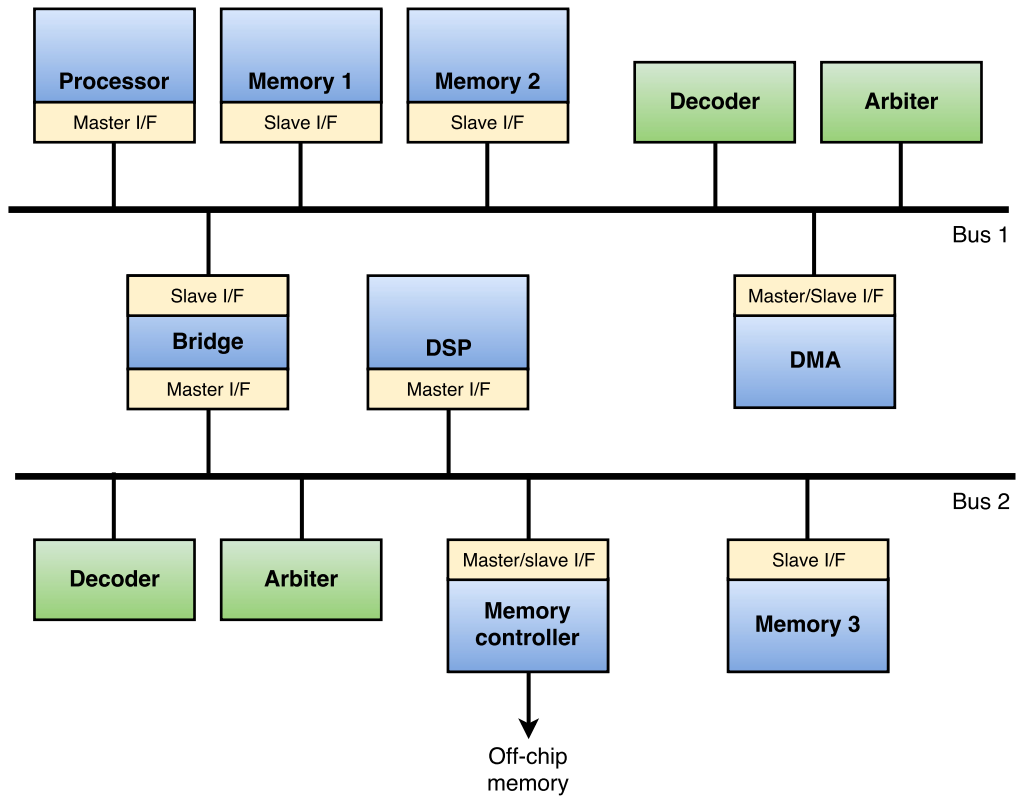


Figure 2.3: Example of bus-based communication architecture. Adapted from source: [27]



Figure 2.4: An example of bidirectional bus lines. Adapted from Source: [27]

The Figure 2.4 shows the three common signal lines for a bidirectional bus. The lines are:

- **Address bus** - Destination address for the transmitted data
- **Data bus** - Information to be transferred
- **Control bus** - Information about the data transfer (e.g. req/ack signals)

The address signals transmits the address for the destination of the data transfer. The data signals transmits the data values to their destination address. The control signals are used to transmit information about the data transfer. This is usually specified by the bus protocol and can be "request", "acknowledge", "data size indicator" and/or "error" signals.

## 2.3 Interconnects

The trend seems to be that the physical dimensions of interconnects are frequently getting reduced and the operating speed increased. The performance of interconnects is progressively affected by their electrical parasitics, i.e. capacitance, inductance and resistance. These parasitics may increase propagation delays for signals traveling in interconnects or in signals distorted by noise. To transmit a signal in an interconnect, the wire capacitances are charged and discharged, which consumes energy. The energy dissipated in the interconnects can grow greatly due to higher frequencies and the increased number of metal layers which seems to be a trend in today's technology. This increased aspect ratio of the wires and metal layers leads to increased wire-to-wire capacitance, compared to wire-to-ground capacitance, which increases the power consumption [29].

In the paper [17] over 50% of the overall dynamic power consumption of a microprocessor was determined to be consumed by interconnect switching. This means that there is a huge potential for reducing power dissipation in interconnects. To reduce the total wire load, buses can be implemented using techniques such as bus splitting [12].

The simplest model of a wire is typically a microstrip. A microstrip can be described simply as a metal strip on top of a ground plane. In Figure 2.5A we see a metal strip on top of a ground plane.

These wires can be modelled through:

- **c** - Capacitance to ground per unit length
- **r** - Resistance per unit length
- **l** - Inductance per unit length

We can simplify the parasitics for very short interconnect lengths, so that the capacitance is sufficient enough as a model. As the length increases the resistance and inductance becomes more important.

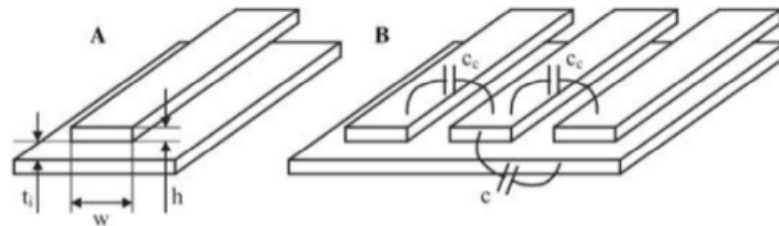


Figure 2.5: Single and multiple microstrip wire. Source: [29]

When wires are surrounded by several other wires, as shown in Figure 2.5B, we have to consider capacitances to neighboring wires as well to the ground plane. They also have a potential problem with crosstalk which can affect power consumption.

However, the power consumption occurs as in logic. The capacitance is charged by the supply voltage and then discharged. Power consumption related to interconnects is given by the Equation 2.1.

$$P_w = 1/2 \alpha f_c C_w \Delta V^2 \quad (2.1)$$

The symbol  $\alpha$  is the signal activity which is the probability that the signal will switch per clock cycle. The  $f_c$  symbol represents the clock frequency,  $C_w$  is the total capacitance and  $\Delta V$  is the signal voltage swing. The signal activity ( $\alpha$ ) in Equation 2.1 makes power predictions difficult when it comes to power optimization, because it depends on data statistics, architectures and what applications run on the system.

Power consumption for a digital Complementary Metal–Oxide–Semiconductor (CMOS) circuit is mainly contributed from three parts:

- Power consumed by leakage current (Leaking diodes and transistors)
- Short circuit current between supply rails during switching
- Charging-discharging capacitors (Dynamic power consumption)

$$\begin{aligned}
 P_{avg} &= P_{switching} + P_{short-circuit} + P_{leakage} \\
 &= \alpha_{0 \rightarrow 1} C_L \cdot V_{dd}^2 \cdot f_{clk} + I_{sc} \cdot V_{dd} + I_{leakage} \cdot V_{dd}
 \end{aligned}
 \tag{2.2}$$

This is represented in Equation 2.2. The first term represent the switching component of power, where  $C_L$  is the load capacitance,  $V_{dd}$  is the supply voltage,  $f_{clk}$  is the clock frequency and  $\alpha_{0 \rightarrow 1}$  is the node transition activity factor (average number of times the node makes a power consuming transition) [7]. The second and third term is not that interesting for the topic in this paper. The second term however is due to the direct-path short circuit current (happens when both NMOS and PMOS transistors are active simultaneously) and last term is leakage current that can arise from substrate injection and subthreshold effects. The power consumption relevant to the charging-discharging current is the dominant part in the total power consumption.

### 2.3.1 Power Savings in Interconnects

There are several methods for reducing the power consumption in interconnects [29]. The most obvious one is to shorten the wire length to reduce the capacitance, resistance and inductance. Another method is to reduce the signal voltage swing  $\Delta V$  via a technique called "*reduced voltage swing*". However, this can lead to "delay penalty". Unless you are making a high performance system this can be accepted. There is also a method to reduce the product of data activity and capacitance by considering the data activity during floorplanning and routing, which optimizes the power consumption based on wire activity/length product.

Another way to reduce the power consumption is to reduce the data activity on buses, which can be done through coding. These methods can be utilized for on-chip and off-chip wires. The Figure 2.6 shows the idea behind the bus coding technique, which is to generate an encoder and a decoder. This can be done by encoding the data signals sent by reducing the switching activity on the bus line, and the decoder will interpret the signals decode it back to the original message.

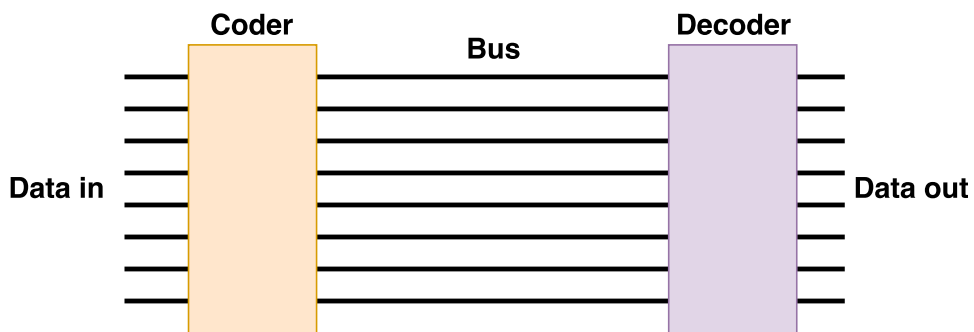


Figure 2.6: Using coding on a bus. Adapted from source: [29]



## 2.4 Coding Techniques

The main reason for coding techniques are:

- Minimizing the power consumption on the bus
- Minimizing the crosstalk delay

Reducing power on bus transaction by lowering transition-switching activity on the bit lines of a bus, will lead to significant reduction in the (dynamic) bus power consumption. There are several coding schemes developed over the years to reduce the logical activity in interconnects. It is also a promising method to avoid problematic switching patterns for long buses. However, this is achieved with additional logic circuitry, which can be problematic for small size systems. One of the most popular techniques to reduce the number of transitions on a bus is the "Bus Invert Coding scheme" [33]. This is a suitable technique for a data bus with uncorrelated data patterns. It is also a good method when you lack prior knowledge of the data statics. For instruction address buses, "inc-xor code" [30], "Gray code" [34], "T0 code" [6] and the "Beach code" [5] have been proposed. In the paper [32] more of these techniques are listed and explored in depth.

## 2.5 Serial communication protocols

Buses have different properties, and one bus interface can be preferred over the other, depending on what environment or requirements you may have. In this section the serial communication protocols I<sup>2</sup>C, 1-Wire, SPI and MBus are listed.

### 2.5.1 I<sup>2</sup>C

Inter-Integrated Circuit (I<sup>2</sup>C) bus requires only two bidirectional open-drain bus lines, a Serial Data Line (SDA) and a Serial Clock Line (SCL) [25]. This protocol is widely used and popular because of its simplicity. It was developed in the early 1980's by Phillips Semiconductors (now NXP Semiconductors) for easy communication between Integrated Circuits (IC) on the same circuit board. The bus is 8-bit oriented, and has bidirectional data transfers up to 100 kbit/s in *Standard-mode*, up to 400 kbit/s in *Fast-mode*, up to 1 Mbit in *Fast-mode plus* or up to 3.4 Mbit/s in *High-speed mode*. The unidirectional data transfers can come up to 5 Mbit/s in *Ultra-Fast mode*. The protocol is a "multi-master", "multi-slave", "single-ended", serial computer bus which includes collision detection and arbitration to prevent data corruption if two or more masters tries to initiate data transfer simultaneously. The interface I<sup>2</sup>C has a 7-bit or a 10-bit address space depending on the devices used. The devices connected to the bus is software addressable with a unique address. The master can operate as master-transmitter or as master-receiver. The Figure 2.7 shows an example of the interface with a microcontroller as master and a Digital to Analog Converter (DAC), Analog to Digital Converter (ADC) and another microcontroller as slaves.

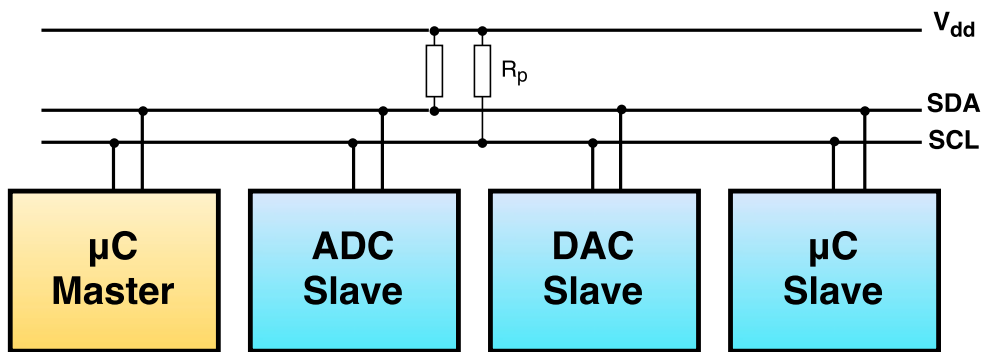


Figure 2.7: A simple I<sup>2</sup>C example. Adapted from source:[38]

### 2.5.2 1-Wire

The 1-Wire bus uses only a single data line and a ground reference for communication [36]. This is accomplished by the bus system using a capacitor to store charge, and power the devices when the data line is active. A network made of 1-Wire devices are called "Microlans" and can be used in different sensor applications. This bus has a rather low-speed data signaling, but then again benefits from the low pin count and longer range than for example I<sup>2</sup>C.

The 1-Wire master can initiate and control the communication with one or more slave devices on the bus. Each slave has a unique 64-bit Identification Number (ID), which is factory programmed. This ID serves as the device address on the bus. A subset of the 64-bit ID, (the 8-bit family code) identifies the device type and functionality. The slave devices usually operates between the voltage range 2.8V(min) to 5.25V(max). The grand part of 1-Wire devices take their energy from the bus (parasitic supply), which means they do not need an extra pin for power supply. In Figure 2.8 we can see an example of a microcontroller as master with multiple temperature sensor as slaves.

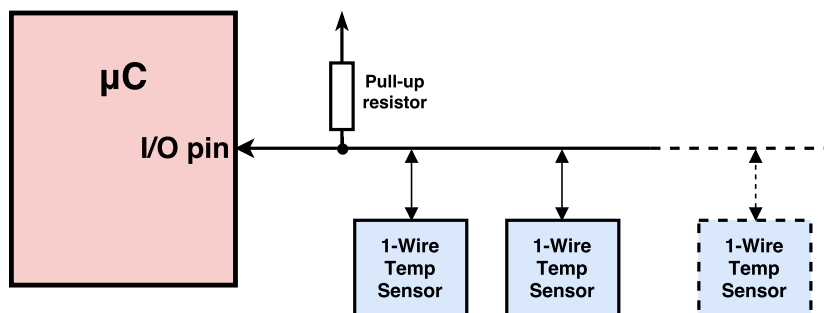


Figure 2.8: A microcontroller with multiple 1-Wire temperature sensors. Adapted from source:[28]

### 2.5.3 SPI

Serial to Peripheral interface (SPI) have a 4-wire synchronous serial interface developed by Motorola in the late 1980s [18]. The bus lines are namely:

- **SCLK** - *Serial Data Clock*:

Outputs the clock from the SPI master to the selected slave on this line to synchronize transfer

- **MOSI** - *Master Output, Slave Input* :

Transmits data out from the SPI master to the selected slave on this line

- **MISO** - *Master Input, Slave Output*:

SPI master receives data from the selected slave on this line.

- **$\overline{CS/SS}$**  - *Chip Select/Slave Select*:

Outputs the slave select signal from the SPI master to a specific slave. The transmission of the data starts with a falling edge of the  $\overline{SS}$  and ends with the rising edge. (The  $\overline{SS}$  line is an input for the slave).

The SPI protocol is not a well defined protocol, and has numerous of versions on the market today. This can be seen as an upside when it comes to the numerous options and adaptation it can have, depending to the users needs. Unlike the protocol I<sup>2</sup>C, SPI does not define which clock frequency to be used.

The pin count increases with the number of slaves because of the "slave select" lines, this can be unfavorable to the area cost if the number of slaves are high. The SPI protocol is primarily used for short distance communication (like I<sup>2</sup>C) and it can communicate in full duplex mode, which means it can send and receive data at the same time. The master selects which slave to transfer to or receive from, by the individual "slave select" lines ( $\overline{SS}$ ). A simple SPI example with one master and several slaves is shown in Figure 2.9.

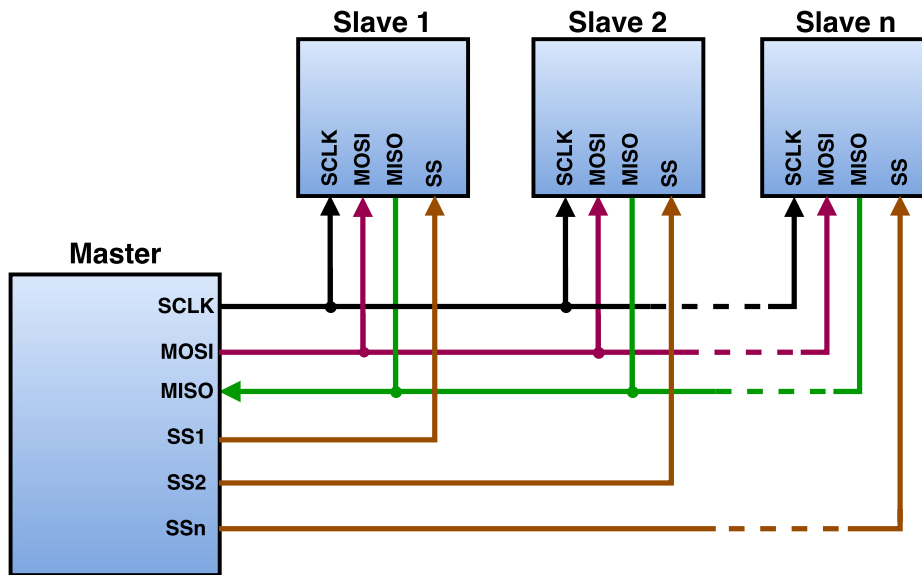


Figure 2.9: A simple SPI example. Adapted from source:[20]

When the SPI master wants to initiate communication, it configures the clock with the frequency supported by the slave device. Then the master selects the slave it wants to write/read from, by switching the  $\overline{SS}$  from logic "1" to "0" for that specific slave line. For example if the master from Figure 2.9 wants to communicate with slave 2, it will switch the "SS2" from logic "1" to "0". The slave for that line will then be listening to the bus and the master is free to transmit data. For each clock cycle a full duplex transmission occurs. This means that data is transmitted and received simultaneously. Slaves devices that are not selected (have  $\overline{SS}$  line high) do not interfere with the SPI bus activities.

Table 2.1: The different SPI modes

Mode	CPOL	CPHA
0	0	0
1	1	0
2	0	1
3	1	1

There are 4 different SPI bus standards modes with regards to the **SCLK** signal. The modes are selected by one of four combinations of serial Clock Phase (**CPHA**) and Clock Polarity (**CPOL**) signals. The different modes and combinations are listed in Table 2.1 and represented in the timing diagram in Figure 2.10 and Figure 2.11. The **CPOL** designates the default value (high or low) of the **SCLK** signal when the bus is in the state "idle". If **CPOL = 0**, the **SCLK** starts out at logic "0" and raises to a "1". If **CPOL = 1**, the default value of **SCLK** is a logic "1" (high) and falls to "0" when it starts transmitting. When **CPHA = 0**, the data on the **MISO** and **MOSI** line is sampled on the first **SCLK** transition (falling or rising edge) seen in Figure 2.10. This means that **CPHA** determines which edge of the **SCLK** the data is sampled. When **CPHA = 0**, the data is interpreted on the first change of the clock seen in Figure 2.10. When **CPHA = 1**, the data is interpreted on the second change of the clock seen in Figure 2.11.

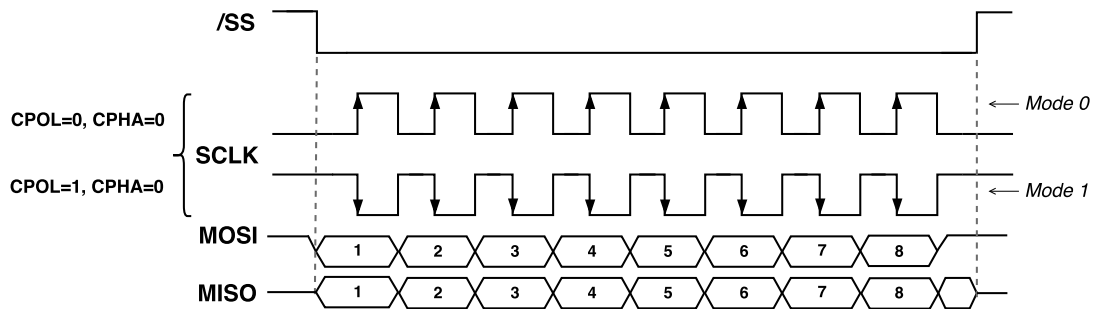


Figure 2.10: Timing diagram of SPI modes 0 and 1

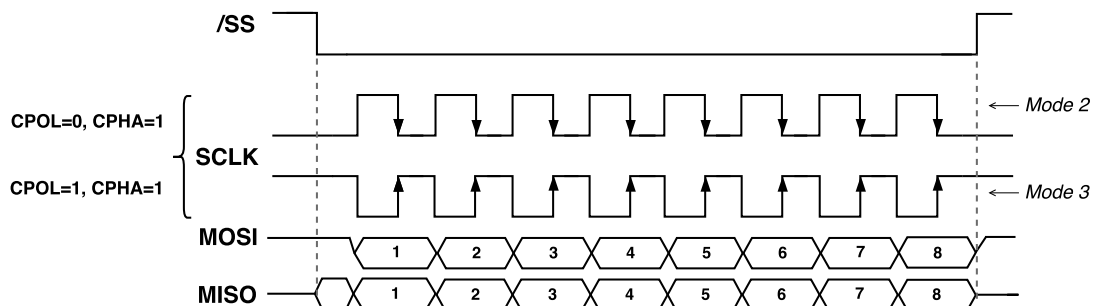


Figure 2.11: Timing diagram of SPI modes 2 and 3

### 2.5.4 MBus

The MBus<sup>2</sup> is described as an ultra-low power interconnect bus which is targeted for microscale systems [26]. It is most known for being a key element for the Michigan Micro Mote platform [37]. Other than that it is not broadly used like the other bus interfaces. MBus is a 4 pin, 22.6pJ/bit/chip (on average) chip-to-chip interconnect.

The MBus system consist of a mediator node and one or more member nodes connected in two "shoot-through" rings, *CLK* and *DATA* which is represented in Figure 2.12. The mediator node generates the MBus clock and resolves arbitration. It can be a stand-alone component or attached to a core device like a microcontroller. MBus delivers a superset of the features from SPI and I<sup>2</sup>C, but with fixed area, fixed pin count and lower power. It also uses a fully synthesizable logic and minimal protocol overhead. The big drawback from this protocol is that it is not commonly used, and there are few products that have this protocol implemented. This causes problems when you for example want a sensor for a project, since it is not yet in commercial "off-the-shelf" products.

---

<sup>2</sup>They have released the MBus specification and a reference Verilog implementation for free at <http://mbus.io/>.

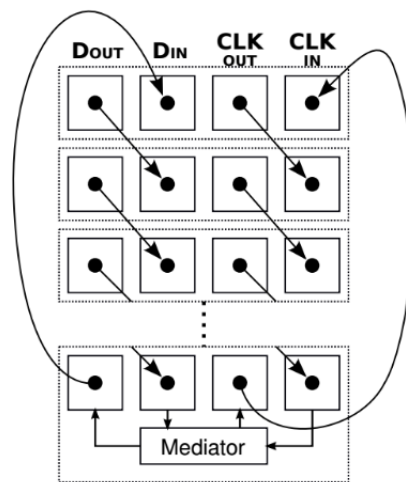


Figure 2.12: MBus physical topology for stepped 3D stacking with wirebonding.  
Source: [26]



# Chapter 3

## Comparing the Bus Protocols I<sup>2</sup>C, 1-Wire, SPI and MBus

In this chapter we look closer at the protocols I<sup>2</sup>C, 1-Wire, SPI and MBus. We will examine the advantages and disadvantages, and determine which protocol is the most suited for an ultra low power interface.

### 3.1 Comparison of bus protocols

We are going to look at the difference between the bus protocols listed in Section 2.5 based on different parameters. In Table 3.1 we can see that SPI can potentially have the highest data rate (70Mbit/s), and the 1-Wire the lowest (16Kbit/s). 1-Wire however has the least number of wires, and SPI can have the most depending on how many slaves are in the system. The SPI bus is the only protocol that can transfer data in full duplex, which means it can send and receive data at the same time. The other bus protocols can only communicate one way at the time. They all share the master-to-slave communication model, and that they can have multiple slaves, but only MBus and I<sup>2</sup>C can have multiple masters. A typical disadvantage for "mas-

ter/slave" architectures is that the communications between slave devices must go through the master. This again doubles the communication costs because the messages is sent twice and there is an extra energy cost of running the central controller. The SPI uses the  $\overline{SS}$  lines to communicate with slaves, while the rest uses an addressing method. Out of all the parameters in Table 3.1 the MBus and SPI are arguably the most favourable for an ultra low power interface.

Table 3.1: Comparison of the serial interfaces. Adapted from source: [26] and [21]

Parameters	SPI	I <sup>2</sup> C	1-Wire	MBus
Number of lines:	3 + 1 per slave	2 with pull-ups	1 with pull-ups	4
Data transfer:	full duplex	half duplex	half duplex	half duplex
Communication model:	master-to-slave	master-to-slave	master-to-slave	master-to-slave
Number of master devices:	single	multiple	single	multiple
Number of slave devices:	multiple	multiple <sup>a</sup>	multiple	multiple
Maximum data rate, Mbit/s:	70 <sup>b</sup>	3.4 <sup>c</sup>	16Kb/s	10 <sup>d</sup>
<i>Interface features:</i>				
Device identification mechanism:	slave select	address	address	address
<b>Critical</b>				
I/O pads (n nodes)	3 + n	2/4 <sup>e</sup>	-	4
Standby power:	Low	Low	Low	Low
Active power:	Low	High	Low	Low
Synthesizable:	Yes	Yes	Yes	Yes
Globally unique addresses:	-	128	2 <sup>48</sup>	2 <sup>24</sup>
Multi-master (interrupt):	No	Yes	-	Yes
<b>Desirable</b>				
Broadcast messages:	Option	No	Yes	Yes
Data independent:	Yes	Yes	-	Yes
Bits overhead (n byte message):	2	10 + n	-	19 or 43
Hardware acknowledgement:	No	Yes	-	Yes
Power aware:	No	No	No	Yes

The SPI bus do not suffer the same power challenges faced by "open-collectors" such as I<sup>2</sup>C and 1-Wire. "Open-collectors" usually suffer from high power dissipation. The SPI interface also has the advantage with little to no "protocol overhead". A potentially disadvantage is that the SPI requires a "slave-select" ( $\overline{SS}$ ) line for every slave

<sup>a</sup>119 when using 7-bit addresses or 1024 when using 10-bit addresses

<sup>b</sup>According to [13]

<sup>c</sup>according to [25]

<sup>d</sup>According to [31]

<sup>e</sup>When wirebonding, a shared bus requires two pads/chip

device, which means it can be challenging when we think of small and more I/O constrained devices. For modular systems with a variable and unknown number of components until design time, it can be difficult to choose the right amount of "slave select" lines. This can lead to choosing too many lines which can violate the area constraints, or insufficient lines which will disrupt the modularity.

## 3.2 Previous Power Measurement Experiments

There has been done experiments comparing different bus protocols by power consumption. First of is the article [21]. It involves investigating and comparing three digital serial interfaces (UART, I<sup>2</sup>C and SPI) by their energy efficiency under a given "test bed" build around PICKit 3 development boards [19]. The experiments was executed using one hardware platform PIC18F45K20 microcontroller. The Table 3.2 shows the results of the experiment. It is not representative for energy consumption on a bus for SoC, but it shows the difference which can be propagated for a SoC. The Table 3.2 show the power consumption for data transmissions over the interface implemented in software vs. the same interface implementation using the inbuilt microcontroller hardware module. According to the Table 3.2 the SPI interface is seen to be the lowest of the three interfaces, from the point of power consumption. The conclusion for this paper was that the SPI has the lowest power consumption both with inactive interface and during data exchange when all the interfaces were using the same data rate. This applied for both implementation in hardware and in software.

The second research paper done on the same topic is [23]. They compared the two digital serial interfaces SPI and I<sup>2</sup>C on a PIC16F877 microcontroller. They constructed the serial communication between two microcontrollers where one was master and

Table 3.2: Energy consumption for evaluated serial interfaces [21]

Parameters	UART		SPI				I <sup>2</sup> C			
	1 byte	9 bytes	RX selected		RX unselected		RX selected		RX unselected	
			1 byte	9 bytes	1 byte	9 bytes	1 byte	9 bytes	1 byte	9 bytes
<i>Interface implementation in hardware</i>										
Power consumption with inactive interface, mW	6.06	6.06	<b>4.66</b>	<b>4.66</b>	<b>4.66</b>	<b>4.66</b>	6.13	6.13	6.13	6.13
<i>Communication:</i>										
Required time, ms	0.68	5.78	<b>0.52</b>	<b>4.6</b>	<b>0.52</b>	<b>4.6</b>	1.26	5.91	1.26	5.91
Required energy, $\mu$ J	7.31	53.31	<b>2.5</b>	<b>23.53</b>	<b>2.42</b>	<b>22.52</b>	8.11	31.71	8.01	30.23
<i>Interface implementation in software</i>										
Power consumption with inactive interface, mW	11.04	11.04	<b>11.17</b>	<b>11.17</b>	<b>11.17</b>	<b>11.17</b>	11.23	11.23	11.23	11.23
<i>Communication:</i>										
Required time, ms	0.64	5.72	<b>0.51</b>	<b>4.62</b>	<b>0.51</b>	<b>4.61</b>	1.23	5.85	1.23	5.86
Required energy, $\mu$ J	9.00	80.06	<b>7.26</b>	<b>65.25</b>	<b>7.26</b>	<b>65.25</b>	17.45	82.84	17.44	83.6

the other one was a slave. They then transmitted one-byte data from master to slave and displayed it in two 7-segment LEDs and observed. From their experiment with transmitting one byte of data (0x30 in Hex) from master to slave with 5V supplied, the energy consumed was:

- **SPI** - 9.325nJ
- **I<sup>2</sup>C** - 126.68mJ

The results from their experiment shows that the SPI interface consumes considerably less energy than the I<sup>2</sup>C interface for sending one byte of data. The SPI interface consumes approximately  $1.36 \cdot 10^7$  less energy than I<sup>2</sup>C according to these results. This is mainly due to the extra pull-up resistors the I<sup>2</sup>C interface needs, which are power hungry. By looking at these two articles the SPI shows best promise out of the known interfaces, regarding power/energy consumption.

### 3.3 Suggestion for solution

The I<sup>2</sup>C bus is a widely used communication protocol mainly because it only requires two wires, independent of the number of connected devices. The big drawback from both I<sup>2</sup>C and 1-Wire bus is that each wire requires an open-collector circuit which needs pull-up resistors and they are not considered energy efficient.

The SPI bus do not suffer the same power challenges faced by open-collectors (like I<sup>2</sup>C and 1-Wire) and have little to no protocol overhead. The articles mentioned in the previous Section 3.2 comes to the conclusion that SPI uses the least amount of energy (compared to I<sup>2</sup>C). However as mentioned, the SPI requires a "slave-select" ( $\overline{SS}$ ) line for every slave device (represented in Figure 2.9), so this can be challenging when we think of small and more I/O constrained devices.

In the paper [26] they "taped out" a chip that implemented the MBus into a temperature sensor system where the power traces required 22.6pJ/bit/chip, which where two orders of magnitude better than the standard I<sup>2</sup>C. The limitations with MBus is that it does not guarantee fairness (nor does I<sup>2</sup>C). The intention behind the development of MBus was to be used for ultra low power devices and micro scale systems. As mentioned, it is not widely used like the other protocols, which causes problems to get modules with MBus interface.

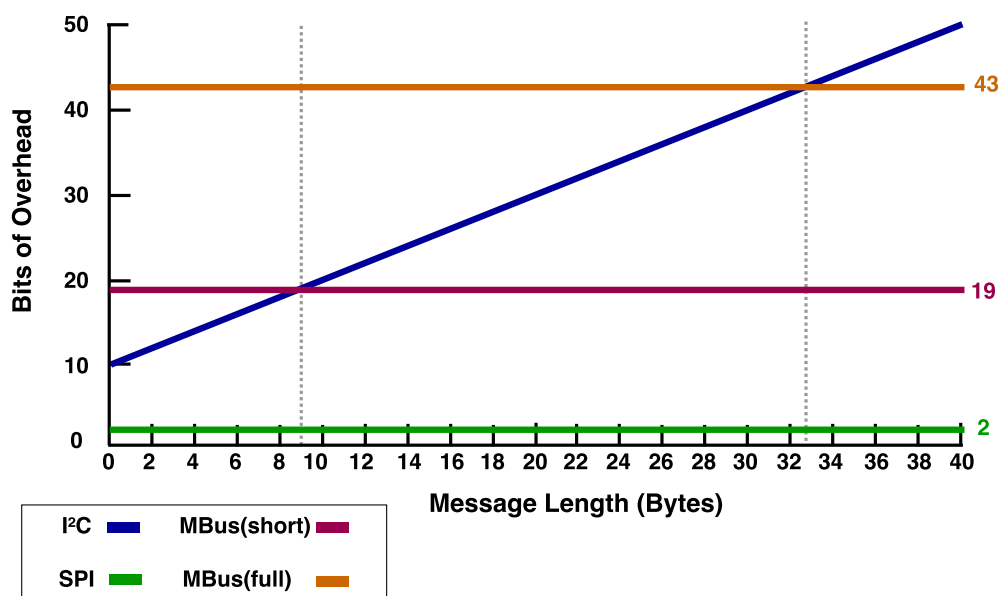


Figure 3.1: Bus overhead. Adapted from source: [26]

In Figure 3.1 we can see the comparison of bits of overhead and message length in bytes as they increase. We see that message overhead is independent for SPI and MBus. For short addressed MBus messages it is 19 bits of overhead and for full addressed messages it is constant 43 bits of overhead. SPI holds the lowest constant of overhead at 2 bits. This makes these buses scale efficiently to messages, while I<sup>2</sup>C steadily increases the bits of overhead when the message length increases. For short-addressed MBus messages we see that they become more efficient than I<sup>2</sup>C after 9 bytes.

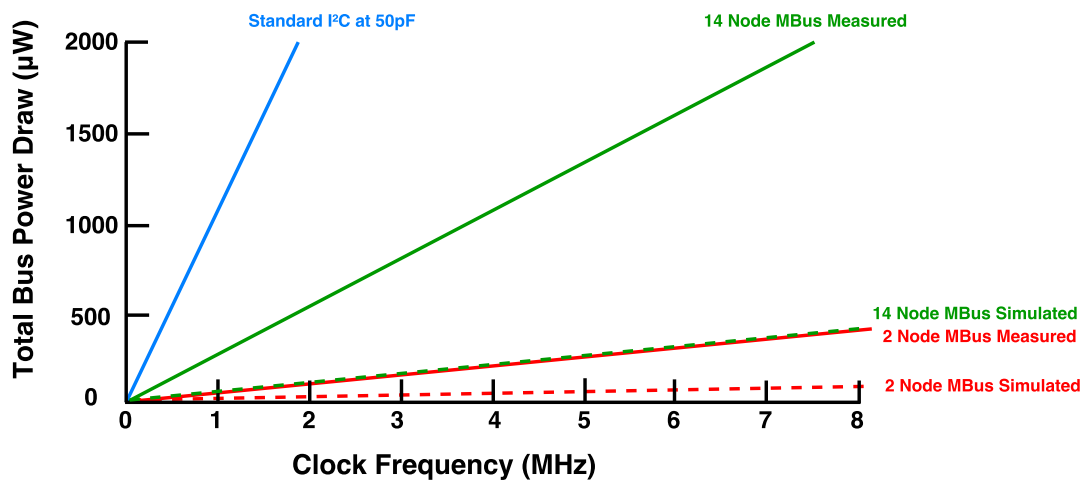


Figure 3.2: Total Power Draw. Adapted from source: [26]

In Figure 3.2 we can see the different power draws when clock frequency and node population increases for the I<sup>2</sup>C and MBus configurations [26]. We can see that for both simulated and measured nodes, MBus outperforms the I<sup>2</sup>C.

The best approach for a communication solution for an ultra low power device would either be the SPI or MBus protocol. They both consumes considerably less power than the I<sup>2</sup>C based on the results seen in Section 3.2 and from the paper [26]. The SPI is a widely used interface compared to MBus. There are many modules such as sensors with implemented SPI interfaces, but minimal amount of modules with the

Mbus interface. This means it would be much cheaper and easier to built a system with the SPI interface. Therefore it will be more beneficial for someone to design an ultra low power system with the SPI interface. This is why I choose to design the SPI master in the next chapter.





# Chapter 4

## Design Process of an SPI Master

This chapter contains the design process of an SPI master. The design process I am following for my design is seen in Figure 4.1. The first section is to provide the specification for the design, then write the Register Transfer Level (RTL) code for the master design, parallel with the "test bench" in "Xilinx Vivado Design Suite"<sup>a</sup>. We will further run a simulation based on the design file and the "test bench". After that we do the synthesis for the design and then the implementation. When this is done, we program and debug the code and export it to "Xilinx Software Development Kit" (SDK) to write an application software

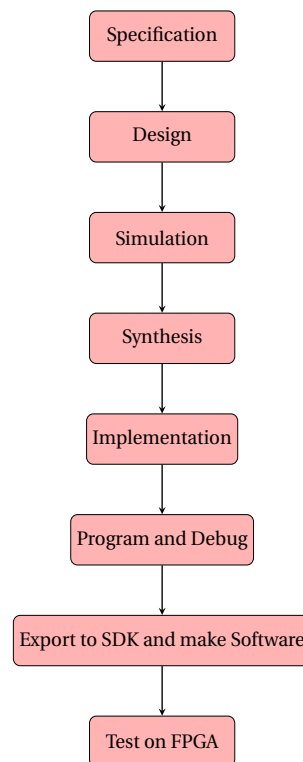


Figure 4.1: Flowchart of the design process

<sup>a</sup><https://www.xilinx.com/products/design-tools/vivado.html>

program in "C". This is going to run on the design which will be implemented on the FPGA on the "Zedboard development kit"<sup>b</sup>. The Zedboard uses the "Xilinx Zynq®-7000 All Programmable System on Chip (SoC)". The last step is to test the design with an external temperature sensor. The Appendix B shows how to connect the sensor "LM74" (Appendix A) to the Zedboard.

## 4.1 SPI Specification

For any design it is important to have a detailed and complete specification, so that it works as a foundation for the whole design.

### 4.1.1 Port List

The port list made for this SPI master design can be seen in the Table 4.1. This port list is custom made by me for this design, but sections of the port list is typical for every SPI master design [22]. This includes the external signals **o\_SPI\_sclk**, **o\_SPI\_ss\_n**, **o\_SPI\_mosi** and **i\_SPI\_miso** which is necessary for an SPI master to be valid [22]. The rest of the signals are for the "user logic" and can be different from one SPI master to another.

The signal **g\_divide\_value** is based on the Equation 4.1, and determines the frequency of the SCLK signal sent to the slave, seen in the Figure 4.2. In Equation 4.1 the  $f_{sclk}$  represents the **o\_SPI\_sclk** signal and the  $f_{clock}$  represents the **i\_clock** signal. An example on how to use the Equation 4.1 could be to set the **g\_divide\_value** = 1, and the clock frequency (**i\_clock**) to 50MHz. This would equal the SCLK frequency to 25MHz.

$$f_{sclk} = \frac{f_{clock}}{2 \times g\_divider\_value} \quad (4.1)$$

---

<sup>b</sup><http://zedboard.org/product/zedboard>

Table 4.1: SPI signals in the design

Term used	Definition	InputOutput
g_data_bus_width	Data bus Width	Generic
g_num_of_slaves	Number of slaves	Generic
g_divide_value	Divider value for sclk	Generic
g_CPOL	Clock Polarity	Generic
g_CHPA	Clock Phase	Generic
i_clock	System Clock	Input
i_reset_n	Reset (Active Low)	Input
i_enable	Initiates transfer	Input
i_write_data	Data to Transmit	Input
o_done	Signal the end of transfer	Output
o_read_data	Received data from Slave	Output
i_addr	Address for the Slave	Input
o_SPI_sclk	Serial Clock Line	Output
o_SPI_ss_n	Slave select (Active Low)	Output
o_SPI_mosi	Master out Slave In	Output
i_SPI_miso	Master In Slave Out	Input

The SPI master can have several slaves for this design. This would also imply that the number of wires would increase with the increased number of slaves. This can be negative for a small scaled system.

#### 4.1.2 SPI Architecture

The proposed architecture based on this SPI master module can be seen in the Figure 4.2. The inputs on the left side of the block is signals from the user logic and the signals on the right side is for communication with the slave devices. In our example we use only one slave, a Temperature sensor (Appendix A). This architecture is for my SPI master design and is not a standard for other SPI master designs. Another SPI Master design can be seen in [1] and has other signals and architecture than my design.

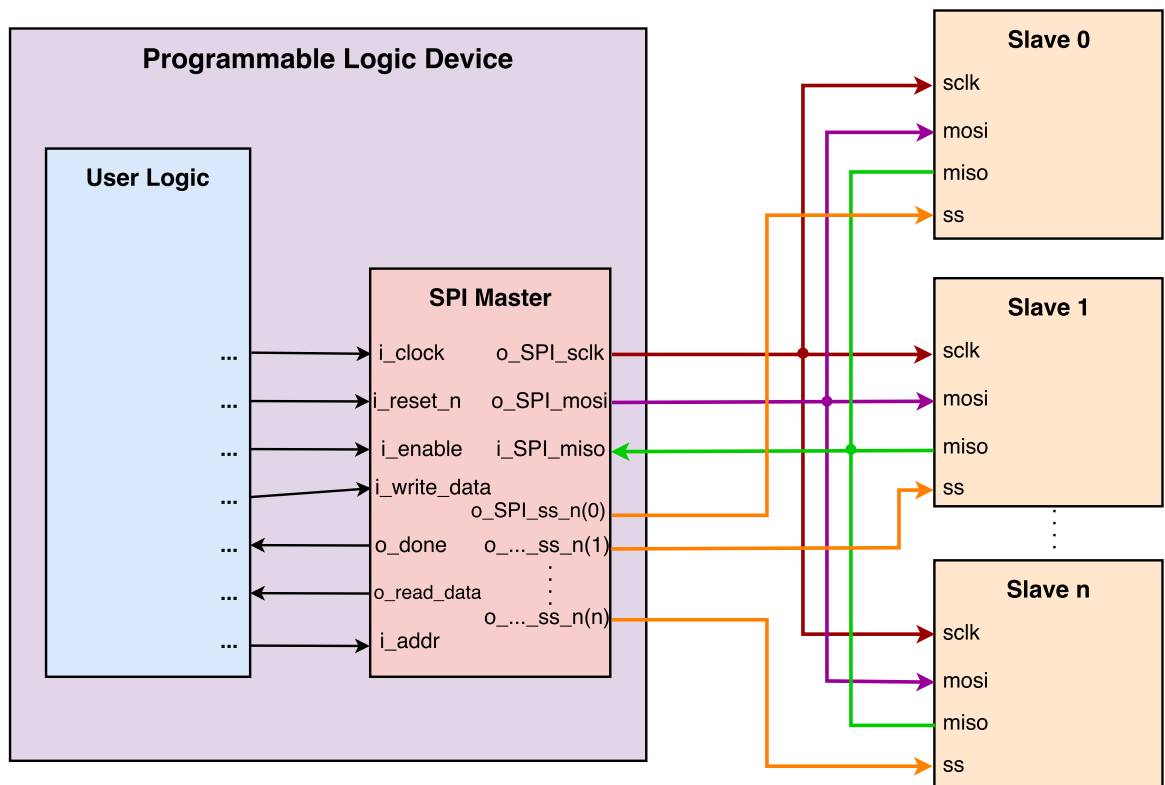


Figure 4.2: This is the architecture of the design

### 4.1.3 SPI FSM

The Figure 4.3 shows how the Finite State Machine (FSM) of the design is constructed. It starts of in the state **IDLE** and jumps to the next state when **r\_transmit\_start** signal goes high. In this state (**s\_transmission**) the transmission happen (sending and receiving data). When the signal **w\_counter\_data** goes high and either **r\_rise\_sclk** or **r\_fall\_sclk** signal goes high (depending which mode you use) it jumps to the next state. The next state is **s\_transmission\_end** and in this state the transmission stops and goes to the next state when **r\_rise\_sclk** or **r\_fall\_sclk** signal goes high (depending which mode you use).

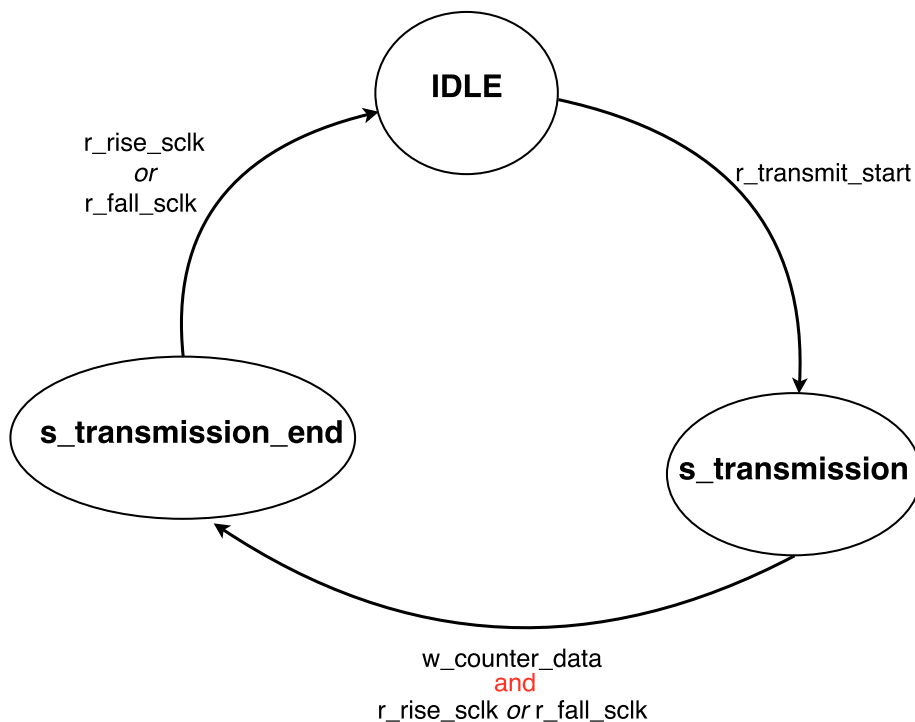


Figure 4.3: This is the FSM of the SPI Master

In the SPI Master design we have three states for executing the transfer of data, seen in Figure 4.3. The standard state machine encoding in Vivado is binary encoding. For this design we use Gray encoding which is known for using lesser power. We should

also note that State machine encoding in Gray eliminates any glitches/hazards on combinational equations that depend on the state [2].

Table 4.2: Binary vs. Gray Encoding

Binary code		Gray code	
Sequence	No. toggles	Sequence	No. toggles
00	1	00	2
01	1	01	1
10	2	11	1

The Table 4.2 represents Binary and Gray encoding for three states. We can see the number of toggles per transition between the states in combination with the sequences of the bits.

```

type state is (s_idle, s_transmission, s_transmission_end);
  signal present_state, next_state : state;
  attribute enum_encoding: string;
  -- Optional attribute IMPORTANT FOR OPTIMAL USAGE
  attribute enum_encoding of state: type is "00 01 11";
  -- Describes which encoding style used

```

Figure 4.4: FSM VHDL code

In Figure 4.4 we see how this is implemented in VHDL. We can see the three states **s\_idle**, **s\_transmission** and **s\_transmission\_end** is declared in the first line. The last line determines the encoding style, which in our case is Gray. This is retrieved from Appendix D.1.

#### 4.1.4 SPI Addressing

The Table 4.3 show how the addressing works with 4 slaves for this design. If you choose to address slave 3 you send 0x3 to the **i\_addr** signal and the MSB of **o\_SPI\_ss\_n** goes low for slave 3. You can see this simulated in the Figure 4.5. If you want to communicate with slave 1 you can send 0x1 to **i\_addr** and the **o\_SPI\_ss\_n** goes low for the LSB seen in Table 4.3. This can be seen simulated in the Figure 4.6.

Table 4.3: SPI Master addressing

Name	i_addr		o_SPI_ss_n
	Hexadecimal	Binary	
slave 0	0	0000	1110
slave 1	1	0001	1101
slave 2	2	0010	1011
slave 3	3	0011	0111

## 4.2 SPI Simulation

The RTL code for the SPI master logic can be found in Appendix D.1 and the code for the "test bench" can be found in Appendix D.2. The behaviour simulation for this design can be seen in Figure 4.5 and Figure 4.6. The Figure 4.5 represents the simulation for the SPI in mode 2 (CPOL = 0 and CPHA =1) and Figure 4.6 represents the simulation of the SPI in mode 3 (CPOL = 1 and CPHA = 1). The modes are explained in Section 2.5.3, and the timing diagram for these modes are defined in Figure 2.11. The purpose for the "test bench" is to test the behavior of the SPI master, and verify that it functions correctly. We use the "test bench" to evaluate all the "user logic" signals and the standard SPI signals.

Note that the "test bench" code in Appendix D.2 have some parameters that may vary from the simulations. The **i\_clock** signal for these simulations runs at 50MHz which

means the **c\_CLOCK\_PERIOD** from the "test bench" is set to 20ns. This will assign the **o\_SPI\_sclk** signal to run at a frequency of 25MHz. The **g\_num\_of\_slaves** are set to "4" for these simulations. Also the line that addresses the slaves in Appendix D.2 (page 99) is set to slave 3 (0x03) in Figure 4.5 and to slave 1 (0x01) in Figure 4.6.

The difference between these two simulations are the SPI modes and the slave addressing. We can see the difference in the **o\_SPI\_sclk** signal behaving as low (logic "0") when not used (IDLE state) in mode 2 (Figure 4.5), and high (logic "1") when not used (IDLE state) for mode 3 (Figure 4.6). From Figure 4.5 the SPI master addresses "slave 3" and in the Figure 4.6 the SPI master addresses "slave 1". This can be distinguished by the signal **i\_addr** = 3 and the **o\_SPI\_ss\_n** signal set to logic "0" for slave 3 (0111) seen in Figure 4.5.

From the Figure 4.5 we see the **i\_clock** signal running at 50MHz and the **i\_reset\_n** signal high (not enabled). The **i\_enable** signal asserts at 164,2ns, which assert the **r\_transmit\_start** register signal in the SPI master design. From Figure 4.3 we can see that this register is responsible for the state jumping from "IDLE" to "s\_transmission". Now the transition can begin. From Figure 4.5 we see the **o\_SPI\_ss\_n** signal go low for the slave addressed as "3" seen by the **i\_addr** signal being set to "3". This initiates the communication between the master and the selected slave. The data sent from the slave to master can be seen by that the **i\_SPI\_miso** signal is "0xC9" in Hex ("11001001" in binary). The first bit in **i\_SPI\_miso** is sampled on the second falling edge of the **o\_SPI\_sclk** signal (because CPOL=1). The data from **i\_SPI\_miso** is sent to the "user logic" signal **o\_read\_data** seen by value "4980" in Hex, (0100 1001 1000 000 in binary). This is "4980" in Hex because it reads **i\_SPI\_miso** from all the sixteen cycles.



After eight clock cycles of the **o\_SPI\_sclk** signal, the first bit of the data in **o\_SPI\_mosi** is sampled. The data that is sent from the master to the slave is seen by the "user logic" signal **i\_write\_data** equal to "0x93" in Hex ("10010011" in binary). After the full sixteen clock cycles the signal **o\_done** is asserted, and the **o\_SPI\_ss\_n** goes high for the selected slave. This also includes the **w\_counter\_data** counter signal in the SPI master seen in Figure 4.3, which means we jump to state "s\_transmission\_end". This concludes that the transmission is finished and we end up in state "IDLE". The signals that are not mentioned from the figures are for debug purposes and can be overlooked.

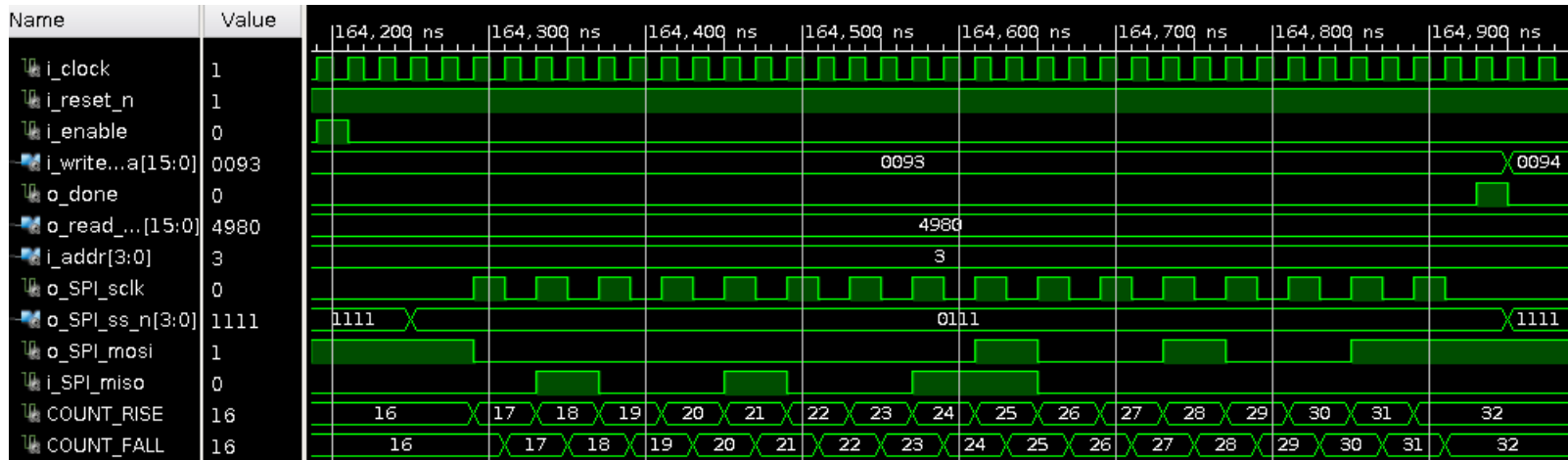


Figure 4.5: Simulation of the SPI Master (Mode 2)

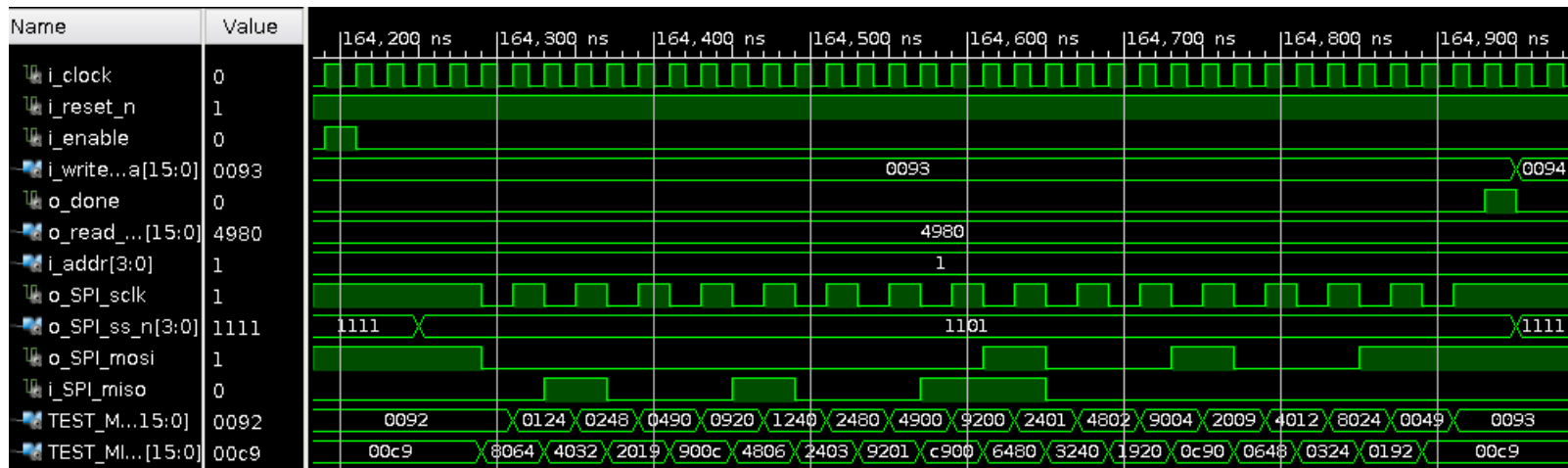


Figure 4.6: Simulation of the SPI Master (Mode3)

## 4.3 SPI Registers

There are two registers used for this design. Both have a width of 32 bits and one is primarily for writing (input for the SPI master) and one is for reading (output from the SPI master). In Figure 4.7 we can see how this is implemented in VHDL. The two registers are named **slv\_reg0** and **spi\_to\_ps**.

Table 4.4: slv\_reg0, D31 to D16

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
i_reset_n	i_enable	i_write_data													

Table 4.5: slv\_reg0, D15 to D0

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
i_write_data	X	i_addr	X	X	X	X	X	X	X	X	X	X	X	X	X

The Table 4.4 shows what data is stored in the range bit 31 to bit 16 in the **slv\_reg0**. The Table 4.5 shows the data stored in register **slv\_reg0** from bit 15 to bit 0. The first bit is **i\_reset\_n** which can be set to "0" to reset the SPI master. The second bit place (D30) is reserved for the **i\_enable** signal which is set to high to enable the transmission. The next places are for the **i\_write\_data**, which is the size of 16, which means that it takes up 16 bits. This is customized and can also be selected to be the size of 8, by changing the **g\_data\_bus\_width** value to 8 instead of 16. This will affect the register size automatically to only contain 8 bits of **i\_write\_data**. The **i\_addr** in Table 4.5 only takes up 1 bit because in our design we use one slave. If we want to use more than one slave, we can increase the **g\_num\_of\_slaves** value, and this will in turn increase the space taken in the register **slv\_reg0**.

Table 4.6: spi\_to\_ps, D31 to D16

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
o_done	o_read_data														

Table 4.7: spi\_to\_ps, D15 to D0

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
o_read_data	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

The Table 4.6 and Table 4.7 shows where the output signals takes place in the **spi\_to\_ps** register. Table 4.6 shows what data is stored in the range bit 31 to bit 16, and Table 4.7 shows what data is stored in the range bit 15 to bit 0. In Table 4.6 we see that first place is reserved to the **o\_done** signal, which goes high when the transmission in the SPI master design is finished. The next places are for **o\_read\_data**, which takes up 16 bits because of the **g\_data\_bus\_width** = 16 in our design. If we change this down to 8 this will also automatically affect the **spi\_to\_ps** register so **o\_read\_data** only takes up 8 bits.

### 4.3.1 Implementing the Registers

To communicate with the design through the AXI interface we need to read and write from registers. This is made custom for this design in the "port map" section in the "my\_spi\_v1\_0\_S00\_AXI.vhd" file. The whole code is listed in Appendix E.

```

port map(

  -- USER LOGIC -----
  i_clock  => S_AXI_ACLK,
  i_reset_n => slv_reg0(31),

  -----
  i_enable   => slv_reg0(30),
  i_write_data => slv_reg0(29 downto (30 - g_data_bus_width)),
  o_done     => spi_to_ps(31),
  o_read_data => spi_to_ps(30 downto (31 - g_data_bus_width)),
  i_addr     => slv_reg0(12 downto (13 - g_num_of_slaves)),

  -- SPI MASTER INTERFACE -----
  o_SPI_sclk => o_SPI_sclk,
  o_SPI_ss_n => o_SPI_ss_n,
  o_SPI_mosi => o_SPI_mosi,
  i_SPI_miso => i_SPI_miso

);

```

Figure 4.7: VHDL code of the registers

In Figure 4.7 you can see the VHDL code for the "port map" where the registers are appointed to the different signals. This is taken from the Appendix E.2. We use only two of the four registers in the AXI lite interface for this design, **slv\_reg0** and **spi\_to\_ps**. The **slv\_reg0** contains the inputs **i\_reset\_n** (at bit 31), **i\_enable** (at bit 30), **i\_write\_data** (at bit 29 down to a set size) and **i\_addr** (at bit 12 down to a set size). The size of the **i\_write\_data** depends on the size of the **g\_data\_bus\_width** chosen for our design. This is set at size 16 for our experiment, which means it takes the place bit 30 down to 15 in **slv\_reg0**. The same goes for **i\_addr**, it depends on **g\_num\_of\_slaves** which is set to value "1" for this experiment. This means our design can have a different number of slaves and data bus size depending on what we need it for.

The second register **spi\_to\_ps** is the outputs from our design. We have the signal

**o\_done** and **o\_read\_data**. The signal **o\_done** goes high when the transaction is complete and we can read this from the register and read out the data stored in the **spi\_to\_ps** register bit 30 down to bit 14. This makes the design easier to program an application for. The **o\_read\_data** also depends on the size of the data bus width.

### 4.3.2 AXI lite Connection

To implement the SPI master to the FPGA there are some additions we need work out. In Appendix C there is a tutorial on how this can be executed. We have to implement the AXI interface to be able to communicate between the Processor System (PS) and the Programmable Logic (PL). We are using AXI lite for our implementation because it is a "light-weight", single transaction memory mapped interface [9]. It includes the necessary functionality to communicate between the PS and the SPI master.

The SPI master will be in the PL part, but we need the PS to operate it. In Figure 4.8 you can see the block design of my custom SPI master Intellectual Property (IP) block with the Zynq processing IP block together with the AXI interconnects and the external input/outputs.

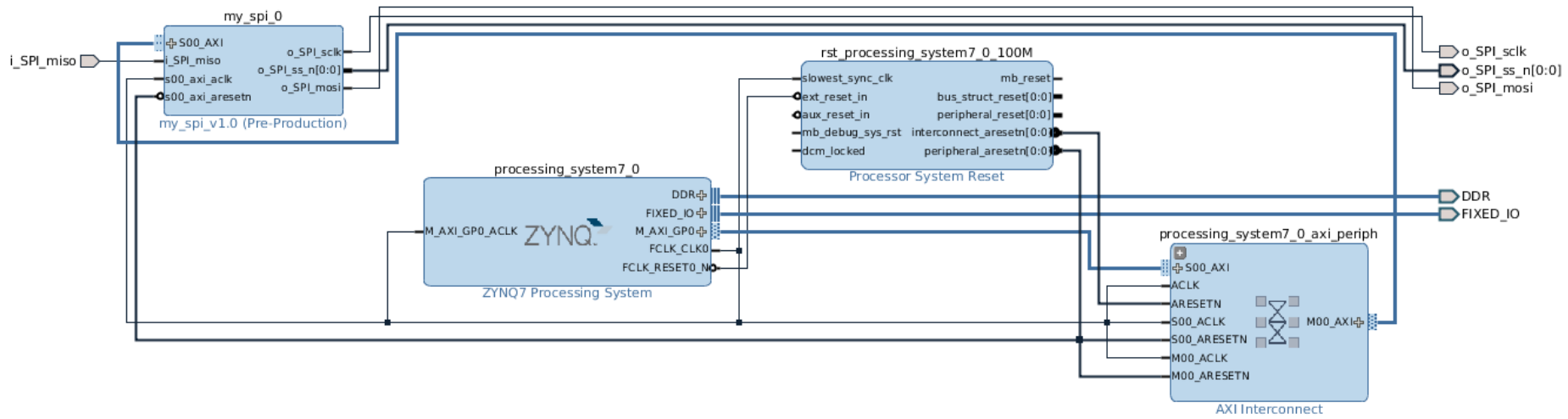


Figure 4.8: Block diagram of the AXI interconnects with the SPI Master and PS

The Figure 4.9 shows a close up of the SPI Master IP block also seen in the top left of the Figure 4.8. This IP block is called "my\_spi\_0" and contains the code from Appendix E. We use the **i\_SPI\_miso** as an external input which is addressed to the Pmod "JA1" described in Appendix B. The same goes for the signals **o\_SPI\_sclk**, **o\_SPI\_ss\_n**, **o\_SPI\_miso** which is external outputs also connected to the Pmod "JA1". The input and output "user logic" is sent through the **S00\_AXI** input/output for the IP block. The clock signal for the **i\_clock** inside the SPI IP block is sent through the **s00\_axi\_aclk** with the frequency 100MHz. For the full description of the addressing see Appendix B.

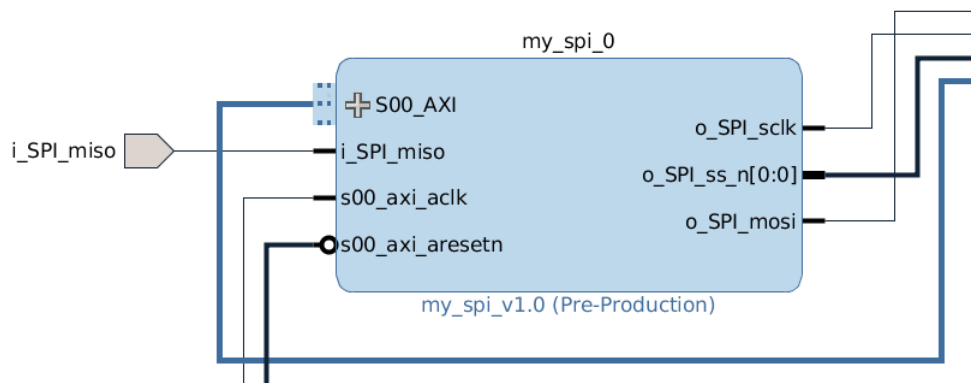


Figure 4.9: Block diagram of the SPI master



# Chapter 5

## Results for the SPI Experiment

In this section we test the SPI master module on the Zedboard with the LM74 Temperature sensor listed in Appendix A. A test program written in C (seen in Appendix E) on Vivado's SDK software runs a test application which sets different registers and outputs to read data from the LM74 sensor to a serial terminal. The serial terminal used for this experiment is called "GtkTerm". A short tutorial on how to use the serial terminal can be found at [10]. The next section contains testing different encoding techniques for the FSM on the SPI master and what this does for the utilization report gathered from Vivado.

### 5.1 SPI with LM74

In Figure 5.1 you can see the setup for the testing environment using a standard household temperature sensor to validate the temperature. For this experiment the input clock for the system (*i\_clock*) is set to 100MHz and the divider value (*g\_divide\_value*) is set to 4. From the Equation 4.1 the serial clock (*o\_SPI\_sclk*) for the SPI master runs at frequency 12.5MHz.

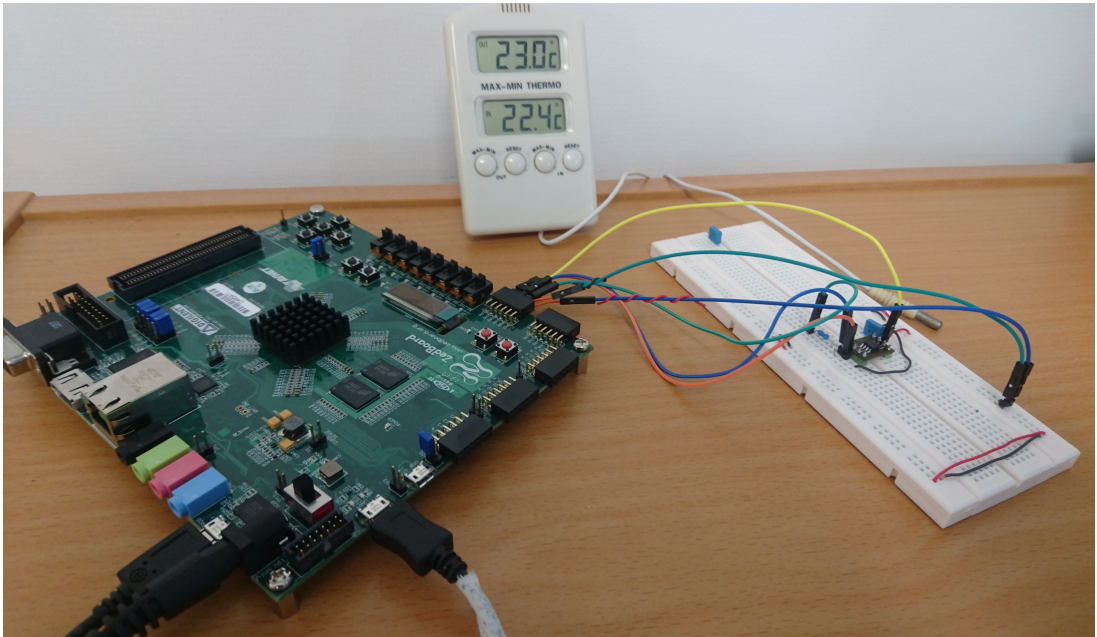


Figure 5.1: Testing environment

In Figure 5.2 we can see the output from the serial terminal "GtkTerm" read from the sensor when running the test application software. The first three lines in the Figure 5.2 is just for debug purposes to validate the correct value of the registers. The fourth line shows that the temperature sensor outputs 22.93°C.

```
GtkTerm - /dev/ttyACM0 115200-8-N-1 - + x
File Edit Log Configuration Control signals View Help
-----Test Project -----
---FØR WHILE: slv_reg1 = 0x5BF8000
---ETTER WHILE slv_reg1 = 0x5BF8000
---ETTER WHILE slv_reg1 (18) = 0x16F
TEMPERATUR MÅLER = 22.937500 °C
-----Test Project END-----
/dev/ttyACM0 115200-8-N-1          DTR RTS CTS CD DSR RI
```

Figure 5.2: Read from terminal

The temperature read from the sensor through the design is validated by a household

thermometer seen in Figure 5.3. In the Figure we can see that the thermometer shows two different values; 23.1°C from its outside sensor and 22.8°C for its internal sensor.



Figure 5.3: Validating temperature

```
GtkTerm - /dev/ttyACM0 115200-8-N-1 - + x
File Edit Log Configuration Control signals View Help
-----Test Project -----
---FØR WHILE: slv_reg1 = 0x66B8000
---ETTER WHILE slv_reg1 = 0x66B8000
---ETTER WHILE slv_reg1 (18) = 0x19A
TEMPERATUR MÅLER = 25.625000 °C
-----Test Project END-----
/dev/ttyACM0 115200-8-N-1 DTR RTS CTS CD DSR RI
```

Figure 5.4: Temperature increasing

In Figure 5.4 we can see the output temperature after running the application software again with me touching my finger on the LM74 sensor. The temperature outputs 25.62°C in the serial terminal.

## 5.2 Power Saving Techniques

In Section 4.1.3 the different encoding techniques "Gray code" and "Binary code" is mentioned. The VHDL code implementation of this technique is also showed in Figure 4.4. When implementing the design in "Vivado" the software presents the utilization report of the design. In Figure 5.5 we see the utilization report for using "Binary encoding" exclusively for the SPI master design seen in Appendix D.1 (without AXI lite and the PS). The Figure 5.6 shows the utilization report for the SPI master design when using "Gray encoding".

Resource	Estimation	Available	Utilization %
LUT	34	53200	0.06
FF	65	106400	0.06
IO	40	200	20.00
BUFG	1	32	3.13

Figure 5.5: Utilization Report of Binary Encoding

Resource	Estimation	Available	Utilization %
LUT	26	53200	0.05
FF	65	106400	0.06
IO	40	200	20.00
BUFG	1	32	3.13

Figure 5.6: Utilization Report of Gray Encoding

The effect of switching from "Binary -" to "Gray encoding" in the FSM is marked with the red square in Figure 5.5 and Figure 5.6. The number of Look Up Tables (LUT) goes from 34 to 26, by switching to "Gray encoding". This is an improvement of 23,53% in minimizing the LUTs.

# Chapter 6

## Discussion

### 6.1 Power Consumption

From Section 3.2 we found that the I<sup>2</sup>C interface used  $1.36 \cdot 10^7$  times more power than SPI to transfer a byte of data. In the paper [26] they "taped out" a chip that implemented the MBus into a temperature sensor system, where the power traces required 22.6pJ/bit/chip, which were two orders of magnitude better than the standard I<sup>2</sup>C. The MBus is not a widely used interface, which is why the SPI master was chosen to be designed instead of a MBus master.

### 6.2 SPI with LM74

The results seen in Figure 5.4 where I touched my finger on the "LM74" sensor, the temperature increase was not as expected. The average temperature on hands lies around 32°C [4], but in my experiment the temperature showed 25.62°C for my finger. This may be caused by several factors. My hand might have been cold at the moment of testing, or perhaps my finger was not long enough on the sensor. Another reason might be that my finger did not cover the whole surface of the "LM74".

Regarding the different frequencies we can have for the **i\_clock** and **o\_SPI\_sclk**, the highest frequency the SPI master could run without any faults was to run the **i\_clock** with 100MHz (input from the PS) and the **o\_SPI\_sclk** with 12.5MHz (**g\_divide\_value** = 4). When running design with higher frequencies (by reducing the **g\_divide\_value**), the serial terminal could not output any sensible data. I tried with 25MHz and 16.6MHz for the **o\_SPI\_sclk**, but that caused incoherent data output on the serial terminal. However, reducing the frequency worked without any difficulty. For example the **o\_SPI\_sclk** could operate at 10MHz and function correctly. When I reduced the system clock (**i\_clock**) from 100MHz down to 50MHz, the design could not function correct with the **o\_SPI\_sclk** frequency over 6.25MHz. (**i\_clock** = 50MHz and **g\_divide\_value** = 4). This could be because of delay in the relatively long wires from the "Zedboard" to the external sensor.

From the Equation 2.1 in Section 2.3 we see that the frequency has an effect on the power consumption in interconnects. Usually we want a low frequency while still managing to handle tasks. This lowers the dynamic power, but keeping the frequency high might however be a tradeoff when it comes to the time of the transition of data. If we can send the data fast and the system goes back to a "power saving mode" (IDLE) this would be preferable to lingering in the "power hungry" transmission state longer than it have to.

### 6.3 Power Saving Techniques

By switching from "Binary-" to "Gray encoding" the results show that it lowers the LUT count which means it takes up less area. This again could lead to lower power dissipation. It is an easy change and can have big improvement of the design, so it's always important to try different encoding techniques to see what improvements it

might lead to. In this design we only have three states, but if we had more states other encoding techniques could improve the utilization even more.

## 6.4 Limitations of the Findings

The limitations of this design is that it's directed for a FPGA, and may therefore be have a little different than if implemented on an ASIC. However, this could be very expensive to test this. It is also shown from the paper [15] that FPGAs are estimated to be 3-4 times slower, 5-35 times larger and 7-14 times less energy efficient than ASICs. This is dependent on the flexibility of the FPGA and the application on it. The SPI design could therefore be assumed to have a better outcome if implemented on an ASIC.

The SPI master design includes two modes, mode 2 and 3 seen in Figure 2.11, but lacks mode 0 and 1 from Figure 2.10. This is a limitation of the SPI design because it makes it less usable if a given slave requires one of the modes that this SPI master lacks.

The software application (Appendix E.3) used to set and read registers is a simple test program. This could be improved by being more structured, but it is sufficient enough when testing the SPI master with an external sensor.





# Chapter 7

## Conclusion

In this thesis we have looked at different types of communication methods and what can be done to decrease the power usage of such methods. We found out that SPI is arguably the most suitable bus protocol for an ultra low power device. MBus is a good competitor by being an ultra low power interface, but has yet to expand in the commercial market, which makes it impractical to use. We have designed an SPI master from the ground-up with some power saving techniques to make it more optimal for ultra low power devices.

The SPI master got implemented and tested on a Zedboard with a temperature sensor (LM74) as a slave. A test application program was made to read and write from the registers and output it on a serial terminal on the computer. The experiment showed that the SPI master design functioned in communicating with the external sensor. It was fast, responsive and outputted the correct data validated by a household thermometer. It managed this with a max frequency of 12.5MHz for the *SCLK*. We also got to see the effect by switching the "Binary" standard encoding on the FSM to "Gray encoding". This minimized the LUTs used on the FPGA by 23.53% seen in the utilization report.

## 7.1 Recommendations for Further Work

A short term goal could be to get the other two SPI modes (mode 0 and mode 1) to work with the design. No code is perfect, so it could also be beneficial for a more experienced digital designer to look at the code and maybe make some improvements. The exact power consumption usage of the SPI master is also very hard to measure when implemented on a FPGA. It would be interesting to research and compare different power saving techniques used in the VHDL code. It could also be interesting to implement this SPI master on an ASIC and see how that would affect the overall performance and efficiency.

# References

- [1] Adelkrim Kamel Oudjida et al. (2009). "SPI-Master-Transceiver Specifications". [https://www.researchgate.net/publication/309514060\\_SPI-Master\\_Specification\\_Document](https://www.researchgate.net/publication/309514060_SPI-Master_Specification_Document). [Online; accessed 28-June-2017].
- [2] Arora, M. (2011). *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*. SpringerLink : Bücher. Springer New York.
- [3] Avnet (2014). "Zedboard Hardware User's Guide". [http://zedboard.org/sites/default/files/documentations/ZedBoard\\_HW\\_UG\\_v2\\_2.pdf](http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf). [Online; accessed 30-May-2017].
- [4] Benedict, F. G., Miles, W. R., and Johnson, A. (1919). The temperature of the human skin. *Proceedings of the National Academy of Sciences of the United States of America*, 5(6):218–222.
- [5] Benini, L., Micheli, G. D., Macii, E., Poncino, M., and Quez, S. (1997a). "system-level power optimization of special purpose applications: the beach solution". In *Proceedings of 1997 International Symposium on Low Power Electronics and Design*, pages 24–29.
- [6] Benini, L., Micheli, G. D., Macii, E., Sciuto, D., and Silvano, C. (1997b). "asymptotic zero-transition activity encoding for address busses in low-power

- microprocessor-based systems". In *Proceedings Great Lakes Symposium on VLSI*, pages 77–82.
- [7] Chandrakasan, A. P. and Brodersen, R. W. (1995). Minimizing power consumption in digital cmos circuits. *Proceedings of the IEEE*, 83(4):498–523.
- [8] Digilent (2013). "Zedboard Schematics Rev. D.2". [http://zedboard.org/sites/default/files/documentations/ZedBoard\\_RevD.2\\_Schematic\\_130516.pdf](http://zedboard.org/sites/default/files/documentations/ZedBoard_RevD.2_Schematic_130516.pdf). [Online; accessed 30-May-2017].
- [9] Digilent (2015). "UG1037 AXI Reference Guide". [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_ref\\_guide/latest/ug1037-vivado-axi-reference-guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf). [Online; accessed 12-June-2017].
- [10] Elinux (2013). "Communicate with hardware using USB cable for Ubuntu". [http://elinux.org/Communicate\\_with\\_hardware\\_using\\_USB\\_cable\\_for\\_Ubuntu](http://elinux.org/Communicate_with_hardware_using_USB_cable_for_Ubuntu). [Online; accessed 15-June-2017].
- [11] Evans, D. (2011). "the internet of things: How the next evolution of the internet is changing everything". *Cisco Internet Business Solutions Group*.
- [12] Hsieh, C.-T. and Pedram, M. (2002). "architectural energy optimization by bus splitting". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(4):408–414.
- [13] Johnson, D. (2010). "implementing serial bus interfaces using general purpose digital instrumentation". *IEEE Instrumentation Measurement Magazine*, 13(4):8–13.
- [14] Konstantin Mikhaylov, J. T. and Fadeev, D. (2012). "development of energy efficiency aware applications using commercial low power embedded systems". In *Embedded Systems - Theory and Design Methodology*, pages 407–430.

- [15] Kuon, I. and Rose, J. (2007). Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215.
- [16] Liu, D. and Svensson, C. (1994). Power consumption estimation in cmos vlsi chips. *IEEE Journal of Solid-State Circuits*, 29(6):663–670.
- [17] Magen, N., Kolodny, A., Weiser, U., and Shamir, N. (2004). "interconnect-power dissipation in a microprocessor". In *Proceedings of the 2004 International Workshop on System Level Interconnect Prediction, SLIP '04*, pages 7–13, New York, NY, USA. ACM.
- [18] Microchip (2011). "Section 23. Serial Peripheral Interface (SPI)". Microchip.
- [19] Microchip Technology (2010). PICKit 3 Programmer/Debugger Users Guide. [ww1.microchip.com/downloads/en/DeviceDoc/51795B.pdf](http://ww1.microchip.com/downloads/en/DeviceDoc/51795B.pdf). [Online; accessed 19-January-2017].
- [20] Mike Grusin (2017). Serial Peripheral Interface. <https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi>. [Online; accessed 21-January-2017].
- [21] Mikhaylov, K. and Tervonen, J. (2012). "evaluation of power efficiency for digital serial interfaces of microcontrollers". In *5th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5.
- [22] Motorola (2003). "SPI Block Guide V03.06". [https://engineering.purdue.edu/ece362/Refs/9S12C\\_Refs/S12SPIV3.pdf](https://engineering.purdue.edu/ece362/Refs/9S12C_Refs/S12SPIV3.pdf). [Online; accessed 28-June-2017].
- [23] Nithiyananthan, K. and Nithya, S. (2015). "enhanced energy consumption

- model for digital serial interfaces in embedded systems". *International Journal of Electronics and Electrical Engineering*, 3(1):1–6.
- [24] Nordrum, A. (2016). "Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated".  
<http://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated>  
[Online; accessed 19-January-2017].
- [25] NXP Semiconductors (2014). "*I<sup>2</sup>C-bus specification and user manual*", volume 6. NXP Semiconductor.
- [26] Pannuto, P., Lee, Y., Kuo, Y. S., Foo, Z., Kempke, B., Kim, G., Dreslinski, R. G., Blaauw, D., and Dutta, P. (2015). "mbus: An ultra-low power interconnect bus for next generation nanopower systems". In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 629–641.
- [27] Pasricha, S. and Dutt, N. (2008). "*Chapter 2 Basic Concepts of Bus-Based Communication Architectures*", pages 17–41. Morgan Kaufmann.
- [28] picprojects.net (2017). DS1820 1-Wire Temperature Sensor. <http://www.picprojects.net/ds1820/>. [Online; accessed 21-January-2017].
- [29] Piguet, C. (2005). "*Low-Power CMOS Circuits: Technology, Logic Design and CAD Tools*". CRC Press.
- [30] Ramprasad, S., Shanbhag, N. R., and Hajj, I. N. (1999). "a coding framework for low-power address and data busses". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(2):212–221.
- [31] s. Kuo, Y., Pannuto, P., Kim, G., Foo, Z., Lee, I., Kempke, B., Dutta, P., Blaauw, D., and Lee, Y. (2014). Mbus: A 17.5 pj/bit/chip portable interconnect bus for

- millimeter-scale sensor systems with 8 nW standby power. In *Proceedings of the IEEE 2014 Custom Integrated Circuits Conference*, pages 1–4.
- [32] Saini, S. (2015). *Bus Coding Techniques*, pages 115–152. Springer New York, New York, NY.
- [33] Stan, M. R. and Burleson, W. P. (1995). "bus-invert coding for low-power i/o". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(1):49–58.
- [34] Su, C. L., Tsui, C. Y., and Despain, A. M. (1994). "saving power in the control path of embedded processors". *IEEE Design Test of Computers*, 11(4):24–31.
- [35] Texas Instruments (2013). LM74 Temperature sensor. [www.ti.com/lit/ds/symlink/lm74.pdf](http://www.ti.com/lit/ds/symlink/lm74.pdf). [Online; accessed 19-May-2017].
- [36] Texas Instruments (2016). "1-Wire Enumeration". [www.ti.com/lit/an/spma057b/spma057b.pdf](http://www.ti.com/lit/an/spma057b/spma057b.pdf). [Online; accessed 18-January-2017].
- [37] University of Michigan (2015). "Michigan Micro Mote (M3) Makes History". <https://www.eecs.umich.edu/eecs/about/articles/2015/Worlds-Smallest-Computer-Michigan-Micro-Mote.html>. [Online; accessed 18-January-2017].
- [38] wikipedia (2017). I2C. <https://en.wikipedia.org/wiki/I%C2%B2C>. [Online; accessed 21-January-2017].
- [39] Žilvinas Nakutis (2013). "Embedded Systems Power Consumption Measurement Methods Overview". *MATAVIMAI*, 44(2):29–35.





# Appendix A

## LM74 Temperature Sensor

This appendix contains information about the Temperature sensor used with the SPI Master design. Most of this information is sourced from the LM74 Specification datasheet [35].

### A.1 LM74

The LM74 is a temperature sensor (Figure A.1) by Texas Instruments, Delta-Sigma analog-to-digital converter with an SPI and MICROWIRE compatible interface. The LM74 provides resolution of up to 0.0625 degree Celsius and operates between -55 degree to -150 degree Celsius. It works as a slave in a system.

For my project I am going to use the SPI interface. The type used in this project is a "Small Outline Integrated Circuit" (SOIC), which is a surface-mounted integrated circuit package. This causes problems with easy connecting and testing of the sensor. I got help at the electronic lab at NTNU to make a custom Printed Circuit Board (PCB) that functions as a breakout board for the SOIC pins seen in Figure A.2. This makes it compatible to mounting to a breadboard and simplifies the connection process to the Zedboard.

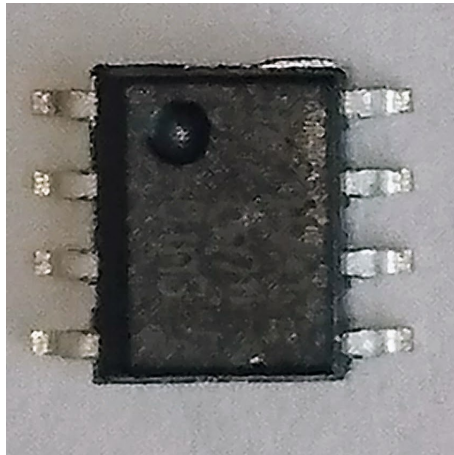


Figure A.1: The LM74 chip

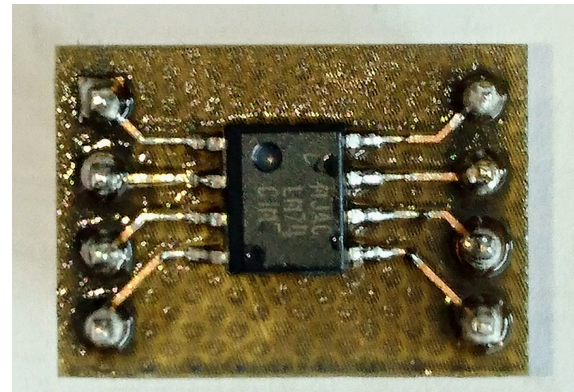


Figure A.2: The LM74 on a breakout board

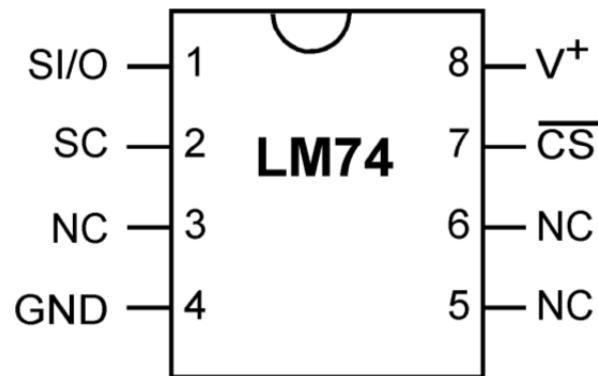


Figure A.3: SOIC - Top View. Source [35]

## A.2 Serial Bus Interface

The LM74 operates as a slave and is compatible with SPI and MICROWIRE bus specification. Data is clocked out on the falling edge of the serial clock (SCLK), while data is clocked in on the rising edge of SCLK. The first 16 clocks comprise the transmit phase of communication, while the second 16 clocks are the receive phase. This means that a complete transmit/receive communication will consist of 32 serial clocks.

### A.3 Interfacing with 3-wire SPI

The SPI master module made in this project is a normal four signal, SCLK,  $\overline{\text{CS/SS}}$ , MISO and MOSI. The LM74 temperature sensor slave uses a 3-wire SPI interface, so we have to connect the sensor in a different way.

Figure A.4 shows how the LM74 temperature sensor is connected to a microcontroller's general purpose I/O. This is also how it is connected into the Zedboards Pmod which is described in Appendix B. The Figure A.5 shows how the LM74 sensor is connected to a basic breadboard with the all the signal wires.

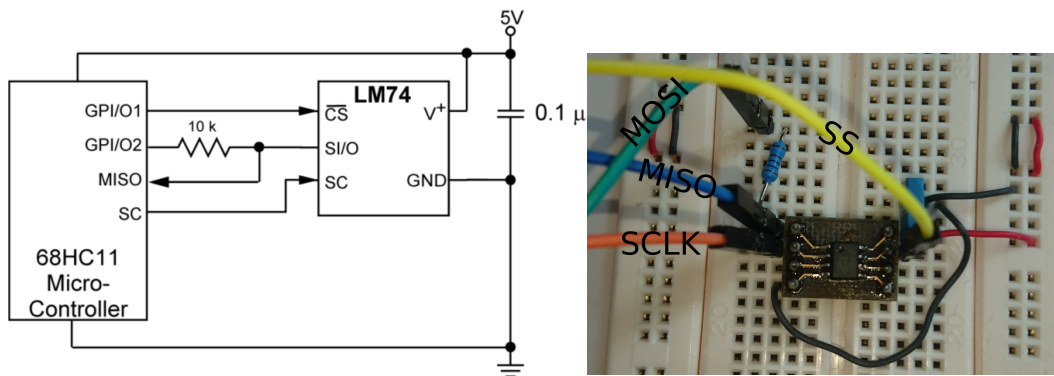


Figure A.4: LM74 digital input control using  $\mu\text{C}$ 's GPIO. Source [35]



# Appendix B

## Connecting to the Zedboard

The Appendix B contains the details used to connect the temperature sensor LM74 to a Pmod on the Zedboard. The Figure B.1 shows which Pmod (JA1) we are using, and the connections of UART USB and JTAG USB. UART is used for transferring the data from the Zedboard to a serial terminal on the computer. The JTAG is used to program the Zedboard.

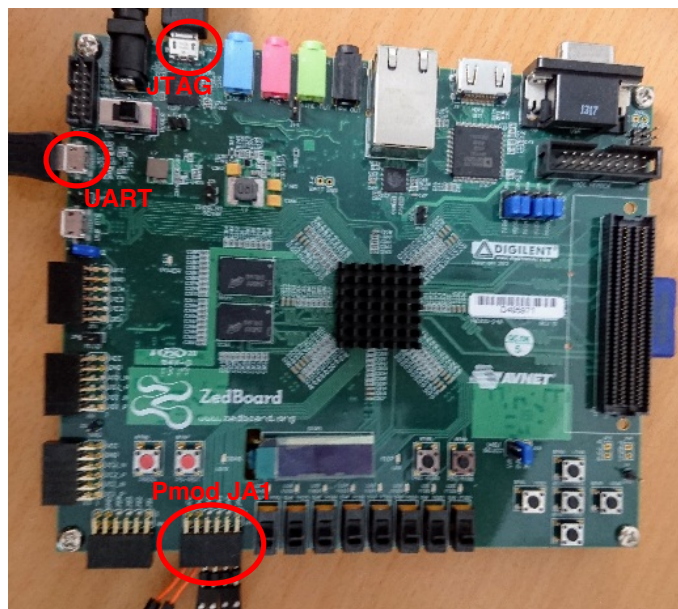


Figure B.1: The Zedboard Connections

The Figure B.2 shows the physical female connection pins of the Pmod. We only need to use the second row for our connections (marked with red). We connect the wires following the Figure B.2 and the Table B.1 accordingly.

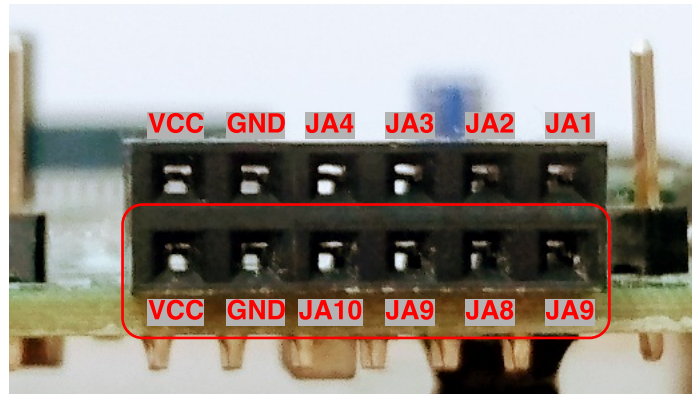


Figure B.2: The Zedboard Connections

The Table B.1 shows the signal name, what the Zynq pin is called and what we connect from the LM74 sensor seen in Appendix A. The Zynq Pin names are used when we write the I/O addresses for the external pins in Vivado. (Seen in Appendix C in Figure C.25).

Table B.1: Pmod Connection. Adapted from source [3]

Pmod	Signal Name	Zynq pin	from LM74
JA1	JA1	Y11	
	JA2	AA11	
	JA3	Y10	
	JA4	AA9	
	JA7	AB11	sclk
	JA8	AB10	ss
	JA9	AB9	Miso
	JA10	AA8	Mosi

The Figure B.3 is the schematic of the Pmod JA1.

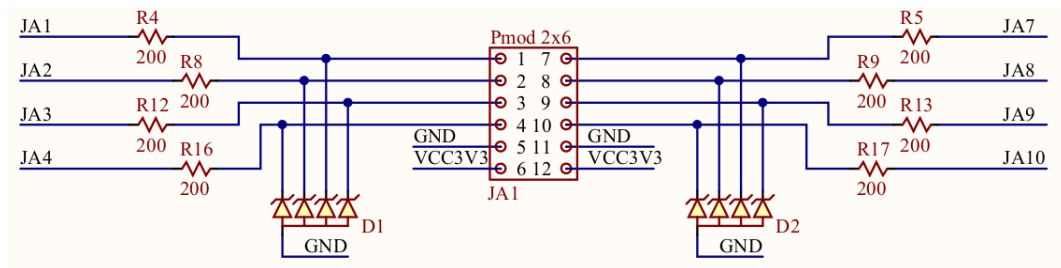


Figure B.3: Pmod schematic. Source [8]





# Appendix C

## Axi Lite core

In this appendix we will see the walkthrough of how to implement our design to the ZEDBOARD (ZYNQ-7000). We are going to create an embedded system design using Vivado and SDK, the objectives for this appendix is listed below.

- Configure the Processing System (PS)
- Add a custom IP in the Programmable Logic (PL) section
- Use SDK to build a software application and verify the hardware functionality

The AXI-Lite interface is a lightweight and low bandwidth version of the AXI-Full protocol. It is suited for simple control logic between software and hardware, it allows for communication between the PS and PL. For this tutorial VHDL is used and where not specified, assume that you should click "finish" or "next" where applicable.

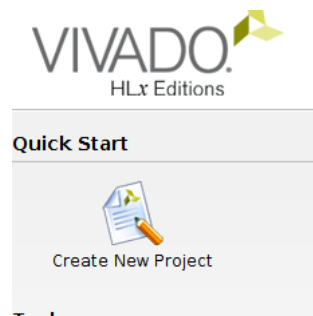


Figure C.1: Create a new project

First of launch Vivado (Vivado 2016.3 WebPACK Edition for this example). Click **Create a New Project** to make the new project. (Figure C.1)

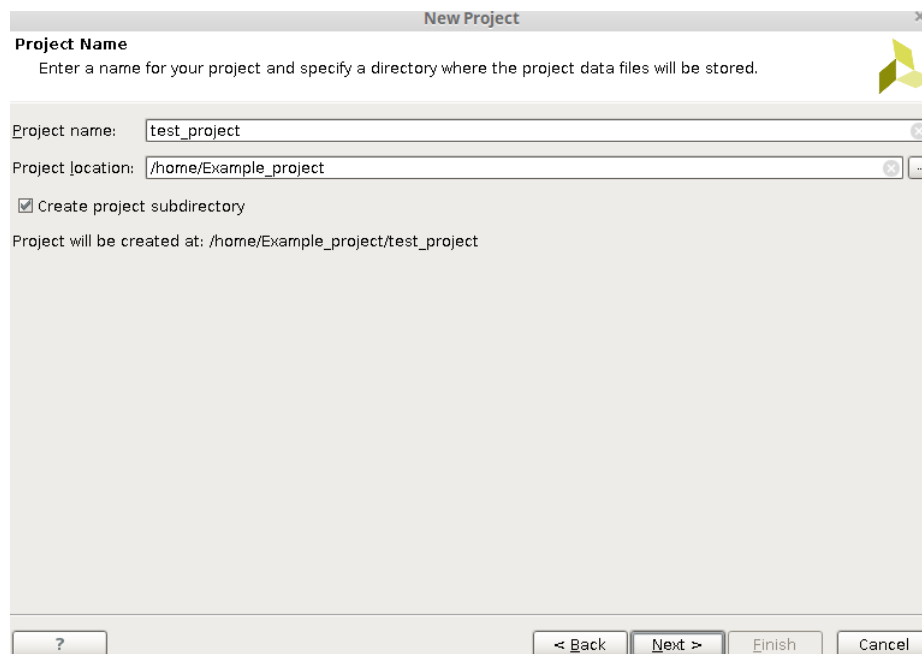


Figure C.2: Pick a suitable project name

Give the project a appropriate name and chose a suitable location for saving the design. Click "next".(Figure C.2)

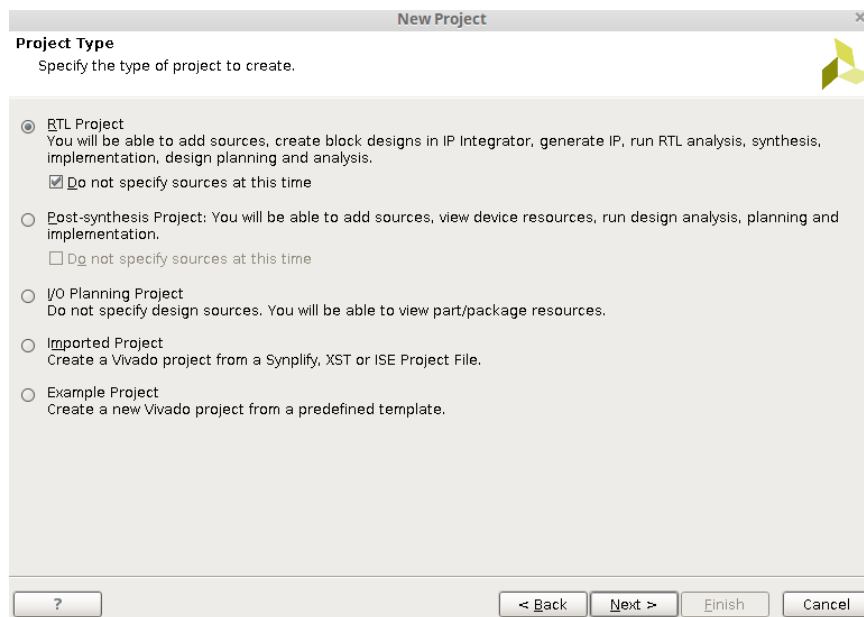


Figure C.3: Choose RTL Project

Choose "RTL Project" and check the "Do not specify sources at this time". Click "next". (Figure C.3)

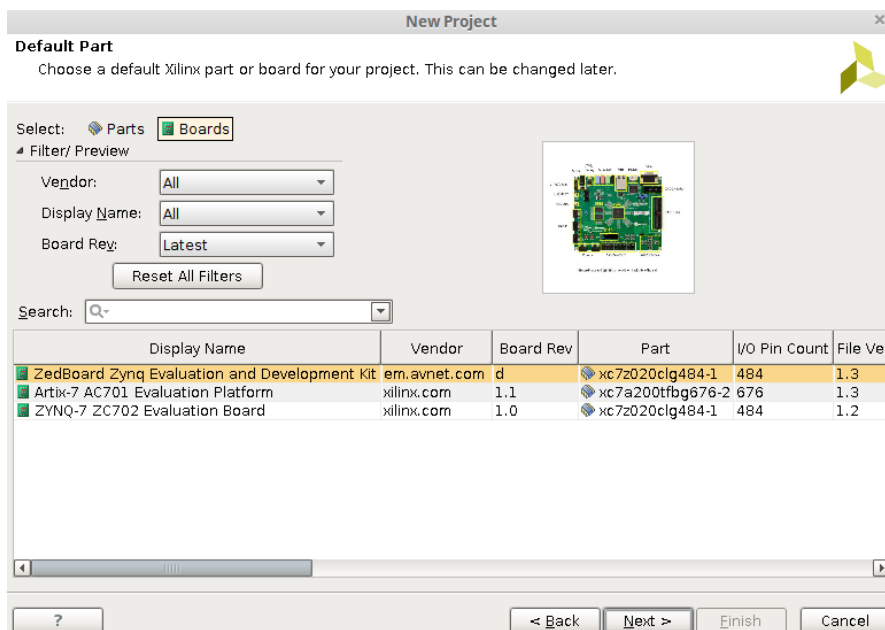


Figure C.4: Select what platform you are using

Select the board you are using. We are using the "Zedboard" so we select this one. Click next and verify that the summary section is correct for your project and click Finish. (Figure C.4)

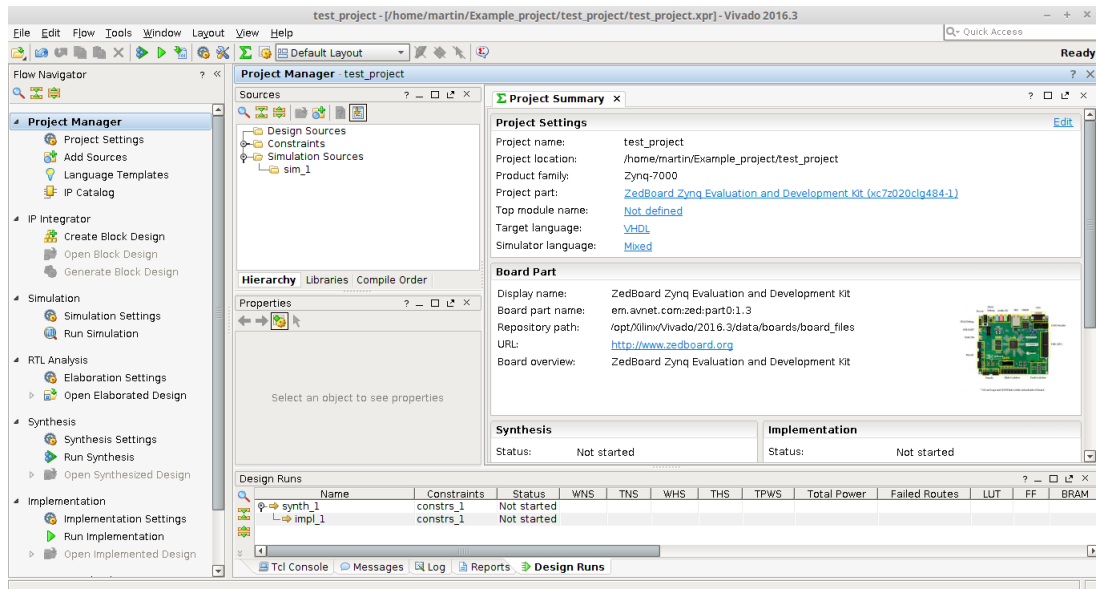


Figure C.5: GUI of Vivado

This brings up the project manager in the Graphical user interface (GUI) of Vivado. Look at the "Flow Navigator" at the left and select "create Block Design" under the "IP Integrator" section. (Figure C.5)

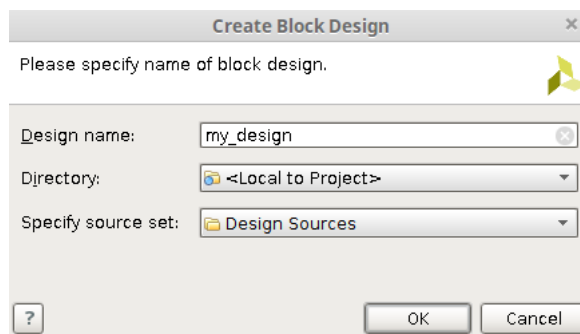


Figure C.6: Creating a block design

Select an appropriate name for your block design and click "next". The Block Design

page will open. (Figure C.6)

This design is empty. Press the  button to add IP.

Figure C.7: Add a new IP

Click the button to add an new IP seen in Figure C.7.

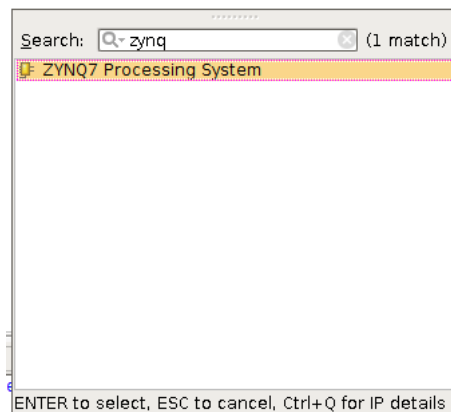


Figure C.8: Find the Zynq IP

Search for "zynq" and double click the "ZYNQ7 Processing System" to add it to the block design. (Figure C.8)

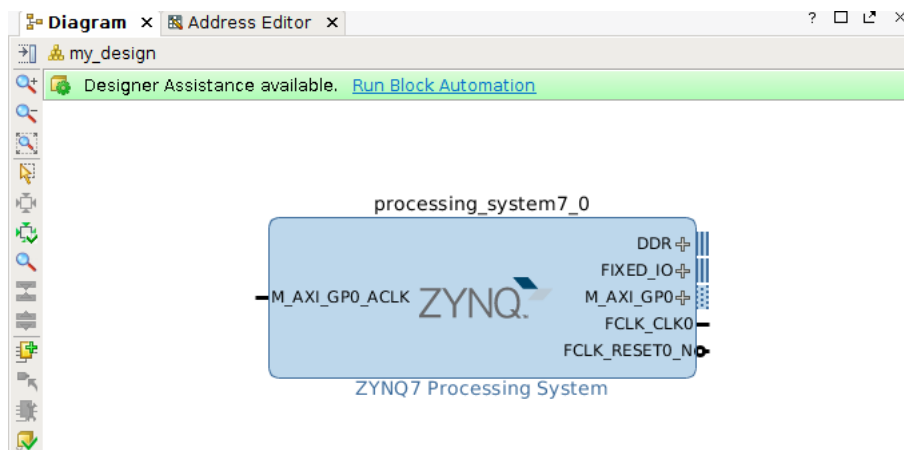


Figure C.9: Zynq added to design

Your block design will look like the Figure C.9. Click "run block automation" at the top to configure it to your board. When that is done double click the updated block to "customize block".

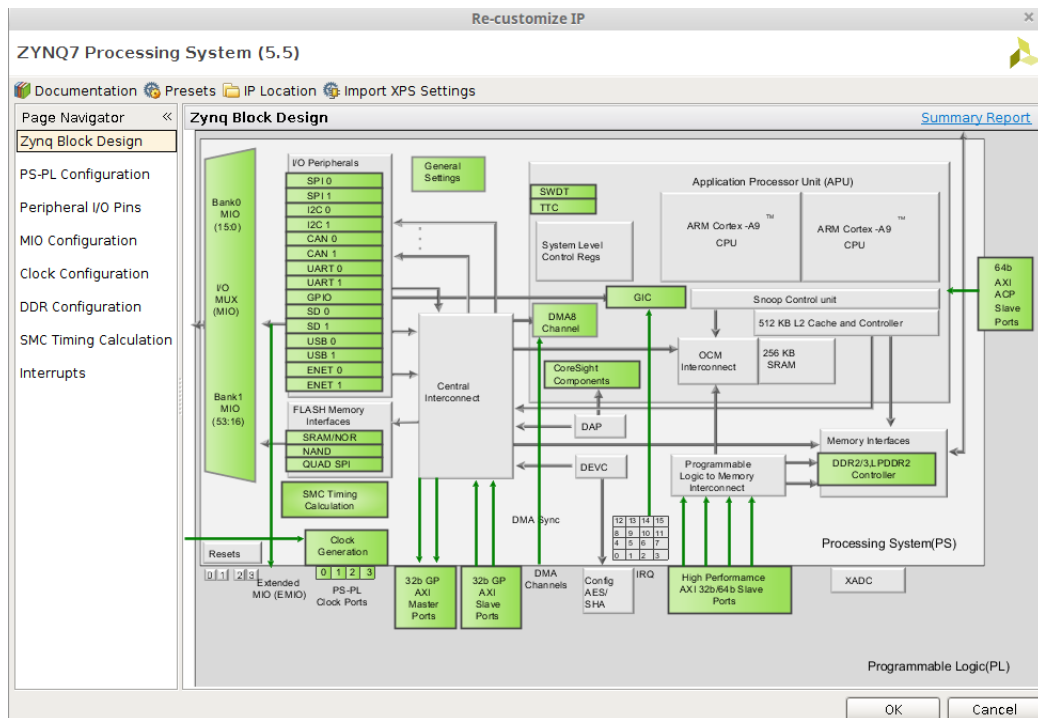


Figure C.10: Re-Customize IP

A new window will pop up, namely the "Re-customize IP" seen in Figure C.10.

1. Click the "MIO Configuration" panel
2. Expand the "IO Peripherals on the right and uncheck "ENET 0, SD 0, USB 0 and GPIO (GPIO IO)"
3. Keep the UART 1 selected. In the "MIO Configuration" panel expand "Application Processor Unit" and uncheck the "Timer 0"
4. Click the "Clock Configuration panel" and expand the "PL Fabric Clocks"
5. Select "FCLK\_CLK0" and choose what frequency you want the clock. For our design we choose 100 MHz

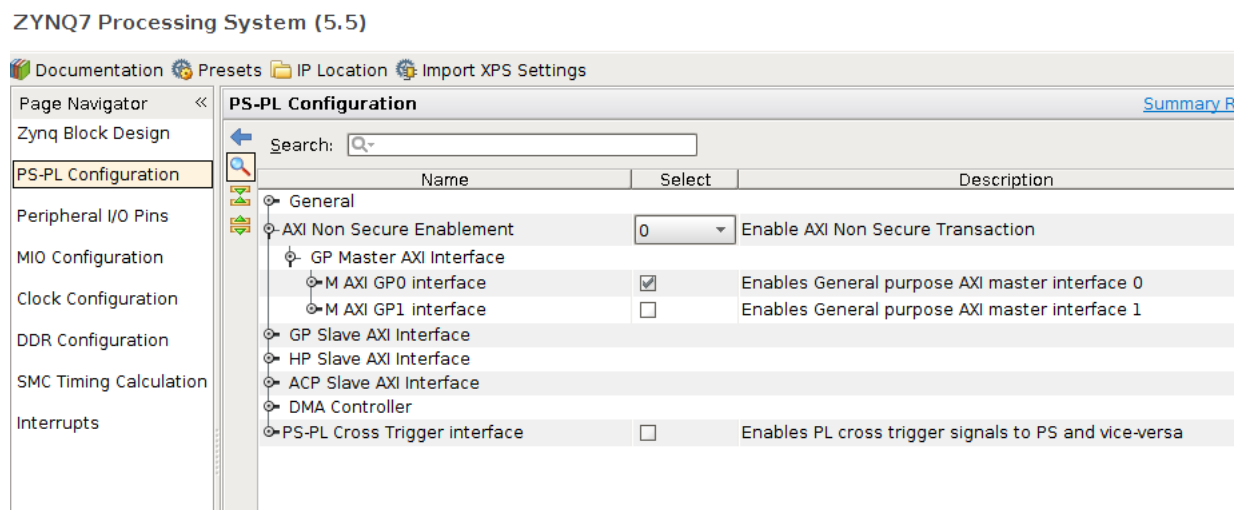


Figure C.11: PS-PL Configuration panel

Next click the "PS-PL Configuration" and expand the "GP Master AXI Interface" and make sure "M Axi GP0 interface" is selected. Now you can click "ok" to close the panel. (Figure C.11)

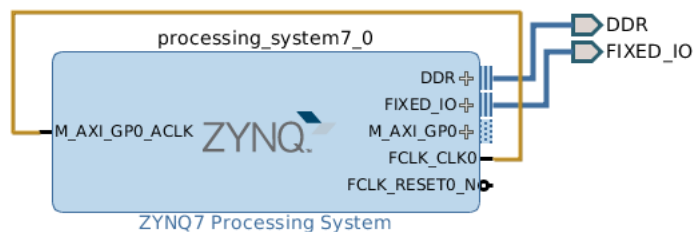


Figure C.12: Connect the wire highlighted in the block design

Now we have to connect the "FCLK\_CLK0" output to "M\_AXI\_GPO\_ACLK" input to use it as our clock. To do this click one of the former and drag to the other one and a wire will be traced just like in the Figure. (Figure C.12)

## C.1 Creating a new IP

Next we are going to create a custom IP. At the top panel select **"Tools" -> "Create and package new IP"**. Click next in the first window.

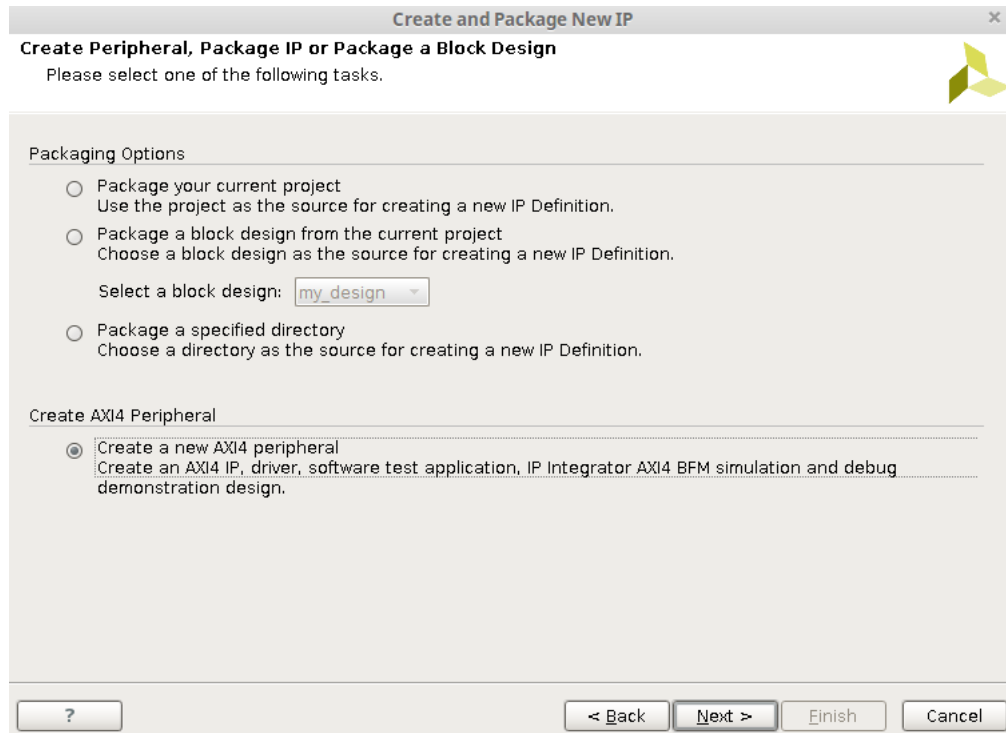


Figure C.13: Choose AXI4 peripheral

Then select "Create a New AXI4 peripheral shown in Figure C.13. Click "next". Add a suitable name and check the "overwrite existing" box and then click "next".



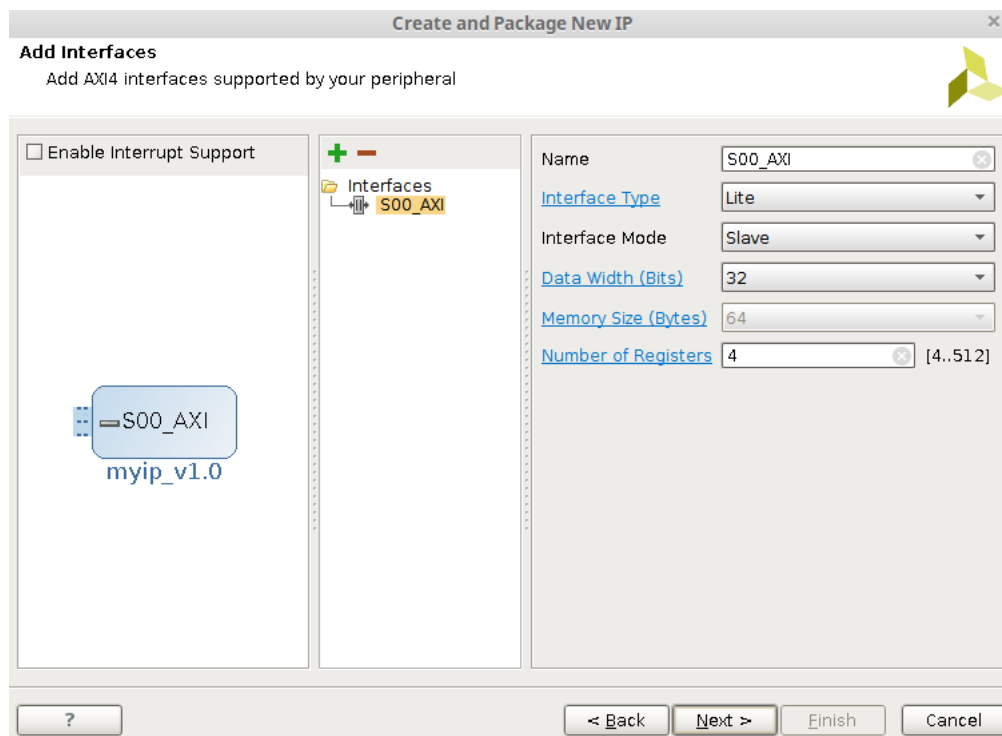


Figure C.14: Default settings for AXI Lite

Keep the default options just like the Figure C.14 and click "next". In the next section, select "Edit IP" and click "finish". Now the peripheral has been generated by Vivado and it is an AXI Lite slave with 4 x 32 bit read/write registers. Now we must modify it to our needs.

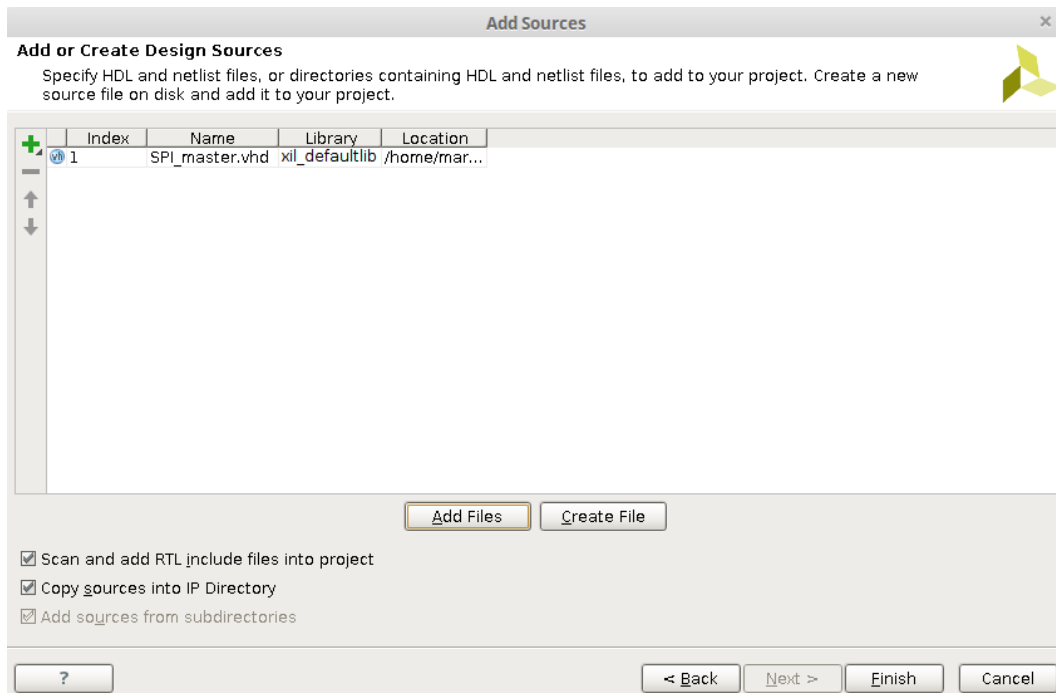


Figure C.15: Choosing the SPImaster.vhd

From the Flow Navigator click "Add Sources" and add the file you want. For our example we use the "SPI\_master.vhd" seen in Figure C.15. Make sure that the two options down to the left is checked. Click "finish".

Next you have to edit the spimaster\_v1\_0.vhd and spimaster\_v1\_0\_S00\_AXI\_inst.vhd. You can see how I edited the code in Appendix E.

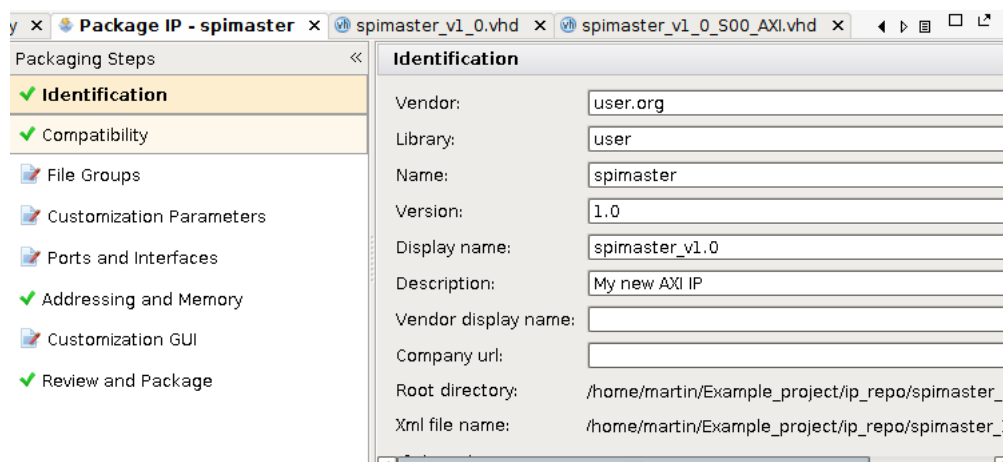


Figure C.16: Package IP panel

When you have configured the code to your liking, you go back to the package IP section and make sure everything is updated and looks correct. (Figure C.16)

1. Click the "File Groups" tab and click "merge changes from File Groups Wizard"
2. Then click "Customization Parameters" tab and click "Merge Changes from Customization Parameters Wizard"
3. Go to the "Review and Package" tab and press "Re-Package IP" and the customize IP project will close

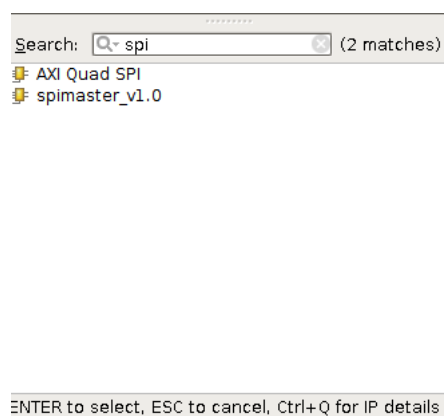


Figure C.17: Add your custom IP

Then you go back to the block diagram and add a new IP, namely the IP you just

created seen in Figure C.17.

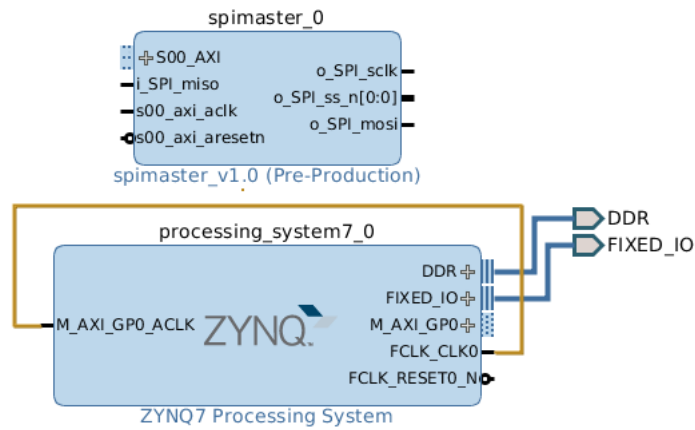


Figure C.18: Updated block diagram

The block diagram will look like in Figure C.18.

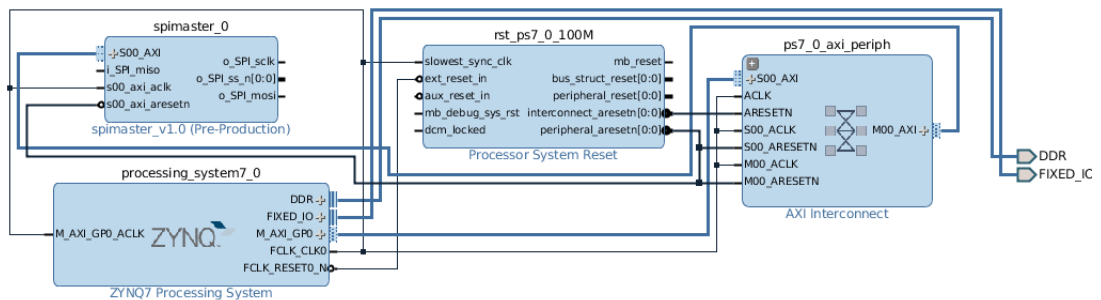


Figure C.19: Block diagram after connection automation

Now click "run connection automation" at the top and the block diagram will look like the Figure C.19.

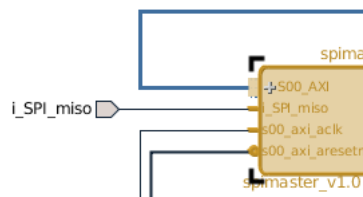


Figure C.20: Make inputs/outputs external

Right click on "i\_SPI\_miso" and click "Make external", do the same for "o\_SPI\_sclk", "o\_SPI\_ss\_n" and "o\_SPI\_mosi" seen in Figure C.20.

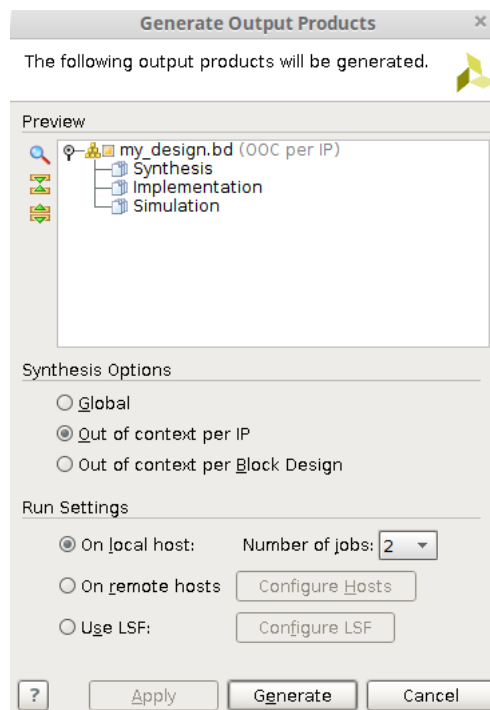


Figure C.21: Generate output products

Now you are going to generate the HDL design files.

1. Go to the "Project manager"
2. Under your design sources right click your design and click "Generate Output Products"

3. Click Generate seen in Figure C.21
4. Right click your design once again and click "Create HDL Wrapper"
5. Select "Let Vivado manage wrapper.." seen in Figure C.22 and click "ok".

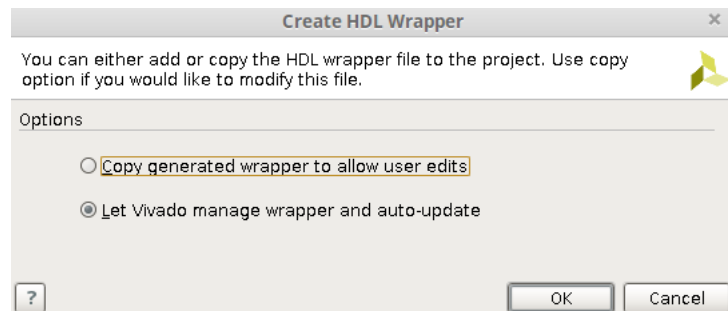


Figure C.22: Creating HDL Wrapper

Next go to flow navigator and look at the last panel "program and Debug" and select "Generate Bitstream". Click yes.

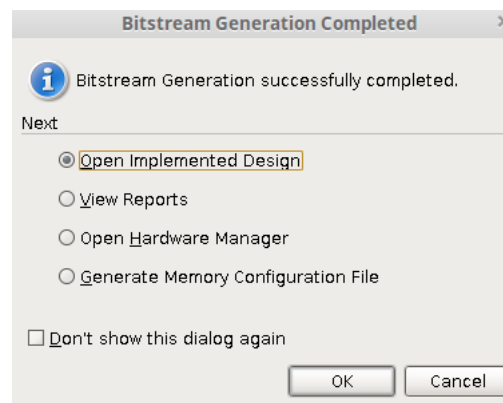


Figure C.23: Bitstream finished

This can take a while. When it is done with "synthesizing" and "implementation" and "generating bitstream" the pop-up seen in Figure C.23 will display. Click "Open Implementation Design" to configure the external i/o ports.

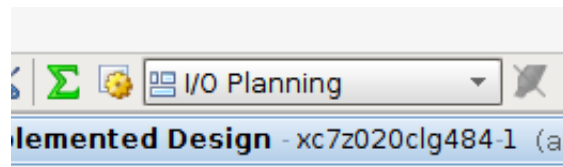


Figure C.24: I/O planning section

Make sure you are in the I/O planning environment at the top seen in Figure C.24. For our design we are using a Pmod to connect a temperature sensor (Appendix A) so we need to port the input/outputs to the correct addresses. These are listed in Appendix B and can be seen in Figure C.25.

I/O Ports									
	Name	Direction	...	B...	...	Package Pin	Fixed	Bank	I/O Std
+	All ports (134)								
+	DDR_42720 (71)	INOUT					<input checked="" type="checkbox"/>	502 (Multiple)*	
+	FIXED_IO_42720 (59)	INOUT					<input checked="" type="checkbox"/>	(Multiple) (Multiple)*	
+	o_SPI_ss_n (1)	OUT					<input checked="" type="checkbox"/>	13 LVCMOS25*	
+	o_SPI_ss_n[0]	OUT				AB10	<input checked="" type="checkbox"/>	13 LVCMOS25*	
+	Scalar ports (3)								
+	i_SPI_miso	IN				AB9	<input checked="" type="checkbox"/>	13 LVCMOS25*	
+	o_SPI_mosi	OUT				AA8	<input checked="" type="checkbox"/>	13 LVCMOS25*	
+	o_SPI_sclk	OUT				AB11	<input checked="" type="checkbox"/>	13 LVCMOS25*	

Figure C.25: I/O addressing

After you have edited the port planning, click save. A new window will pop-up asking if you want to add this into a new constraint file. Choose a suitable name and click "save".

Now we have to go to the panel "program and Debug" again and click generate bit-stream" to include these new constraints.

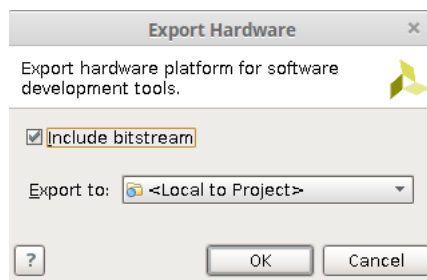


Figure C.26: Export Hardware

When this is done open block design from flow navigator and then click "File" at the top bar and "Export hardware" seen in Figure C.26, and make sure you have selected "include Bitstream". After that click File -> Launch SDK. Click "Ok" on the next pop-up window. Now we are in SDK and now you click File -> New -> Application Project

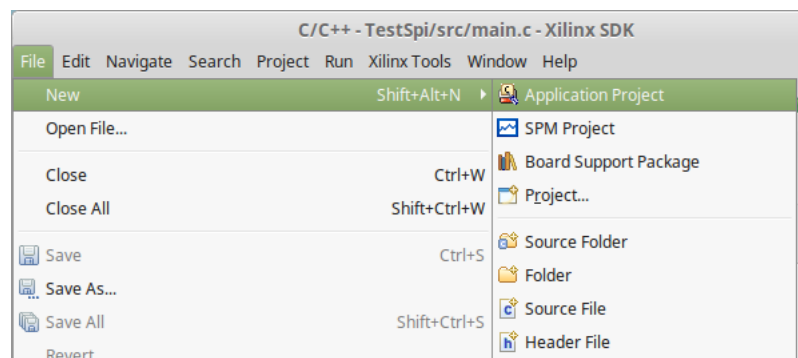


Figure C.27: Make a New Project

seen in Figure C.27. Give the project a suitable name and click "next" and double click the "hello World" template and then click "finish".



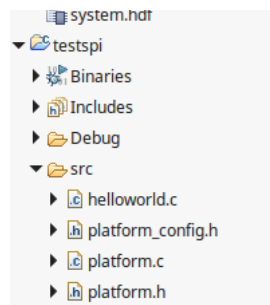


Figure C.28: Src folder for the project

Open the "testspi" folder and go into the "src" folder seen in Figure C.28. Open the "helloworld.c" source file. Replace this code with the code in Appendix E.3. Change the name "helloworld.c" to "main.c".

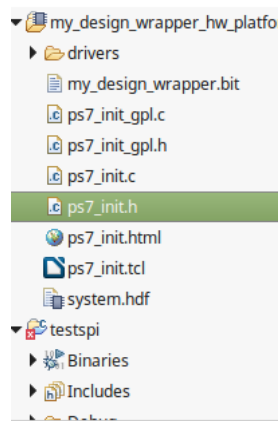


Figure C.29: Copy "ps7\_init.h" into your src folder

Copy the file "ps7\_init.h" and add it into your source file seen in Figure C.29.

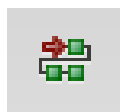


Figure C.30: Program FPGA

Turn on the Zedboard and connect one usb to JTAG and one to the UART port. Press the "Program FPGA" seen in the Figure C.30. and press "program" button. Now the FPGA is programmed.

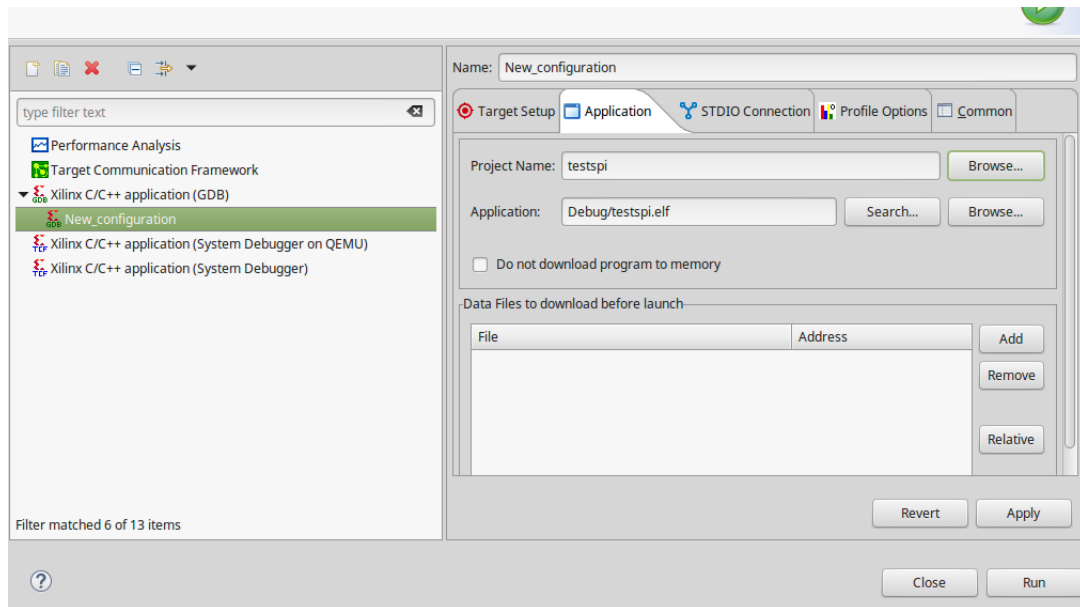


Figure C.31: Select test file

Click Run -> Run Configurations and then double click "Xilinx C/C++ application (GDB)". Go to the application tab and select your project. Then press "apply" seen in Figure C.31.

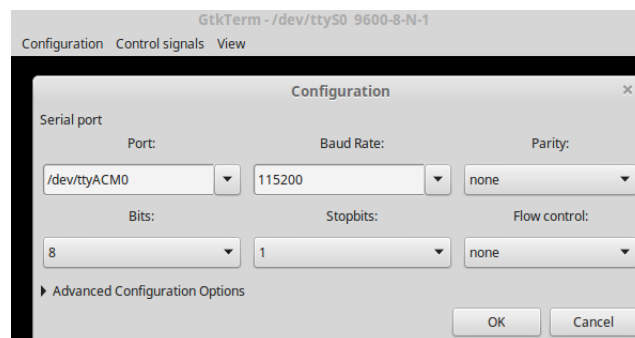
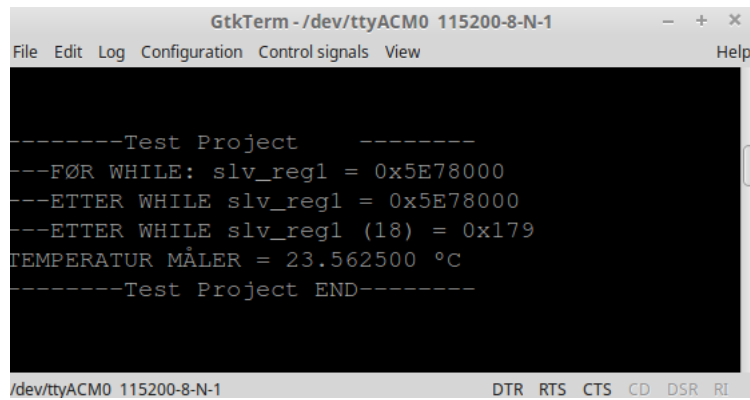


Figure C.32: Serial terminal port options

Now open a serial terminal to read out from UART. For this example I am using "Gtk-Term". Select the correct port for the UART usb and the correct baud rate seen in Figure C.32. Now you can go back to SDK and press run for the application.



The image shows a screenshot of a serial terminal window titled "GtkTerm - /dev/ttyACM0 115200-8-N-1". The window has a menu bar with "File", "Edit", "Log", "Configuration", "Control signals", "View", and "Help". The main area displays the following text:

```
-----Test Project -----  
--FØR WHILE: slv_reg1 = 0x5E78000  
--ETTER WHILE slv_reg1 = 0x5E78000  
--ETTER WHILE slv_reg1 (18) = 0x179  
TEMPERATUR MÅLER = 23.562500 °C  
-----Test Project END-----
```

The status bar at the bottom shows "/dev/ttyACM0 115200-8-N-1" on the left and "DTR RTS CTS CD DSR RI" on the right.

Figure C.33: Read data from UART

Now we get the outputs from our application program in the serial terminal seen in Figure C.33. This is the end of this tutorial.



# Appendix D

## SPI Master code

This is the VHDL code of the SPI master and the testbench for the SPI master.

### D.1 SPI Master

This is the VHDL code for the SPI Master module.

```
-----  
-- Company: NTNU  
-- Engineer: Martin Lesund  
--  
-- Create Date: 03/23/2017 03:19:50 PM  
-- Design Name:  
-- Module Name: SPI_master - Behavioral  
-- Project Name: Master thesis  
-- Target Devices:  
-- Tool Versions: vivado 2016.3  
-- Description: This is the master file for the SPI  
--  
-- Dependencies:  
--  
-- Revision:  
-- Revision 0.01 - File Created  
-- Additional Comments:  
-----
```

```

--PARAMETER description:
-- g_ = generic
-- i_ = input
-- o_ = output
-- r_ = register signal (has registered logic)
-- s_ = State Machine state
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SPI_master is
  generic (
    g_data_bus_width : integer := 16; -- Data bus width
    g_num_of_slaves  : integer := 1 ; -- Number of SPI slaves
    g_divide_value   : integer := 4 ; --input o_SPI_sclk freq = i_clock/(2*g_divide_value)
    -----CPOL and CPHA MODES -----
    g_CPOL           : std_logic := '1';
    g_CPHA           : std_logic := '1'
  );
  Port
  (
    -- USER LOGIC -----
    i_clock          : in    std_logic; -- System clock (50Mhz)
    i_reset_n        : in    std_logic; -- Async reset (active high)
    -----
    i_enable         : in    std_logic; -- Enable signal
    i_write_data     : in    std_logic_vector(g_data_bus_width-1 downto 0); -- Transmitted Data to slaves
    o_done           : out   std_logic; -- Transmission done signal
    o_read_data      : out   std_logic_vector(g_data_bus_width-1 downto 0); -- Received data from slave
    i_addr           : in    std_logic_vector(g_num_of_slaves-1 downto 0); -- Address of slave

    -- SPI MASTER INTERFACE -----
    o_SPI_sclk       : out   std_logic; -- SPI clock
    o_SPI_ss_n       : out   std_logic_vector(g_num_of_slaves-1 downto 0); -- Slave select
    o_SPI_mosi       : out   std_logic; -- SPI master out, slave in, serial data line
    i_SPI_miso       : in    std_logic -- SPI master in, slave out, serial data line
  )
end entity SPI_master;

```

```

);
end SPI_master;

architecture logic of SPI_master is
    type state is (s_idle, s_transmission, s_transmission_end);
    signal present_state, next_state : state;
    attribute enum_encoding: string;
    -- Optional attribute IMPORTANT FOR OPTIMAL USAGE (sequential vs. Gray vs. One-Hot)
    attribute enum_encoding of state: type is "00 01 11";
    -- Describes which encoding style used

    --Clock signals
    signal r_rise_sclk      : std_logic;
    signal r_fall_sclk     : std_logic;
    signal r_counter_enable : std_logic;
    signal r_counter       : integer range 0 to g_divide_value*2;
    signal r_counter_data  : integer range 0 to g_data_bus_width;
    signal w_counter_data  : std_logic;
    signal r_transmit_start : std_logic;
    signal r_tx_data       : std_logic_vector(g_data_bus_width-1 downto 0);
    signal r_rx_data       : std_logic_vector(g_data_bus_width-1 downto 0);
    signal r_slave         : integer := 0;      -- slave selected for current transaction

begin

w_counter_data <= '0' when(r_counter_data>0) else '1';

-----
-- Counter Clock generator
-----

p_spi_clock_gen : process (i_clock, i_reset_n)
begin
    if(i_reset_n = '0') then
        r_rise_sclk <= '0';
        r_fall_sclk <= '0';
        r_counter   <= 0;
    elsif (rising_edge (i_clock)) then
        if (r_counter_enable = '1' and g_CPOL = '1') then -- r_Mode11
            if (r_counter = g_divide_value-1) then -- first edge = fall

```

```

        r_rise_sclk <= '0';
        r_fall_sclk <= '1';
        r_counter   <= r_counter + 1;
    elsif(r_counter = (g_divide_value*2)-1) then
        r_rise_sclk <= '1';
        r_fall_sclk <= '0';
        r_counter   <= 0;
    else
        r_rise_sclk <= '0';
        r_fall_sclk <= '0';
        r_counter   <= r_counter+1;
    end if;
elseif (r_counter_enable = '1' and g_CPOL = '0' ) then  -- r_Mode01
    if (r_counter = g_divide_value-1) then  --first edge = rise
        r_rise_sclk <= '1';
        r_fall_sclk <= '0';
        r_counter   <= r_counter + 1;
    elsif(r_counter = (g_divide_value*2)-1) then
        r_rise_sclk <= '0';
        r_fall_sclk <= '1';
        r_counter   <= 0;
    else
        r_rise_sclk <= '0';
        r_fall_sclk <= '0';
        r_counter   <= r_counter+1;
    end if;
else
    r_rise_sclk <= '0';
    r_fall_sclk <= '0';
    r_counter   <= 0;
end if;
end if;
end process p_spi_clock_gen;

-----

-- FSM

-----

----- LOWER SECTION OF FSM: ----- (Sequential Circuit)
p_fsm_present_state : process ( i_clock, i_reset_n)

```



```

begin
    if (i_reset_n = '0') then
        present_state <= s_idle;
    elsif(rising_edge(i_clock)) then
        present_state <= next_state;
    end if;
end process p_fsm_present_state;

----- UPPER SECTION OF FSM: ----- (Combinational Circuit)
p_fsm_next_state: process (present_state, r_rise_sclk, r_fall_sclk, r_transmit_start,w_counter_data)
begin
    case present_state is
    when s_transmission =>
        if ((r_rise_sclk = '1' and g_CPOL = '1') or (r_fall_sclk = '1' and g_CPOL = '0')) and (w_counter_data = '1') then
            next_state <= s_transmission_end;
        else
            next_state <= s_transmission;
        end if;
    when s_transmission_end =>
        if (r_fall_sclk = '1' and g_CPOL = '1') or (r_rise_sclk = '1' and g_CPOL = '0') then
            next_state <= s_idle;
        else
            next_state <= s_transmission_end;
        end if;
    when others =>                                     --s_IDLE
        if r_transmit_start = '1' then
            next_state <= s_transmission;
        else
            next_state <= s_idle;
        end if;
    end case;
end process p_fsm_next_state;

----- OUTPUT SECTION: -----
p_fsm_outputs : process (i_clock, i_reset_n)
begin
    if(i_reset_n = '0') then
        r_transmit_start <= '0';
        r_tx_data        <= (others => '0');
        r_rx_data        <= (others => '0');
        o_done           <= '0';
    end if;
end process p_fsm_outputs;

```

```

o_read_data      <= (others => '0');

r_counter_data <= g_data_bus_width -1;
r_counter_enable <= '0';
o_SPI_sclk      <= g_CPOL;           -- Clock polarity
o_SPI_ss_n     <= (others => '1');   -- Set all slave select outputs high
o_SPI_mosi     <= '1';
elsif (rising_edge(i_clock)) then
  r_transmit_start <= i_enable;

case present_state is
  when s_transmission =>
    r_counter_enable <= '1';
    o_done          <= '0';
-----MODE11-----
    if(r_rise_sclk = '1' and g_CPOL = '1') then
      o_SPI_sclk      <= '1';
      r_rx_data      <= r_rx_data(g_data_bus_width-2 downto 0)&i_SPI_miso; --left shift
      if(r_counter_data>0) then
        r_counter_data <= r_counter_data -1;
      end if;
    elsif(r_fall_sclk = '1'and g_CPOL = '1')then
      o_SPI_sclk      <= '0' ;
      o_SPI_mosi     <= r_tx_data(g_data_bus_width-1);
      r_tx_data      <= r_tx_data(g_data_bus_width-2 downto 0)&'1';
-----MODE01-----
    elsif(r_fall_sclk = '1' and g_CPOL ='0')then
      o_SPI_sclk      <= '0' ;
      r_rx_data      <= r_rx_data(g_data_bus_width-2 downto 0)&i_SPI_miso; --left shift
      if(r_counter_data>0) then
        r_counter_data <= r_counter_data -1;
      end if;

    elsif(r_rise_sclk ='1' and g_CPOL ='0') then
      o_SPI_sclk      <= '1';
      o_SPI_mosi     <= r_tx_data(g_data_bus_width-1);
      r_tx_data      <= r_tx_data(g_data_bus_width-2 downto 0)&'1';

```

```

        end if;

        o_SPI_ss_n(r_slave) <= '0';

when s_transmission_end =>
    if (g_CPOL = '1' ) then
        o_done          <= r_fall_sclk;
        o_SPI_ss_n(r_slave) <= '0';
        o_read_data     <= r_rx_data;
        r_counter_data  <= g_data_bus_width-1;
        r_counter_enable <= '1';
    elsif(g_CPOL = '0') then
        o_done          <= r_rise_sclk;
        o_SPI_ss_n(r_slave) <= '0';
        o_read_data     <= r_rx_data;
        r_counter_data  <= g_data_bus_width-1;
        r_counter_enable <= '1';

    end if;

--s_idle
when others =>
    o_SPI_sclk <= g_CPOL;           --set spi clock polarity
    o_SPI_ss_n<= (others => '1');    -- Set all slaves to '1'
    o_SPI_mosi <= '1';
    r_tx_data <= i_write_data;
    o_done <= '0';
    r_counter_data <= g_data_bus_width-1;
    r_counter_enable <= '0';
    if(to_integer(unsigned(i_addr)) < g_num_of_slaves) then
        r_slave <= to_integer(unsigned(i_addr)); -- set current slave selection if valid
    else
        r_slave <= 0;                       -- set to first slave if not valid
    end if;

end case ;
end if;
end process p_fsm_outputs;

```

```
end logic;
```

## D.2 Test bench

This is the VHDL code for the SPI Master test-bench.

```
-----
-- Company: NTNU
-- Engineer: Martin Lesund
--
-- Create Date: 03/28/2017 03:04:23 PM
-- Design Name:
-- Module Name: SPI_master_tb - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description: This is the testbench for the SPI_master.vhd
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

-----
-- entity remains blank for testbench
entity SPI_master_tb is
end SPI_master_tb;
-----

architecture Behavioral of SPI_master_tb is

--Component ports
constant c_CLOCK_PERIOD : time := 10ns; -- 10ns = 100MHz
```

```

constant g_data_bus_width: integer := 16; --(16) number of bits to serialize
constant g_num_of_slaves : integer := 1; --Number of slaves
constant g_divide_value  : integer := 1;  --o_SPI_sclk freq = i_clock/(2*g_divide_value)
-----CPOL and CPHA MODES -----
constant g_CPOL          : std_logic := '1';
constant g_CPHA          : std_logic := '1';

signal i_clock           : std_logic := '0';
signal i_reset_n        : std_logic;

-----

signal i_enable          : std_logic;
signal i_write_data     : std_logic_vector(g_data_bus_width-1 downto 0);
signal o_done           : std_logic;
signal o_read_data      : std_logic_vector(g_data_bus_width-1 downto 0);
signal i_addr           : std_logic_vector(g_num_of_slaves-1 downto 0);

-----

signal o_SPI_sclk       : std_logic;
signal o_SPI_ss_n      : std_logic_vector(g_num_of_slaves -1 downto 0);
signal o_SPI_mosi       : std_logic;
signal i_SPI_miso       : std_logic;

-----

signal COUNT_RISE      : integer;
signal COUNT_FALL      : integer;
signal TEST_MOSI       : std_logic_vector(g_data_bus_width-1 downto 0); -- tx data
signal TEST_MISO       : std_logic_vector(g_data_bus_width-1 downto 0); -- rx data
signal r_addr          : integer;

-----

component SPI_master is
generic(
  g_data_bus_width : integer := 8;
  g_num_of_slaves  : integer;
  g_divide_value   : integer;
  g_CPOL           : std_logic;
  g_CPHA           : std_logic
);
port(

```

```

i_clock      : in  std_logic;
i_reset_n    : in  std_logic;

i_enable     : in  std_logic;
i_write_data : in  std_logic_vector(g_data_bus_width-1 downto 0);  -- Transmitted Data to slaves
o_done       : out std_logic;
o_read_data  : out std_logic_vector(g_data_bus_width-1 downto 0);
i_addr       : in  std_logic_vector(g_num_of_slaves-1 downto 0);

o_SPI_sclk   : out std_logic;
o_SPI_ss_n   : out std_logic_vector(g_num_of_slaves -1 downto 0);
o_SPI_mosi   : out std_logic;  -- SPI master out, slave in, serial data line
i_SPI_miso   : in  std_logic;
);
end component SPI_master;

begin
  --Component Instantiation
  -- Instantiate the Unit Under Test (UUT)
  UUT : SPI_master
  generic map(
    g_data_bus_width => g_data_bus_width,
    g_divide_value   => g_divide_value,
    g_CPOL           => g_CPOL,
    g_CPHA           => g_CPHA,
    g_num_of_slaves => g_num_of_slaves
  )
  port map (
    i_clock      => i_clock,
    i_reset_n    => i_reset_n,

    -----
    i_enable     => i_enable,
    i_write_data => i_write_data,
    o_done       => o_done,
    o_read_data  => o_read_data,
    i_addr       => i_addr,
    -- i_cont    => r_CONT,

    -- SPI MASTER INTERFACE -----
    o_SPI_sclk   => o_SPI_sclk,
    o_SPI_ss_n   => o_SPI_ss_n,
    o_SPI_mosi   => o_SPI_mosi,

```

```

        i_SPI_miso      => i_SPI_miso

    );

-- CLOCK Generation
p_CLK_GEN : process is
begin
    i_clock <= '0';
    wait for c_CLOCK_PERIOD/2;
    i_clock <= '1';
    wait for c_CLOCK_PERIOD/2;
end process p_CLK_GEN;

i_reset_n <= '0', '1' after 163ns;
i_addr <= std_logic_vector(to_unsigned(16#00#,g_num_of_slaves)); --Addressing slave
r_addr <= to_integer(unsigned(i_addr));
-- MAIN TESTING -----
-- FSM
p_control : process(i_clock, i_reset_n)
    variable v_CONTROL : unsigned(12 downto 0);
begin
    if(i_reset_n = '0') then
        v_CONTROL := (others => '0');
        i_enable <= '0';
        i_write_data <= std_logic_vector(to_unsigned(16#92#,g_data_bus_width));

    elsif(rising_edge(i_clock)) then
        v_CONTROL := v_CONTROL +1;
        if(v_CONTROL = 10) then -- 10
            i_enable <= '1';
        else
            i_enable <= '0';
        end if;

        if(o_done = '1') then
            i_write_data <= std_logic_vector(unsigned(i_write_data)+1);

        end if;
    end if;
end process;

```

```

end process p_control;

p_control_sclk : process(o_SPI_sclk)
begin
  if(i_reset_n = '0') then
    TEST_MISO <= std_logic_vector(to_unsigned(16#C9#,g_data_bus_width));
    TEST_MOSI <= std_logic_vector(to_unsigned(16#00#,g_data_bus_width));
    i_SPI_miso <= '0';
    COUNT_RISE <= 0;
    COUNT_FALL <= 0;
  else
-----MODE11-----
    if (rising_edge(o_SPI_sclk) and g_CPOL= '1')then
      if(o_SPI_ss_n(r_addr) = '0') then
        TEST_MOSI <= TEST_MOSI(g_data_bus_width-2 downto 0)&o_SPI_mosi;
        COUNT_RISE <= COUNT_RISE +1;
      else
        COUNT_RISE <= 0;
      end if;
    end if;

    if (falling_edge(o_SPI_sclk) and g_CPOL= '1')then
      if(o_SPI_ss_n(r_addr) = '0') then
        TEST_MISO <= std_logic_vector(rotate_right(unsigned(TEST_MISO),1));
        i_SPI_miso <= TEST_MISO(g_data_bus_width-1);
        COUNT_FALL <= COUNT_FALL +1;
      else
        COUNT_FALL <= 0;
      end if;
    end if;
-----MODE01-----
    if (falling_edge(o_SPI_sclk) and g_CPOL = '0')then
      if(o_SPI_ss_n(r_addr) = '0') then
        TEST_MOSI <= TEST_MOSI(g_data_bus_width-2 downto 0)&o_SPI_mosi;
        COUNT_FALL <= COUNT_FALL +1;
      else
        COUNT_FALL <= 0;
      end if;
    end if;

    if (rising_edge(o_SPI_sclk) and g_CPOL = '0') then

```



```
        if(o_SPI_ss_n(r_addr) = '0') then
            TEST_MISO <= std_logic_vector(rotate_right(unsigned(TEST_MISO),1));
            i_SPI_miso    <= TEST_MISO(g_data_bus_width-1);
            COUNT_RISE <= COUNT_RISE +1;
        else
            COUNT_RISE <= 0;
        end if;
    end if;
end process p_control_sclk;

end Behavioral;
```



# Appendix E

## AXI vhd files

### E.1 my\_spi\_v1\_0.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity my_spi_v1_0 is
  generic (
    -- Users to add parameters here
    g_num_of_slaves      : integer := 1;
    -- User parameters ends
    -- Do not modify the parameters beyond this line

    -- Parameters of Axi Slave Bus Interface S00_AXI
    C_S00_AXI_DATA_WIDTH      : integer      := 32;
    C_S00_AXI_ADDR_WIDTH     : integer      := 4
  );
  port (
    -- Users to add ports here
    o_SPI_sclk              : out   std_logic;                -- SPI clock
    o_SPI_ss_n              : out   std_logic_vector(g_num_of_slaves-1 downto 0); -- Slave select
    o_SPI_mosi              : out   std_logic;                -- SPI master out, slave in, serial data line
    i_SPI_miso              : in    std_logic;
    -- User ports ends
  );
end entity my_spi_v1_0;
```

```

-- Do not modify the ports beyond this line

-- Ports of Axi Slave Bus Interface S00_AXI
s00_axi_aclk      : in std_logic;
s00_axi_aresetn   : in std_logic;
s00_axi_awaddr    : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1 downto 0);
s00_axi_awprot    : in std_logic_vector(2 downto 0);
s00_axi_awvalid   : in std_logic;
s00_axi_awready   : out std_logic;
s00_axi_wdata     : in std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
s00_axi_wstrb     : in std_logic_vector((C_S00_AXI_DATA_WIDTH/8)-1 downto 0);
s00_axi_wvalid    : in std_logic;
s00_axi_wready    : out std_logic;
s00_axi_bresp     : out std_logic_vector(1 downto 0);
s00_axi_bvalid    : out std_logic;
s00_axi_bready    : in std_logic;
s00_axi_araddr    : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1 downto 0);
s00_axi_arprot    : in std_logic_vector(2 downto 0);
s00_axi_arvalid   : in std_logic;
s00_axi_arready   : out std_logic;
s00_axi_rdata     : out std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
s00_axi_rresp     : out std_logic_vector(1 downto 0);
s00_axi_rvalid    : out std_logic;
s00_axi_rready    : in std_logic
);
end my_spi_v1_0;

architecture arch_imp of my_spi_v1_0 is

-- component declaration
component my_spi_v1_0_S00_AXI is
    generic (
        C_S_AXI_DATA_WIDTH      : integer      := 32;
        C_S_AXI_ADDR_WIDTH     : integer      := 4
    );
    port (
        S_AXI_ACLK              : in std_logic;
        S_AXI_ARESETN          : in std_logic;
        S_AXI_AWADDR            : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        S_AXI_AWPROT            : in std_logic_vector(2 downto 0);

```

```

S_AXI_AWVALID      : in std_logic;
S_AXI_AWREADY      : out std_logic;
S_AXI_WDATA        : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
S_AXI_WSTREB       : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto 0);
S_AXI_WVALID       : in std_logic;
S_AXI_WREADY       : out std_logic;
S_AXI_BRESP        : out std_logic_vector(1 downto 0);
S_AXI_BVALID       : out std_logic;
S_AXI_BREADY       : in std_logic;
S_AXI_ARADDR       : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
S_AXI_ARPROT       : in std_logic_vector(2 downto 0);
S_AXI_ARVALID      : in std_logic;
S_AXI_ARREADY      : out std_logic;
S_AXI_RDATA        : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
S_AXI_RRESP        : out std_logic_vector(1 downto 0);
S_AXI_RVALID       : out std_logic;
S_AXI_RREADY       : in std_logic;
----- ADDED -----
o_SPI_sclk         : out      std_logic;           -- SPI clock
o_SPI_ss_n         : out      std_logic_vector(g_num_of_slaves-1 downto 0); -- Slave select
o_SPI_mosi         : out      std_logic;          -- SPI master out, slave in, serial data line
i_SPI_miso         : in       std_logic

);
end component my_spi_v1_0_S00_AXI;

begin

-- Instantiation of Axi Bus Interface S00_AXI
my_spi_v1_0_S00_AXI_inst : my_spi_v1_0_S00_AXI
  generic map (
    C_S_AXI_DATA_WIDTH      => C_S00_AXI_DATA_WIDTH,
    C_S_AXI_ADDR_WIDTH     => C_S00_AXI_ADDR_WIDTH
  )
  port map (
    S_AXI_ACLK      => s00_axi_aclk,
    S_AXI_ARESETN   => s00_axi_aresetn,
    S_AXI_AWADDR    => s00_axi_awaddr,
    S_AXI_AWPROT    => s00_axi_awprot,
    S_AXI_AWVALID   => s00_axi_awvalid,
    S_AXI_AWREADY   => s00_axi_awready,

```

```

S_AXI_WDATA      => s00_axi_wdata,
S_AXI_WSTRB     => s00_axi_wstrb,
S_AXI_WVALID    => s00_axi_wvalid,
S_AXI_WREADY    => s00_axi_wready,
S_AXI_BRESP     => s00_axi_bresp,
S_AXI_BVALID    => s00_axi_bvalid,
S_AXI_BREADY    => s00_axi_bready,
S_AXI_ARADDR    => s00_axi_araddr,
S_AXI_ARPROT    => s00_axi_arprot,
S_AXI_ARVALID   => s00_axi_arvalid,
S_AXI_ARREADY   => s00_axi_arready,
S_AXI_RDATA     => s00_axi_rdata,
S_AXI_RRESP     => s00_axi_rresp,
S_AXI_RVALID    => s00_axi_rvalid,
S_AXI_RREADY    => s00_axi_rready,
-----
o_SPI_sclk      => o_SPI_sclk,
o_SPI_ss_n     => o_SPI_ss_n,
o_SPI_mosi     => o_SPI_mosi,
i_SPI_miso     => i_SPI_miso

```

```
);
```

```
end arch_imp;
```

## E.2 my\_spi\_v1\_0\_S00\_AXI.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity my_spi_v1_0_S00_AXI is
  generic (
    -- Users to add parameters here
    g_data_bus_width      : integer := 16;           -- Data bus width
    g_num_of_slaves       : integer := 1 ;         -- Number of SPI slaves
    g_divide_value        : integer := 4 ;        -- input o_SPI_sclk freq = i_clock/(2*g_divide_value)

```

```

-----CPOL and CPHA MODES -----
g_CPOL          : std_logic := '1';
g_CPHA          : std_logic := '1';

-- User parameters ends
-- Do not modify the parameters beyond this line

-- Width of S_AXI data bus
C_S_AXI_DATA_WIDTH  : integer    := 32;
-- Width of S_AXI address bus
C_S_AXI_ADDR_WIDTH  : integer    := 4
);
port (
-- Users to add ports here
o_SPI_sclk          : out      std_logic;                -- SPI clock
o_SPI_ss_n          : out      std_logic_vector(g_num_of_slaves-1 downto 0); -- Slave select
o_SPI_mosi          : out      std_logic;                -- SPI master out, slave in, serial data line
i_SPI_miso          : in       std_logic;

-- User ports ends
-- Do not modify the ports beyond this line

-- Global Clock Signal
S_AXI_ACLK          : in std_logic;
-- Global Reset Signal. This Signal is Active LOW
S_AXI_ARESETN       : in std_logic;
-- Write address (issued by master, accepted by Slave)
S_AXI_AWADDR        : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
-- Write channel Protection type. This signal indicates the
-- privilege and security level of the transaction, and whether
-- the transaction is a data access or an instruction access.
S_AXI_AWPROT        : in std_logic_vector(2 downto 0);
-- Write address valid. This signal indicates that the master signaling
-- valid write address and control information.
S_AXI_AWVALID       : in std_logic;
-- Write address ready. This signal indicates that the slave is ready
-- to accept an address and associated control signals.
S_AXI_AWREADY       : out std_logic;
-- Write data (issued by master, accepted by Slave)
S_AXI_WDATA         : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
-- Write strobes. This signal indicates which byte lanes hold
-- valid data. There is one write strobe bit for each eight

```

```

        -- bits of the write data bus.
S_AXI_WSTRB      : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto 0);
-- Write valid. This signal indicates that valid write
-- data and strobes are available.
S_AXI_WVALID     : in std_logic;
-- Write ready. This signal indicates that the slave
-- can accept the write data.
S_AXI_WREADY     : out std_logic;
-- Write response. This signal indicates the status
-- of the write transaction.
S_AXI_BRESP      : out std_logic_vector(1 downto 0);
-- Write response valid. This signal indicates that the channel
-- is signaling a valid write response.
S_AXI_BVALID     : out std_logic;
-- Response ready. This signal indicates that the master
-- can accept a write response.
S_AXI_BREADY     : in std_logic;
-- Read address (issued by master, accepted by Slave)
S_AXI_ARADDR     : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
-- Protection type. This signal indicates the privilege
-- and security level of the transaction, and whether the
-- transaction is a data access or an instruction access.
S_AXI_ARPROT     : in std_logic_vector(2 downto 0);
-- Read address valid. This signal indicates that the channel
-- is signaling valid read address and control information.
S_AXI_ARVALID    : in std_logic;
-- Read address ready. This signal indicates that the slave is
-- ready to accept an address and associated control signals.
S_AXI_ARREADY    : out std_logic;
-- Read data (issued by slave)
S_AXI_RDATA      : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
-- Read response. This signal indicates the status of the
-- read transfer.
S_AXI_RRESP      : out std_logic_vector(1 downto 0);
-- Read valid. This signal indicates that the channel is
-- signaling the required read data.
S_AXI_RVALID     : out std_logic;
-- Read ready. This signal indicates that the master can
-- accept the read data and response information.
S_AXI_RREADY     : in std_logic
);

```



```

end my_spi_v1_0_S00_AXI;

architecture arch_imp of my_spi_v1_0_S00_AXI is

    -- AXI4LITE signals
    signal axi_awaddr      : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    signal axi_awready    : std_logic;
    signal axi_wready     : std_logic;
    signal axi_bresp      : std_logic_vector(1 downto 0);
    signal axi_bvalid     : std_logic;
    signal axi_araddr     : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    signal axi_arready    : std_logic;
    signal axi_rdata      : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal axi_rresp      : std_logic_vector(1 downto 0);
    signal axi_rvalid     : std_logic;

    -- Example-specific design signals
    -- local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
    -- ADDR_LSB is used for addressing 32/64 bit registers/memories
    -- ADDR_LSB = 2 for 32 bits (n downto 2)
    -- ADDR_LSB = 3 for 64 bits (n downto 3)
    constant ADDR_LSB : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
    constant OPT_MEM_ADDR_BITS : integer := 1;

    -----
    ---- Signals for user logic register space example
    -----

    ---- Number of Slave Registers 4
    signal slv_reg0      :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg1      :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg2      :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg3      :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg_rden   : std_logic;
    signal slv_reg_wren   : std_logic;
    signal reg_data_out   :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal byte_index     : integer;

    ----- ADDED -----
    signal spi_to_ps : std_logic_vector(31 downto 0);

    component SPI_master
    generic (
        g_data_bus_width : integer := 16;                -- Data bus width

```

```

g_num_of_slaves      : integer := 1 ;                               -- Number of SPI slaves
g_divide_value       : integer := 4 ;   --input o_SPI_sclk freq = i_clock/(2*g_divide_value)
-----CPOL and CPHA MODES -----
g_CPOL               : std_logic := '1';
g_CPHA               : std_logic := '1'
    );
port(
-- USER LOGIC -----
i_clock              : in   std_logic;                               -- System clock (50Mhz)
i_reset_n            : in   std_logic;                               -- Async reset (active high)
-----
i_enable             : in   std_logic;
i_write_data         : in   std_logic_vector(g_data_bus_width-1 downto 0); -- Transmitted Data to slaves
o_done               : out  std_logic;
o_read_data          : out  std_logic_vector(g_data_bus_width-1 downto 0); -- Received data from slave
i_addr               : in   std_logic_vector(g_num_of_slaves-1 downto 0); -- Address of slave
-----
-- SPI MASTER INTERFACE -----
o_SPI_sclk           : out  std_logic;                               -- SPI clock
o_SPI_ss_n           : out  std_logic_vector(g_num_of_slaves-1 downto 0); -- Slave select
o_SPI_mosi            : out  std_logic;                               -- SPI master out, slave in, serial data line
i_SPI_miso            : in   std_logic                               -- SPI master in, slave out, serial data line
);
end component;

begin
-- I/O Connections assignments

S_AXI_AWREADY        <= axi_awready;
S_AXI_WREADY         <= axi_wready;
S_AXI_BRESP          <= axi_bresp;
S_AXI_BVALID         <= axi_bvalid;
S_AXI_ARREADY        <= axi_arready;
S_AXI_RDATA          <= axi_rdata;
S_AXI_RRESP          <= axi_rresp;
S_AXI_RVALID         <= axi_rvalid;
-- Implement axi_awready generation
-- axi_awready is asserted for one S_AXI_ACLK clock cycle when both
-- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
-- de-asserted when reset is low.

```

```
process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_awready <= '0';
    else
      if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1') then
        -- slave is ready to accept write address when
        -- there is a valid write address and write data
        -- on the write address and data bus. This design
        -- expects no outstanding transactions.
        axi_awready <= '1';
      else
        axi_awready <= '0';
      end if;
    end if;
  end if;
end process;

-- Implement axi_awaddr latching
-- This process is used to latch the address when both
-- S_AXI_AWVALID and S_AXI_WVALID are valid.

process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_awaddr <= (others => '0');
    else
      if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1') then
        -- Write Address latching
        axi_awaddr <= S_AXI_AWADDR;
      end if;
    end if;
  end if;
end process;

-- Implement axi_wready generation
-- axi_wready is asserted for one S_AXI_ACLK clock cycle when both
-- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
-- de-asserted when reset is low.
```

```

process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_wready <= '0';
    else
      if (axi_wready = '0' and S_AXI_WVALID = '1' and S_AXI_AWVALID = '1') then
        -- slave is ready to accept write data when
        -- there is a valid write address and write data
        -- on the write address and data bus. This design
        -- expects no outstanding transactions.
        axi_wready <= '1';
      else
        axi_wready <= '0';
      end if;
    end if;
  end if;
end process;

-- Implement memory mapped register select and write logic generation
-- The write data is accepted and written to memory mapped registers when
-- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are used to
-- select byte enables of slave registers while writing.
-- These registers are cleared when reset (active low) is applied.
-- Slave register write enable is asserted when valid address and data are available
-- and the slave is ready to accept the write address and write data.
slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and S_AXI_AWVALID ;

process (S_AXI_ACLK)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      slv_reg0 <= (others => '0');
      slv_reg1 <= (others => '0');
      slv_reg2 <= (others => '0');
      slv_reg3 <= (others => '0');
    else
      loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
      if (slv_reg_wren = '1') then

```

```

case loc_addr is
  when b"00" =>
    for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
      if ( S_AXI_WSTRB(byte_index) = '1' ) then
        -- Respective byte enables are asserted as per write strobes
        -- slave register 0
        slv_reg0(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
      end if;
    end loop;
  when b"01" =>
    for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
      if ( S_AXI_WSTRB(byte_index) = '1' ) then
        -- Respective byte enables are asserted as per write strobes
        -- slave register 1
        slv_reg1(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
      end if;
    end loop;
  when b"10" =>
    for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
      if ( S_AXI_WSTRB(byte_index) = '1' ) then
        -- Respective byte enables are asserted as per write strobes
        -- slave register 2
        slv_reg2(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
      end if;
    end loop;
  when b"11" =>
    for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
      if ( S_AXI_WSTRB(byte_index) = '1' ) then
        -- Respective byte enables are asserted as per write strobes
        -- slave register 3
        slv_reg3(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
      end if;
    end loop;
  when others =>
    slv_reg0 <= slv_reg0;
    slv_reg1 <= slv_reg1;
    slv_reg2 <= slv_reg2;
    slv_reg3 <= slv_reg3;
end case;
end if;
end if;

```

```

    end if;
end process;

-- Implement write response logic generation
-- The write response and response valid signals are asserted by the slave
-- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
-- This marks the acceptance of address and indicates the status of
-- write transaction.

process (S_AXI_ACLK)
begin
    if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
            axi_bvalid <= '0';
            axi_bresp <= "00"; --need to work more on the responses
        else
            if (axi_awready = '1' and S_AXI_AWVALID = '1' and axi_wready = '1' and S_AXI_WVALID = '1' and axi_bv
                axi_bvalid <= '1';
                axi_bresp <= "00";
            elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then --check if bready is asserted while bvalid is
                axi_bvalid <= '0'; -- (there is a possibility that bready is always
            end if;
        end if;
    end if;
end process;

-- Implement axi_arready generation
-- axi_arready is asserted for one S_AXI_ACLK clock cycle when
-- S_AXI_ARVALID is asserted. axi_arready is
-- de-asserted when reset (active low) is asserted.
-- The read address is also latched when S_AXI_ARVALID is
-- asserted. axi_araddr is reset to zero on reset assertion.

process (S_AXI_ACLK)
begin
    if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
            axi_arready <= '0';
            axi_araddr <= (others => '1');
        else
            if (axi_arready = '0' and S_AXI_ARVALID = '1') then

```

```

        -- indicates that the slave has accepted the valid read address
        axi_arready <= '1';
        -- Read Address latching
        axi_araddr <= S_AXI_ARADDR;
    else
        axi_arready <= '0';
    end if;
end if;
end if;
end process;

-- Implement axi_arvalid generation
-- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
-- S_AXI_ARVALID and axi_arready are asserted. The slave registers
-- data are available on the axi_rdata bus at this instance. The
-- assertion of axi_rvalid marks the validity of read data on the
-- bus and axi_rresp indicates the status of read transaction. axi_rvalid
-- is deasserted on reset (active low). axi_rresp and axi_rdata are
-- cleared to zero on reset (active low).
process (S_AXI_ACLK)
begin
    if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
            axi_rvalid <= '0';
            axi_rresp <= "00";
        else
            if (axi_arready = '1' and S_AXI_ARVALID = '1' and axi_rvalid = '0') then
                -- Valid read data is available at the read data bus
                axi_rvalid <= '1';
                axi_rresp <= "00"; -- 'OKAY' response
            elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
                -- Read data is accepted by the master
                axi_rvalid <= '0';
            end if;
        end if;
    end if;
end process;

-- Implement memory mapped register select and read logic generation
-- Slave register read enable is asserted when valid address is available
-- and the slave is ready to accept the read address.

```

```

slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid) ;

process (slv_reg0, spi_to_ps, slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
    case loc_addr is
        when b"00" =>
            reg_data_out <= slv_reg0;
        when b"01" =>
            reg_data_out <= spi_to_ps; --- Changed from slv_reg1 to spi_to_ps
        when b"10" =>
            reg_data_out <= slv_reg2;
        when b"11" =>
            reg_data_out <= slv_reg3;
        when others =>
            reg_data_out <= (others => '0');
    end case;
end process;

-- Output register or memory read data
process( S_AXI_ACLK ) is
begin
    if (rising_edge (S_AXI_ACLK)) then
        if ( S_AXI_ARESETN = '0' ) then
            axi_rdata <= (others => '0');
        else
            if (slv_reg_rden = '1') then
                -- When there is a valid read address (S_AXI_ARVALID) with
                -- acceptance of read address by the slave (axi_arready),
                -- output the read data
                -- Read address mux
                axi_rdata <= reg_data_out; -- register read data
            end if;
        end if;
    end if;
end process;

-- Add user logic here

```



```

SPI_master_0 : SPI_master
generic map(
  g_data_bus_width => g_data_bus_width,
  g_num_of_slaves  => g_num_of_slaves,
  g_divide_value   => g_divide_value,
  -----CPOL and CPHA MODES -----
  g_CPOL          => g_CPOL,
  g_CPHA          => g_CPHA
)
port map(

  -- USER LOGIC -----
  i_clock  => S_AXI_ACLK,
  i_reset_n => slv_reg0(31),
  -----
  i_enable   => slv_reg0(30),
  i_write_data => slv_reg0(29 downto (30 - g_data_bus_width)), -- When g_databus = 16
  o_done     => spi_to_ps(31),
  o_read_data => spi_to_ps(30 downto (31 - g_data_bus_width)),
  i_addr     => slv_reg0(12 downto (13 - g_num_of_slaves)),
  -----
  -- SPI MASTER INTERFACE -----
  o_SPI_sclk => o_SPI_sclk,
  o_SPI_ss_n => o_SPI_ss_n,
  o_SPI_mosi => o_SPI_mosi,
  i_SPI_miso => i_SPI_miso
);
  -- User logic ends

end arch_imp;

```

## E.3 C Code Test Program

```

#include <stdio.h>
#include <stdlib.h>
#include "ps7_init.h"
#include "platform.h"
#include "xparameters.h"
#include "xil_printf.h"
#include "xil_io.h"

```

```

// Macros
#define CUSTOM_IP_BASEADDR 0x43C00000
#define REGISTER_0_OFFSET 0x00
#define REGISTER_1_OFFSET 0x04
#define REGISTER_2_OFFSET 0x08 //Not in use
#define REGISTER_3_OFFSET 0x0C //Not in use

//Function prototypes
void set_custom_ip_register(int baseaddr, int offset, int value);
int get_custom_ip_register(int baseaddr, int offset);

//MAIN
int main()
{
    init_platform();

    xil_printf("-----Test Project -----\n\r");
    int r_temp0;
    int r_temp1;
    r_temp1 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_1_OFFSET);

    //set 1 to reset
    set_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_0_OFFSET, 0x80000000);
    for(int i; i < 1000000; i++){
        r_temp0 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_0_OFFSET);
        for(int i; i < 1000000; i++){

    // set 0 to reset
    set_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_0_OFFSET, 0x00000000);
    for(int i; i < 1000000; i++){
        r_temp0 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_0_OFFSET);

    for(int i; i < 1000000; i++){
    // Set 1 to reset and enable
    set_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_0_OFFSET, 0xC0000000);
    r_temp0 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_0_OFFSET);

```

```

for(int i; i < 1000000; i++){
r_temp1 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_1_OFFSET);

for(int i; i < 1000000; i++){
//Read from register spi_to_ps
r_temp1 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_1_OFFSET);
xil_printf("slv_reg1 = 0x%02X\n\r", r_temp1);
set_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_0_OFFSET, 0x80000000);
r_temp1 = get_custom_ip_register(CUSTOM_IP_BASEADDR, REGISTER_1_OFFSET);
xil_printf("---slv_reg1 = 0x%02X\n\r", r_temp1);
xil_printf("---slv_reg1 (18) = 0x%02X\n\r", r_temp1>>18);

float e;

//Transform the data to celsius
e= (r_temp1>>18)* 0.0625;
printf("TEMPERATUR MÅLER = %.2f °C\n\r",e);

    xil_printf("-----Test Project END-----\n\n\n\r");

cleanup_platform();
return 0;
}

// Setting register function
void set_custom_ip_register(int baseaddr, int offset, int value)
{
    Xil_Out32(baseaddr + offset, value);
}
// Reading register function
int get_custom_ip_register(int baseaddr, int offset)
{
    int temp = 0;
    temp = Xil_In32(baseaddr + offset);
    return(temp);
}

```