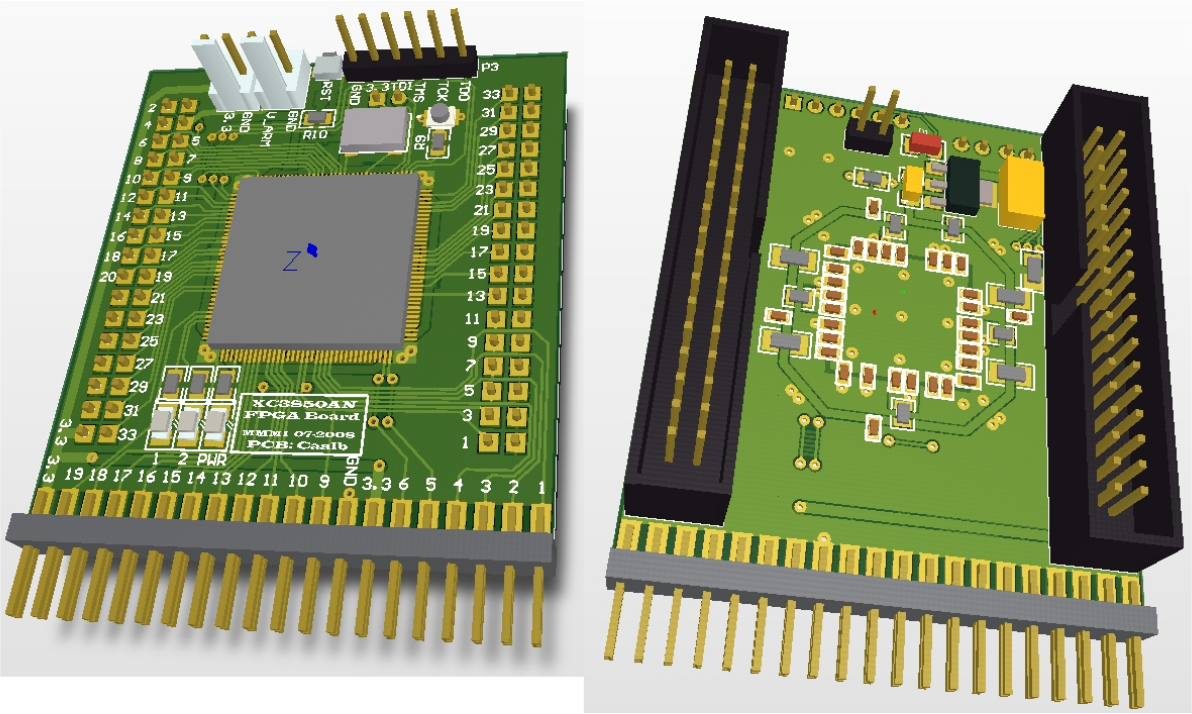
SPARTAN 3 EXPERIMENTATION BOARD

Preliminary USERS GUIDE

Anders Stengaard Sørensen & Carsten Albertsen

November 23, 2009



The AVR **S**partan **3** **Ex**perimentation Board (S3X), enable you to use with the Xilinx Spartan-3 XC3S50AN FPGA in your own circuits, without the hassle of advanced board design and SMD soldering.

Anders Stengaard Srensen & Carsten Albertsen

Contents

WARNING!

Connecting external circuits directly or indirectly to the ports of your PC may cause damage to your computer, if the external circuit it is not properly designed and tested. This is especially true if the external equipment operate with negative voltages, or voltages in excess of 5V.

Neither University of Southern Denmark, nor it's employees can take any responsibility for damage caused to your computer, or any other equipment, related to the use of the procedures or components described in this document.

You use the Spartan-3 experimentation board entirely at your own risk, so be careful!

Copyright notice

About the HOPE projects

Hands **O**n **P**rogrammable **E**lectronics — or HOPE , is a series of projects, aimed at promoting the use of programmable electronic components in research, development and students projects, related to Odense University College of Engineering.

While it is good educational practice to teach classical electronic design, based on discrete components and simple integrated circuits, it is also necessary to enable students to gain practical experience with the highly flexible and complicated devices used in practical electronics today.

As I began teaching in 2003, I was surprised to see the complex circuits students were designing with 74.. and 40.. type IC's, to realize registers, counters, decoders and other small digital systems, that could be realized much easier (and cheaper) in a Programmable Logic Device (PLD) or even in a micro controller. I was even more surprised to learn that most of the students had actually followed courses in PLD's and micro controllers, but thought it too abstract or troublesome to transfer their experience with PLD or micro controller demonstration systems to a practical design in its own contexts.

In order to reduce the *entry barrier* towards programmable electronics, I have initiated a number of small projects, resulting in a series of tools, that should make it easier to begin working with selected PLD's, micro controllers etc. I have launched these projects under the common title *Hands On Programmable Electronics*, with subtle reference to the first commandment of the Hacker Ethic:

> *Access to computers — and anything which might teach you something about the way the world works — should be unlimited and total. Always yield to the Hands-On Imperative!* *(MIT students* ~ 1960*)*

It is our HOPE that the tools provided by the HOPE projects will result in increased use of CPLD's, micro controllers, FPGA's, FPAA's and other programmable electronics in students projects, as well as R&D projects in corporation with the Embedix group and RoboLab at University of Southern Denmark.

Anders Stengaard Srensen — 2004

1 Introduction

The Spartan-3 Experimentation board described here will enable you to work with Xilinx FPGA technology in your own electronic designs, without the hassle of advanced board design/manufacture, or SMD soldering. The board is primarily intended for rapid prototyping related to courses and students projects at University of Southern Denmark (SDU), and the design goals have been:

- Low cost, to put FPGA technology within the reach of everyone.
- Ease of use, to enable anyone to get started with FPGA technology in a few hours.
- Compatibility with two popular micro controller technologies: The Atmel atMEGA8-P IC, and the Olimex Arm-7 demo board.

We have chosen to base this design on Xilinx Spartan-3 XC3S50AN FPGA, for the following reasons:

- Xilinx Spartan-3 family is currently the most popular FPGA technology at SDU, finding it's way into most courses, development and research projects involving FPGA's. All the knowledge you gain from working with this board, should thus be directly re-useable later on in your education.
- Although the ..3S50.. type, with its 50,000 gates is the smallest FPGA in the Spartan-3 family, it is also the cheapest.
- The ..AN series, has a build in configuration FLASH memory, enabling the FPGA to retain it's configuration when powered off.
- The XC3S50AN is available in a PQFP package, enabling the board to be manufactured on a low-cost 2-layer PCB. This fact also enable students to design their own PCB's utilizing the same FPGA, by copying parts of the design presented here.

You will probably be assembling and using the Spartan-3 experimentation board as part of one of your first projects with FPGA's, very likely planning to use it with an ATmega8 CPU, or an Olimex ARM-7 board, which we usually recommend — and keep on stock — for students projects at Odense University College of Engineering.

This guide is written in order to fulfill 3 different purposes:

- As an assembly guide.
- As a user guide
- As a design description, in order to teach basic FPGA design aspects.

If you follow the guidelines and reference designs given below, you should have a FPGA system up and running later to-day.

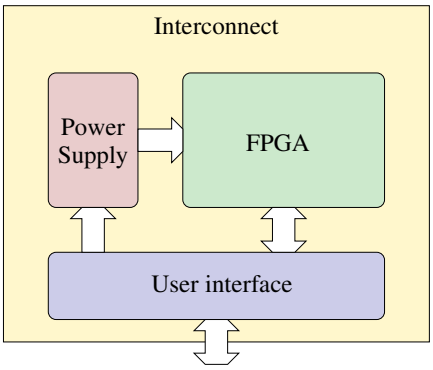
Good luck!

2 Theory of operation

This chapter will describe the theory of operation of the Spartan-3 Experimentation Board (S3XB).

The S3XB is actually really simple, and can conveniently be described as a composition of the following 4 subsystems:

FPGA: The XC3S50AN FPGA, which is the defining component of the S3XB.
Power supply: Suppling the FPGA with stable supply voltages.
User interface: Connectors, LED's and pushbutton.
Interconnect: As the FPGA is a high speed device, the design of the PCB, interconnecting the components, is not trivial.



2.1 The FPGA

An FPGA is basically an IC package, that can be configured to take on any digital functionality, from a NOT-gate, to a processor, limited only by the amount of gates, interconnections, and propagation delays within the FPGA.

Generally, FPGA's are organised with the following 3 main components:

The Logic core is the gate array, that is implementing the logical functionality of the FPGA. It consist of a matrix of configurable gates, overlaid by a lattice of wire traces, crisscrossing the array. A configurable switch at each intersection enable configuration of the connections between gates.

I/O ring: Signals in the logic core are connected to the external world through configurable I/O buffers, that can typically be configured for various voltage levels and signal types. These I/O buffers are physically placed as a 'ring' around the logic core.

Programming interface: All the 'switches' that implement the configurability of gates, wires and I/O buffers, have to be 'programmed' from an external system. Thus the FPGA need a subsystem that will allow an external system to transfer the configuration to the switches inside the FPGA. In Xilinx FPGA's, this is done, using the JTAG standard, which can also be used for various debugging purposes beside configuring the device.

In Spartan-3 FPGA's, the I/O buffers are organised in a number of I/O banks, each with a number of generic I/O buffers, combined with some dedicated I/O functions, for instance dedicated clock inputs. Combined with the interface to the programming interface, this leads to a somewhat complicated mapping between the physical pins of the IC, and the I/O buffers and programming

interface inside the IC, as some of the pins have more functions, depending on the configuration of the FPGA. One of the design efforts, thus becomes to choose the mapping to FPGA pins.

The configuration 'switches' in a Spartan-3 FPGA are based on SRAM technology, causing the FPGA to retun to a 'blank' stage when powered off. A non volatile memory, is thus needed to transfer the configuration to the FPGA at power-on. Traditionally, external flash memories have been required for this, but in the Spartan-3 ...AN series, Xilinx have placed the FLASH memory inside the IC package, which reduces the complexity of the PCB layout.

2.2 Power supply

The power supply has two objectives:

- To provide the voltages needed by the FPGA (1.2V & 3.3V)
- To keep the voltages stable at the entire frequency range, relevant for the FPGA ($DC \ldots 1Ghz$)

The Power supply can be broken down into three subsystems:

Input protection: Will provide *some measure* of protection against users accidentally supplying too high a voltage, or polarzing the supply wrong. This protection is however quite weak, so **be very carefull when connecting the supply**.

1.2V regulator: Will reduce the 3.3V external supply voltage to 1.2V, needed by the logic core of the FPGA.

Decoupling: A network of decoubling capacitors will ensure that the voltages remain stable throughout the relevant frequency range.

2.2.1 Over voltage protection

In order to avoid damage by accidentially supplying the S3XB with e.g. 5V, a zener diode (*D1*) have been placed across the 3.3V supply. If the supply voltage rises above 3.3V, the diode will begin conducting current. If the supply is current-limited, with the limit set to a low setting — e.g. 100mA — the zener diode combined with the current limitter should prevent the voltage from rising critically above 3.3V. After a short amount of time, *D1* will overheat, so it is important to disconnect the supply as soon as the current limit kicks in. If the supply is not limited to a low current setting, *D1* will be destroyed by heat before the user can react, most likely causing the overvoltage do destroy the FPGA as well.

if the supply voltage is polarized the wrong way, *D1* will act as an ordinary diode, short circuiting the supply, until the diode overheats and is destroyed. If the supply is current limited to a low setting, there should be enough time to react.

2.2.2 The 1.2V regulator

To keep things simple, the S3XB have been designed to be used with a $3.3V$ external supply, provided by the user. So only a single 1.2V regulator is needed, to create a 1.2V supply from the external 3.3V.

The reduction from 3.3V to 1.2V is done with an integrated linear regulator: NCP566 — designated *VR1* in the diagram. As with all linear regulators, the voltage drop is achieved by converting electrical energy into heat, at a rate of:

$$P_{heat} = (V_{out} - V_{in}) \times I_{out}$$

Where I_{out} is the current consumed by the FPGA logic core.

As the voltages are given as $V_{in} = 3.3V$ and $V_{out} = 1.2V$, the regulator will generate a total amount of heat:

$$P_{heat} = (3.3V - 1.2V) \times I_{out} \Leftrightarrow P_{heat} = 2.1\frac{W}{A} \times I_{out} \quad (1)$$

Although the regulator can operate at temperatures up to $120°C$, it can still give some unpleasant burns to an unsuspecting user if it gets too hot. As the regulator and its mounting has a total thermal resistance of up to $100\frac{K}{W}$, we recommend that the power dissipation is kept below $500mW$, coresponding to a current consumption below $250mA$.

2.2.3 Decoupling

Voltage fluctuations on the supply is a result of an uneven current consumption, caused by the on/off nature of digital electronics. As the internal switches of the FPGA change state, so does the amount of current used by the FPGA. The frequency spectrum of the current cunsumption is defined by the switching times of the device, which is down to 300ps for the FPGA used here. 300ps switching times corespond to a frequency range of up to approximately 1GHz

The stability of the voltage at DC and low frequencies[1], are handled by the voltage regulators, used to create the 3.3V and 1.2V supplies. As the bandwidth of the control loop in typical voltage regulators is normally measured hundreds - thousands of hertz, the stability at higher frequencies, are ensured by using decoupling capacitors.

One way of thinking of a decoupling capacitor is, that it's low impedance at high frequencies, will 'short circuit' high frequency voltage fluctuations. So we only need to choose a capacitor with a sufficiently low impedance at the approprate frequency range. If the impedance is kept below — say 0.1Ω, it would take a current of $1A$ to alter the voltage more that $100mV$.

[1] Up to the hundred hertz range

Ideally a capacitor has an impedance $|X_c| = \frac{1}{2\pi fC}$ which becomes ever lower as the frequency increase. So decoupling the power supply with a sufficiently large capacitor, should ensure that voltage fluctuations are effectively short circuited. The $100\mu F$ capacitor *C17* should thus have an impedance of 16Ω at $f = 100Hz$, $1,6\Omega$ at $1kHz$, $160m\Omega$ at $10kHz$ and so on.

Unfortunately, all practical capacitors have a some rather annoying parasitic components included:

Parasitic resistances: R_c because of the thin metal they are made from internally.

Parasitic inductance: L_c because of their internal geometry and the geometry of their connecting wires.

So a more realistic model for capacitor impedance is:

$$|X_c| \simeq \frac{1}{2\pi fC} + R_C + 2\pi f L_C$$

It is thus clear, that the impedance can never become lower than R_C, and also that the impedance will actually begin to rise as a function of frequency, when the frequency is high enough to let the term $2\pi f L_C$ become larger than the term $\frac{1}{2\pi fC}$

The point where the parasitic inductance term begins to dominate over the capacitance, is known as the resonant frequency of the capacitor, and is given as:

$$f_{res} = \frac{1}{2\pi\sqrt{C \times L_C}}$$

So, in short, the impedance of a capacitor will decrease with frequency, until reaching f_{res}, where it will begin to increase again. Another way to express this is that: A capacitor works as capacitor up till the resonant frequency, whereafter it begins working as an inductor.

For ceramic decoupling capacitors, the parasitic inductance is primarily a function of the geometry of the package, and the wires leading from the package to the device that is supposed to be decoupled. This inductance can be roughly estimated as a constant times the total length of wires plus capacitor: $L_C \simeq 1.5nH/mm$ Assuming an SMD 0603 capacitor (1.5mm long) mounted with total wirelength of 3mm, then gives a parasitic inductance of $L_C \simeq (3mm + 1.5mm) \times 1.5nH/mm \simeq 7nH$.

As the parasitic inductance is independant of the capacitance, it is evident, that capacitors with identical packages, but different values for capacitance will also have different resonant frequencies. Using the $7nH$ example from above, the resonant frequency will be $6MHz$ for 100nF, $19MHz$ for $10nF$, $60Mhz$ for $1nf$, and $190MHz$ for $100pF$.

In order to provide a sufficiently low impedance all the way from a few KHz to GHz, decoupling is designed as a network of parallel connected capacitors with a variety of values. The effecitive wire length are kept as short as possible by using SMD capacitors, placed directly between the wide GND and power areas of the PCB. On top of that the parasitic capacitance between PCB traces contribute to the decoupling at the highest frequencies.

2.3 User interface

The user interface consist of the following parts:

I/O connectors: A variety of pinheaders that provide connection to external systems.

Programming interface: JTAG connecotr and pushbutton.

LED's One for 'power' and two that can be controlled by the FPGA.

testpoint: A single high speed testpoint.

2.3.1 I/O connectors

There are 4 sets of connecotrs:

Bottom: For breadboard or prototyping PCB's.

Left: For cable, prototyping PCB or an Olimex ARM-7 module (only part of the connector is used for the ARM-7 module).

Right: For cable or PCB connection.

Olimex: For providing an Olimex ARM-7 module with +5V from external source.

Due to the flexibility in mapping functionality to the I/O pins, the connectors have simply been placed around the FPGA IC, and connected to whatever FPGA pins were most accesible for routing the connections. In this way, aproximatly 50% of the FPGA I/O pins have been mapped to the I/O connectors. A comprehensive table of the pin to pin mapping is available in appendix **??**

Some care have been taken to achieve the following goals:

- The bottom connector can be used to connect the S3XB to a standard breadboard for easy experiemntation in a learning environment.
- The pin configuration of the bottom connector allows the S3XB to be placed directly next to a DIL version of the atMEGA-8 microcontroller from Atmel. So the S3XB can easily be connected to thar microcontroller.
- The left connector is configured so an Olimex ARM-7 module can be placed directly on top of the S3XB as a 'daughter-board'. To support this, a 1×2 pinheader has been placed on the S3XB to provide +5V as well as mechanical support for the Olimex ARM-7 board.
- No dedicated clock input pins on the FPGA are wasted, but they are all connected to either the on-board clock generator, or I/O connectors.

2.4 Interconnect

In order to keep cost down, we decided to use a standard 2 layer 1.5mm PCB as base for the interconnect system. Due to the high-speed nature of the FPGA, the signals are laid out as micro striplines, so the bottom side is assigend as ground plane for the signals. In areas where no signals are passing, the bottom side is used to implement the power supply, including decoupling.

In order to ensure low impedance ground connections between the FPGA and the ground plane, the area below the FPGA, on the top side is assigned as a local ground-plane, connected to the bottom side ground plane with two parallel vias at each corner of the FPGA.

3 Assembling the SXB

3.1 Identifying the PCB

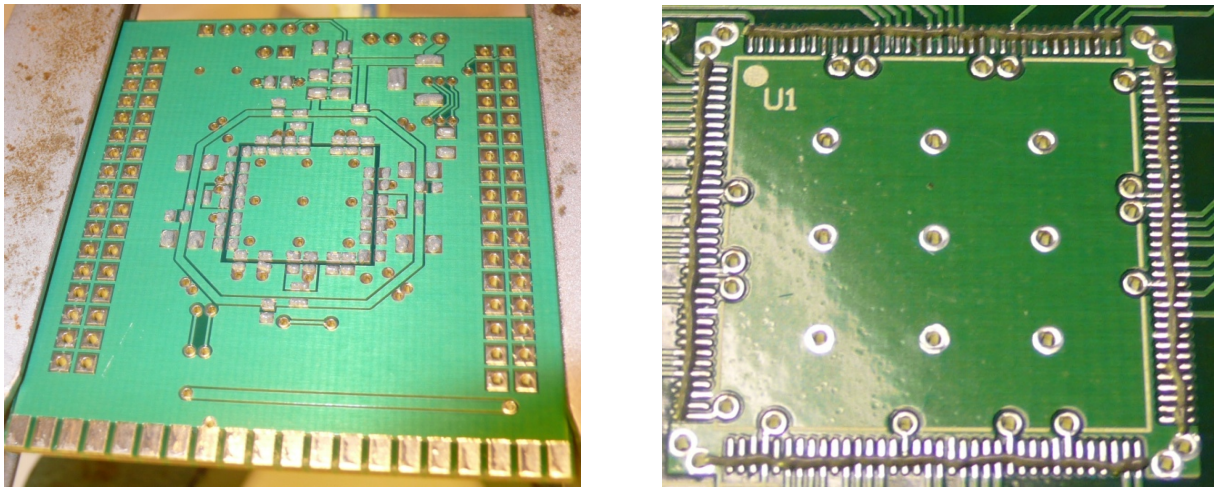
To be written

3.2 Identifying the components

To be written

3.3 Applying solder paste

Apply solder paste on every SMD pad on both sides of the board. Pay special attention to applying enough paste to the components: *VR1* and *C17*. Use a PCB holder to fixate the PCB so paste can be applied to both sides of the board.

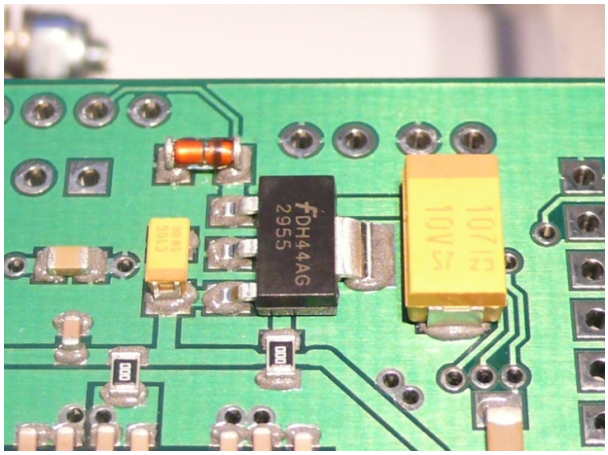


(a) On the bottom side

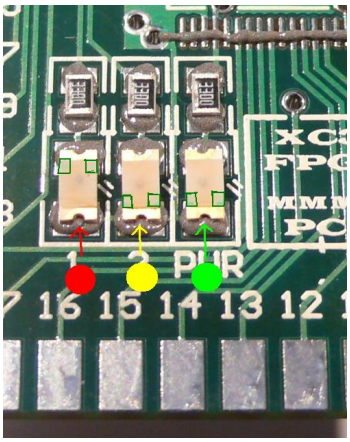
(b) On the FPGA footprint

Figure 1: Solder paste applied to the PCB

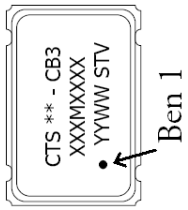
The solder paste for the FPGA IC, is easies to apply, by dispensing a thin thread of paste along the four sides of the footprint, as shown in figure 1(b)



(a) Closeup of bottom side



(b) LED's placement on top side



(c) Oscillator

Figure 2: Mounting details

3.4 Placing SMD components with vacum twezers

1. Place all the components on the bottom side of the PCB, as shown in figure 16(b). Please note that *C20, C17 and D1* are polarized, and must be placed in the right direction. The marking on the components are as follows:

   D1: (Zener diode), the cathode is marked with a dark ring.

   C17: (Tantalium electrolytic capacitor), the positive electrode is marked with a brown stripe.

   C20: (Tantalium electrolytic capacitor), the positive electrode is marked with a brown stripe.

   Mount as shown in figure 16(b), and pay special attention to pressing *VR1* and *C17* hard down on the PCB.

2. Turn the PCB over, very carefully, and mount all the SMD components, except the FPGA IC, on the top side of the PCB, as shown in figure 16(a).

   On the top side, the following components have to be turned the right way:

   X1 (Crystal oscillator) Pin 1 is marked with a dot on the package, and is mounted as shown in figure 2(c).

   D2 (Red LED) Two green dots mark the anode

   D3 (Yellow LED) Two green dots mark the cathode

   D4 (Green LED) Two green dots mark the cathode

3.5 Mounting the FPGA IC

As the very last component, the FPGA IC must be placed on the PCB. For this task, it is best to use the semiautomatic vacum twezer. Be **very** careful to place the IC as accurately as possible, as the pin spacing is a mere 0.5mm. Pin 1 on the IC is marked by a small circular dent of 1mm in diameter. The corner with pin 1 should be placed as shown in figure 3

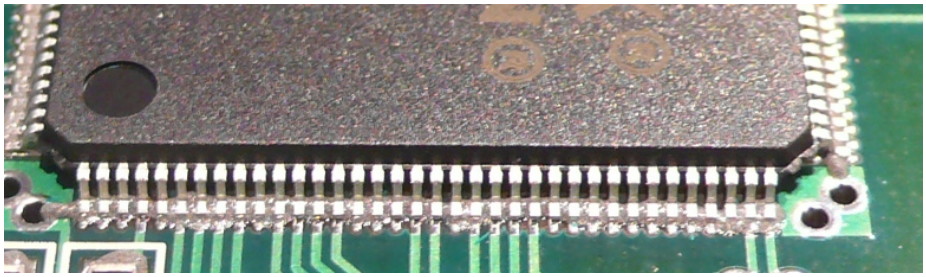


Figure 3: Placement of the FPGA

3.6 Vapor soldering the PCB

The PCB should be soldered with the top (FPGA) side facing up. The PCB should be placed on a frame in order to avoid bottom side components to touch anything. If the bottom side components touch the mesh of the solder oven, the components are prone to become displaced. A strip of metal can conveniently be used to for a frame, as shown in figure 4

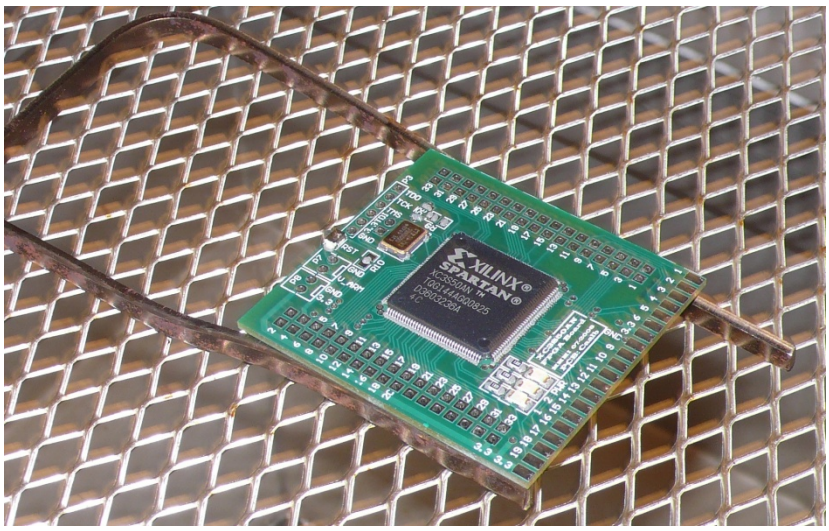


Figure 4: Placing the pcb in the vapor phase oven with a frame

3.7 Verification after soldering

When the PCB has been soldered, it is necessary to check the pads for tin bridges, which will short circuit the pads, potentially destroying the components. The FPGA IC is especially prone to the formation of tin bridges due to the very short distance between pins. It is most likely that you will encounter a few shorts between the FPGA pins, as shown in figure 5

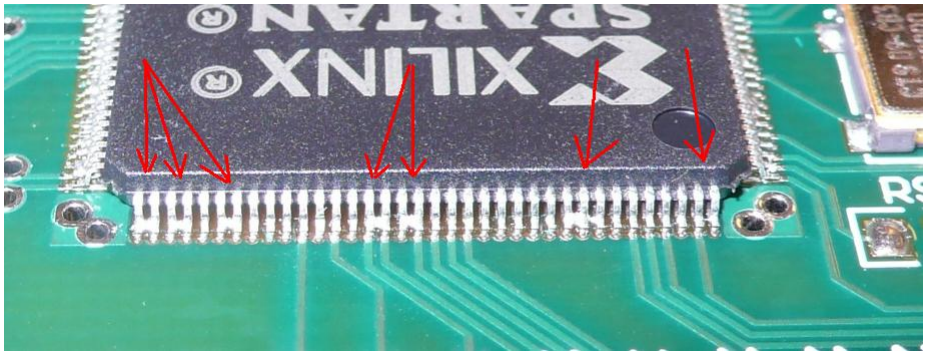


Figure 5: Shorts between fpga pins

Tin bridges are easily removed by using a solder iron and a bit of tin removal litze (3mm wide knitted band of copper wire with embedded fluss). As soon as the heat melts the tin, the litze will absorb any excess tin, like a sponge absorbs water.

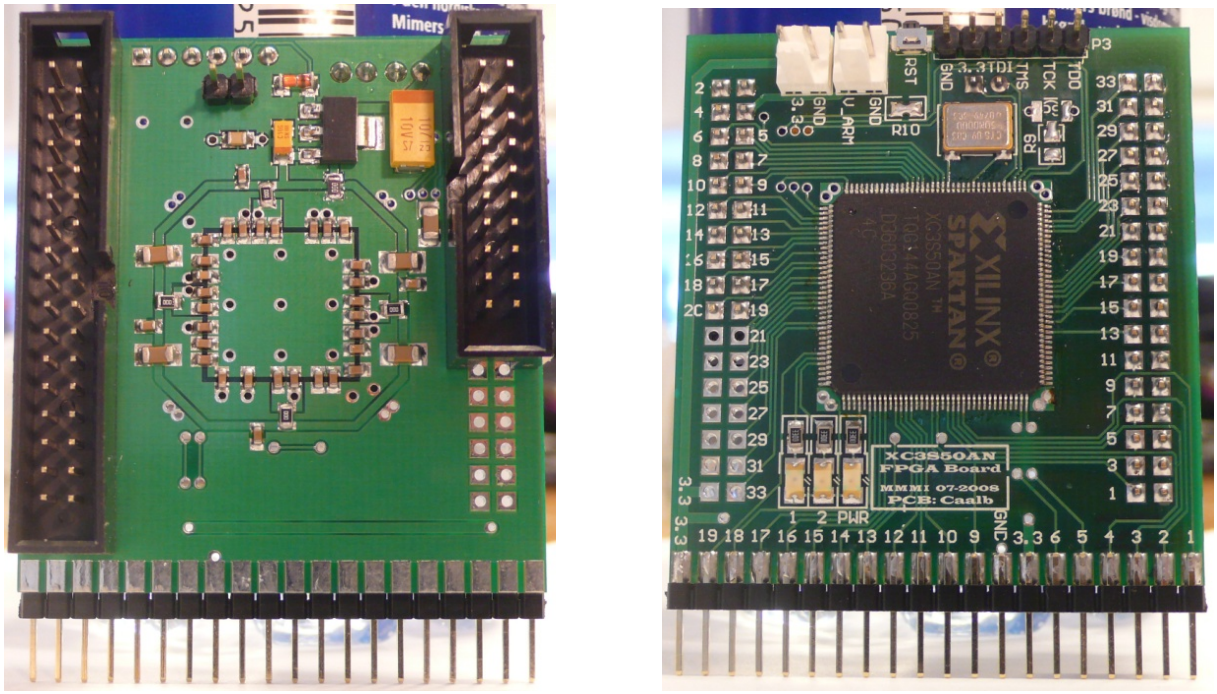
3.8 Mounting the connectors

The connecotrs: *P4, P5* and *P6* are mounted from the bottom side, as shown in figure 6(a). *P4* can either be a 2×10 header connector, or a 2×17 header socket identical to *P5* It is usually best to choose a 2×10 header, as this will enable later connection of an Arm-7 Olimex development board in this socket. In this case, the 2×10 header should be mounted in the holes 1-20, as shown in figure 6(a)

The connectors *P1, P3, P7* and *P8* are mounted on the top side, as shown in the figure 6(b) Note that *P1* is a single row pin header, mounted parallel with the surface of the PCB, as an edge-connector.

The easiest way to solder *P1* is to apply a bit of solder to one of the pads near the corners, and soldering a single pin of the header to this pad, so the rest of the pins allign with the pads. thereafter, it is eas y to apply solder to the rest of the pins.

If *P1* is to be used with high speed signals, it is recommended to mount an identical pinheader on the bottom side, parallel with *P1* to provide HF ground connections to the system carrying the FPGA PCB. Contact one of the authors for further information.



(a) Bottom side

(b) Top side

Figure 6: The finished PCB

3.9 testing the PCB

Before the PCB is cleared for use, it should be tested in the following way:

1. Use an ohmmeter to test for short circuits between the following nets:
   - GND ↔ 3.3V
   - GND ↔ 1.2V
   - 3.3V ↔ 1.2V

   If there is a short circuit, it is adamant to find the course, remove it, and test again. In most cases such shorts will occur due to tin bridges between FPGA pins (see above).
2. Use a 3.3V voltage supply with a current limiter set to approximately 100mA for this test.
   - Make sure that the voltage is 3.3V, using a voltmeter.
   - make sure the current limit is set to approximately 100mA, by short circuiting the output of the voltage supply, and observing the current consumption.
   - Connect the supply to the FPGA boards GND and 3.3V pins.

- If the current limitter becomes active, something is wrong with the FPGA board.
  - Check the polarity and voltage of the supply.
  - Check for wrongly polarized components.
  - Contact one of the authors, or other knowlegeable person.
- If the PCB made it through the above tests, the next step is to see if you can connect to the FPGA using a JTAG interface. If this is the case, you can then configure the FPGA in order to perform further tests, using the LEDS and/or oscillator.

4 Using the S3XB

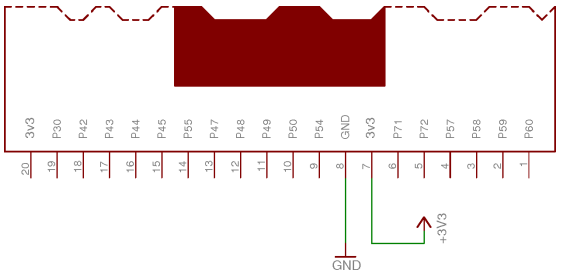
One of the main benefits of the S3XB, is that you can quickly try out simple interface circuits, by placing the S3XB in a breadboard. Obvioulsy, breadboards are not well suited for high speed designs, but for simple experiments with low bandwidth signals — say, below 5MHz — it can be very instructive to play around with a breadboard, for learning, or for quick verification of design ideas. If a more rugged design is necessary, the S3XB can also be used with a perforated prototyping PCB (perfboard).

This section describes how to get started with integrating the S3XB in your own experimental designs, using a breadboard.

4.1 Connecting power

Connecting power to the S3XB is very simple, as all voltage regulation and decoupling is taken care of on the S3XB board. Power can be supplied to the S3XB on different connectors, but when using it with a breadboard, we are using the bottom connector (P1), and you just need to adhere to the following:

- The S3XB **must be supplied with 3.3V!** A higher voltage will damage the circuitry of the S3XB.
- $0V$ should be connected to pin 8 of the bottom connector (P1)
- $+3.3V$ should be connected to either pin 7 or pin 20 of the bottom connector (P1). Pin 7 and 20 are internally connected on the board, so there is no difference in using them.
- There is no need to place ceramic bypass capacitors externally, as there are sufficient bypass capacitors for the FPGA, mounted directly on the FPGA.
- If the FPGA board is driving high currents, e.g. LED's or other low impedance loads, it may be necessary to stabilize the supply voltage, by a large capacitor, e.g. a $100\mu F$ electrolytic capacitor.



(a) Schematic of power connection

--- No picture yet ---

(b) Picture of setup on breadboard

Figure 7: Connecting power to the S3XB

4.2 Programming the FPGA

The S3XB is programmed through the 6-pin JTAG connector, placed on the top edge of the board. Refer to the diagrams below, for the detailed connections.

In order to transfer a `.bit` file from your PC to the board, you need a JTAG interface, connecting to your computers USB or parallel port. Make sure the JTAG interface is compatible with the development software you use. Xilinx USB JTAG interfaces are available for loan, at Odense Technical Library, or can be bought from Xilinx, or Xilinx partners.

We refer to the documentation of the development software — e.g. Xilinx ISE — and relevant litterature on FPGA programming, for further information on programming tools.

In section 5, we have shown a number of examples, demonstrating simple programing concepts, suitbale for beginners in FPGA programming.

4.3 Connecting to an atMEGA-8 microcontroller

The S3XB have been designed for easy interfacing with an atMEGA-8N microcontroller, as sown in figure 8. The connections from pin 1 to 14 can be accomplished without wires, in a breadboard, enabling very easy experiments with mixed CPU/FPGA applications.

The ATmega can be programmed using a wide array of free software tools, and we refer to relevant documentation about the ATmega-8 microcontroller, for further information on the subject. We recommend the site: `www.avrfreaks.org`

The connections from pin 1 to 14, will connect the full 8-bit port-D, and some bits from both port-C and port-B. So with just the 14 connections, a full 8-bit interface with handshaking can be esablished between the FPGA and the microcontroller.

By connecting the remaining 6 signals, all 8 bits of port-B will be connected, including the signals for the SPI interface, that will enable fast serial communication between the FPGA and

microcontroller.

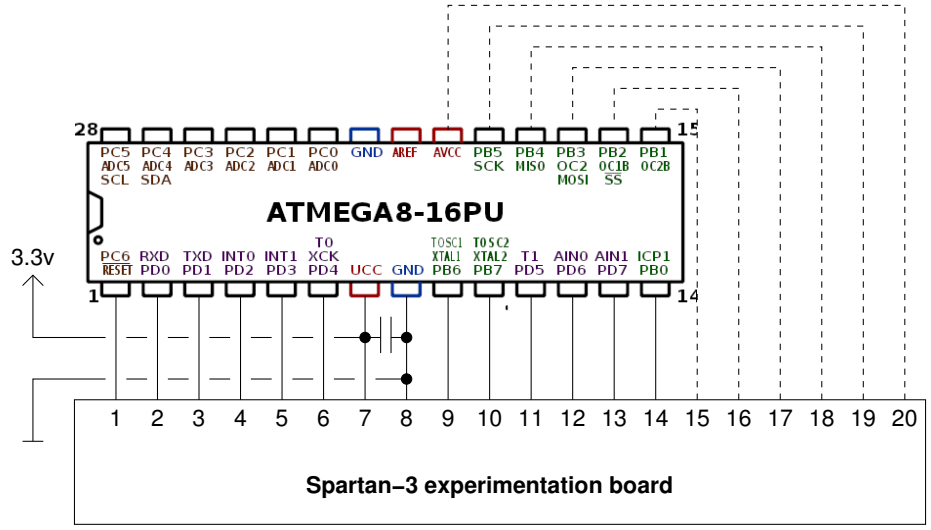


Figure 8: Easy connection between the S3XB and an ATmega-8N microcontroller

4.4 Connecting to the Olimex ARM-7 module

The S3XB has also been designe for easy interfacing with the more powerfull ARM-7 evaluation module from Olimex. The connectros on the S3XB has been laid out, so the Olimex board, can simply be mounted as a daughter-board, on the back of the S3XB. Refer to the Olimex Arm-7 borad documentation and the diagrams below, for details on the connections.

5 Programming examples

In this section, we show some simple examples of interfacing the FPGA to the outside world. We hope that you will get inspired to elaborate on our examples, and maybe combine them into new applications.

For each example, we give the relevant VHDL file(s) that specify the logical functionality, as well as the UCF file that specifies the pin mapping.

5.1 Simple blinker

The very first, and simplest application, is to use the onboard resources on the S3XB to verify that the FPGA works, and can be programmed. The following example, will use a binary counter to derive lower frequencies from the onboard 50MHz clock, and use two counter outputs to drive the two onboard LED's.

Listing 1: blinkled.vhd

```
--------------------------------------------------------------------------------
-- Platform:  SDU/TEK/Embedix Spartan-3 50AN experimentation board
----------------------------+----------+-----------------------------------------
-- History:     Date       | Author   | Action
-- Created:   2009_11_20   | Anss     | Created
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity Blinkled is
  Port (
-- The order of declarations are not important to the compiler, but as humans,
-- we like a certain order so we can read and understand the code easily:

-- First we declare signals to/from onboard components
     EXTCLK_I    : in  STD_LOGIC;    -- From the 50MHz clock generator
     LED1_O      : out STD_LOGIC;    -- To the yellow LED
     LED2_O      : out STD_LOGIC     -- To the red LED
-- Then we declare signals to the xxx board

-- ... but there are none (yet)
      );
end Blinkled;

architecture Behavioral of Blinkled is
  signal divider : STD_LOGIC_VECTOR (25 downto 0):= "00000000000000000000000000";
begin
  LED1_O <= divider(24);  -- connect bit 24 to LED1
  LED2_O <= divider(25);  -- connect bit 25 to LED2

  ClockDivide:
  process (EXTCLK_I) -- this process defines the counter, that counts the rising edges of extclk
  begin -- process
     if(EXTCLK_I'event and EXTCLK_I ='1') then
        divider <= divider + '1';
```

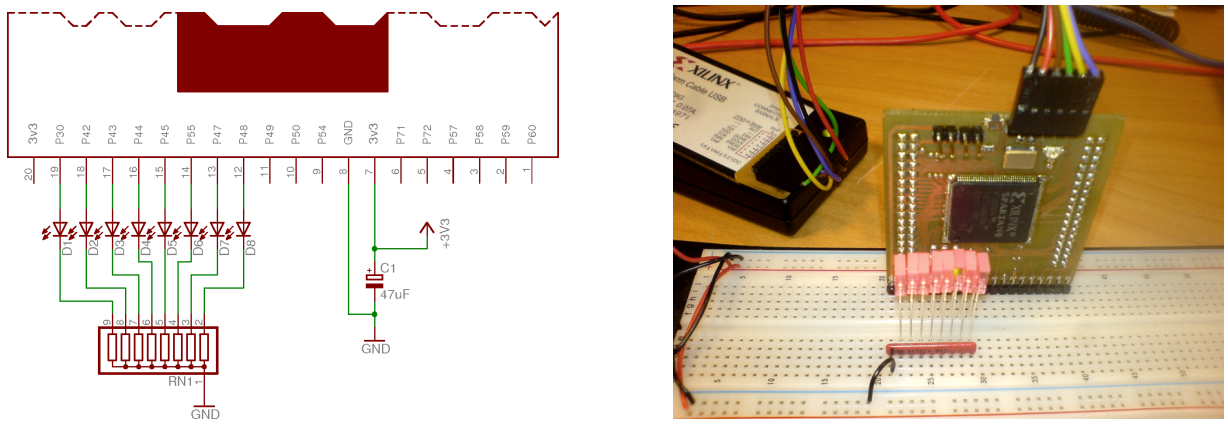
```
    end if;
  end process;

end Behavioral;
```

Listing 2: blinkled.ucf

```
# This is the .ucf file that defines constraints for the design.
# In the .ucf file, you can specify many diffent constraints,
# for instance which pins your signals should be routed to

# First we define pins for the onboard devices
NET "LED1_O"             LOC = P31;
NET "LED2_O"             LOC = P32;
NET "EXTCLK_I"  LOC = P124;
# There are no external connections in this design
```

5.2 8 bit output shift register

In this example, 8 LED's have been connected to pins on the FPGA. Refer to the S3XB diagram, the VHDL and the UCF file, to understand how the mapping from VHDL to LED is accomplished.



(a) Schematic example of 8-LED output

(b) 8-LED output on a breadboard

Figure 9: Connecting 8 LED's to the S3XB, on a breadboard

It is not difficult to specify a shift register in VHDL, but in order to be able to follow the action of the shift register visually, we need to reduce the clock frequency of the shift register from the 50MHz available on the S3XB. This could be done using a slower external clock — building your own osciallator on the breadboard, or by dividing the onboard 50MHz clock.

We have chosen to divide the onboard clock, and leave it as an exercise for the reader to change the design in order use a slower external oscillator.

Xilinx recommend a design practice, where the original clock is used to clock all sequential logic in a design, and speed reductions are achieved by a separate *clock enable* signal, which will then

only be active during a fraction of the clock cycles. This design practice will minimize clock skew, which is very important in the large, complex designs we will meet later on.

Note that the VHDL specifies two processes:

ShiftRegister: Is the shift register. It is sensitive to the `clk` signal, and reacts to rising edges of `clk`, provided that the `enable` signal is active. This means that we can slow down the rate of the shift register with a factor of N, by only having `enable` active at every N clock cycle.

ClockDivide: Is the counter that counts clock edges, uisng a 26 bit binary up counter.

This process also control the `enable` signal, which will be active, when all bits in the counter are low. This will occur during one clock cycle out of every 2^{26}, giving the shift register a rate of $50\text{MHz}/2^{26} \simeq 0.75\text{Hz}$.

The `flash` signal goes on at the same time as the `enable` signal, but stays on, even when the lowest 20 bits of conter difers from 0. This gives `flash` an on time of $2^{20}/50\text{Hz} \simeq 20\text{ms}$, making it ideal to drive a LED, that will emit a short, distinct flash, whenever the shift register chages state.

Listing 3: shiftreg.vhd

```
--------------------------------------------------------------------------------
-- Platform:     SDU/TEK/Embedix Spartan-3 50AN experimentation board
-- Application:  8-bit Shiftregister demo
----------------------------+----------+-----------------------------------------
-- History:     Date        | Author   | Action
-- Created:   2009_11_20    | Anss     | Created
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity Shiftreg is
    Port (
   EXTCLK_I        : in  STD_LOGIC;                    -- From the 50MHz clock generator
   LED1_O          : out STD_LOGIC;                    -- To the red LED
   LED2_O          : out STD_LOGIC;                    -- To the yellow LED

   DATA_O          : out STD_LOGIC_VECTOR(7 downto 0)  -- an 8 bit output vector
         );
end Shiftreg;

architecture Behavioral of Shiftreg is

   signal clk           : std_logic;            -- we use this signal for clk
   signal enable        : std_logic;            -- signal used as clock enable
   signal flash         : std_logic;            -- signal used to blink an LED when enable is activated

   signal divider       : STD_LOGIC_VECTOR (25 downto 0):= "00000000000000000000000000";
   signal shiftreg      : std_logic_vector(7 downto 0) := "00000001";

begin
   clk <= EXTCLK_I;        -- clk is the same as EXTCLK_I (The compiler will make them the same wire)
   LED1_O <= flash;        -- connect flash signal to LED1
   LED2_O <= shiftreg(0);  -- connect bit 0 of the shift register to  LED2
```

```
   DATA_O <= shiftreg;

-- ========================
-- The Shift Register will:
-- Shift the 8 bit register, in a loop
-- ========================
  ShiftRegister:
  process(clk)
  begin -- process
    if (clk'event and clk='1' and enable='1') then
      shiftreg<=shiftreg(0) & shiftreg(7 downto 1);
    end if;
  end process;

-- ==================================
-- The ClockDivide process will:
-- * Divide the clock using a 26 bit binary counter
-- * Provide an enable signal for a single clock period out of the 2^26
-- * Provide a  flash signal (for a LED) which goes on simultanious with enable,
--   and stays on, long enough for the LED to provide a visible flash
-- ==================================
  ClockDivide:
  process (clk) -- this process defines the counter, that counts the rising edges of extclk
  begin -- process

    if(clk'event and clk ='1') then       -- If there is a rising edge on clk
      if divider = 0 then                 --   if all the divider bits are 0
        enable <= '1';                    --     activate enable
      else                                --   At all other combinations
         enable <= '0';                   --     deactivate enable
       end if;

       if divider(25 downto 20) = 0 then  -- If the highest (slowest) bits are all 0
         flash <='1';                     --   activate the LED (the flash signal goes on simultanious with enabl
       else                               --   but stays on for a longer period, so it will be visible)
         flash <='0';                     -- Else, deactivate LED
       end if;

       divider <= divider + '1';          -- Increment the divider counter
    end if;
  end process;

end Behavioral;
```

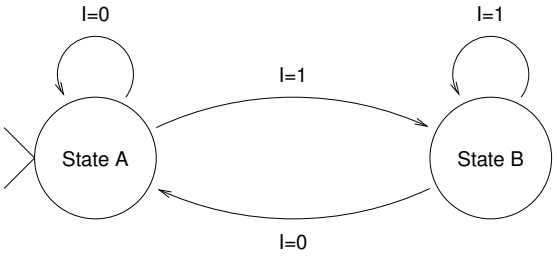
Listing 4: shiftreg.ucf

```
# This is the .ucf file that defines constraints for the design.
# In the .ucf file, you can specify many diffent constraints,
# for instance which pins your signals should be routed to

# First we define pins for the onboard devices
NET "LED1_O"            LOC = P31;
NET "LED2_O"            LOC = P32;
NET "EXTCLK_I"          LOC = P124;

# Then we define pins for the bottom connector (P1)

NET "DATA_O<0>" LOC = P48;  # Bottom connector pin 12
NET "DATA_O<1>" LOC = P47;  # Bottom connector pin 13
NET "DATA_O<2>" LOC = P55;  # Bottom connector pin 14
NET "DATA_O<3>" LOC = P45;  # Bottom connector pin 15
NET "DATA_O<4>" LOC = P44;  # Bottom connector pin 16
NET "DATA_O<5>" LOC = P43;  # Bottom connector pin 17
NET "DATA_O<6>" LOC = P42;  # Bottom connector pin 18
NET "DATA_O<7>" LOC = P30;  # Bottom connector pin 19
```

5.3 Edge detector and counter

Even though it is easy to specify a sequential functionality, that will react to the edges of other external signals than the clock, it is usually desirable to syncronize external signals to the system clock, in order to simplify the overall design, and avoid setup/hold timing problems. Also, we would often like to be able to react to both rising and falling edges of an external signal.

Provided that the global clock is sufficiently faster that the rate of change of the signal we want to monitor, both goals can be accomplished by a the simple state machine shown in figure 10



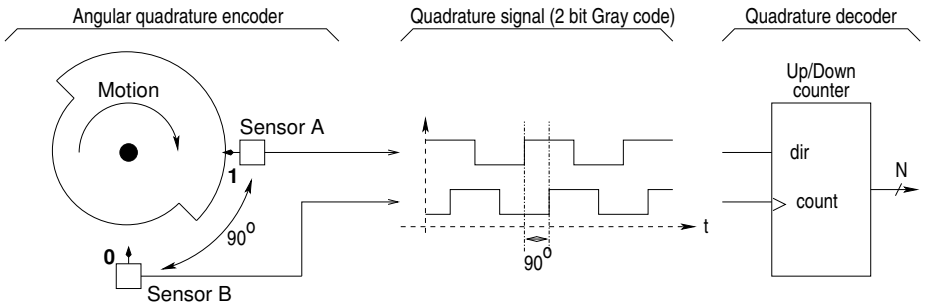
$$\Sigma = \{0, 1\}$$
$$S = \{A, B\}$$
$$s_0 = A$$
$$\delta = \ldots$$

Figure 10: Edge detector DFA

The state machine can be used in several different ways, depending on the application. If actions are associated with the state change, we can implement machines, that react to rising and/or falling edges of input signals.

In the Up/Down counter example below, we show the principle, by merging the simple edge detect state machine, with an 8 bit counter. The counter will change it's number up or down, whenever the input called `count` experiences a rising edge. If the input called `dir` is high, the counter increments, if `dir` is low, the counter decrements. If the counter goes down from 0, it wraps to 255. If it goes up from 255, it wraps to 0. The example uses the same connection of 8 LED's, as the shift register above, to show the state of the counter.

An up down counter like this, is well suited as a primitive quadrature decoder, for e.g. incremental position/angle encoders, as shown in figure 11. Note however, that it is suceptible to prell (contact noise), in this simple form. A prell immune counter can be created by counting on both up and down going flanks, and the resolution can be increased by counting on both flanks on both signals.



Note: Practical encoders usually have more pulses per revolution, typically 16-512

Figure 11: Sketch of quadrature encoding/decoding used in motion sensing

Listing 5: updown.vhd

```
--------------------------------------------------------------------------------
-- Platform:     SDU/TEK/Embedix Spartan-3 50AN experimentation board
-- Application:  8-bit up/down counter using state machine edge detect
----------------------------+----------+-------------------------------------
-- History:     Date        | Author   | Action
-- Created:   2009_11_20    | Anss     | Created
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity UpDown is
    Port (
   EXTCLK_I        : in  STD_LOGIC;                    -- From the 50MHz clock generator
   LED1_O          : out STD_LOGIC;                    -- To the red LED
   LED2_O          : out STD_LOGIC;                    -- To the yellow LED

   COUNT_I         : in STD_LOGIC;
        DIR_I      : in STD_LOGIC;
   DATA_O          : out STD_LOGIC_VECTOR(7 downto 0)  -- an 8 bit output vector
         );
end UpDown;

architecture Behavioral of UpDown is

   signal clk_50M   : std_logic;               -- we use this signal for clk
   signal count_b   : std_logic;               -- syncronized version of count signal
   signal dir_b     : std_logic;               -- syncronized version of dir signal

   signal counter   : std_logic_vector(7 downto 0) := "01111111";  -- start at 0x7f

   type state_type is (S0,S1);            -- Define an enumerated state type, for state machine
   signal state      : state_type := S0;  -- A signal to hold the state

begin
   clk_50M <= EXTCLK_I;                        -- clk_50M is the same as EXTCLK_I (The compiler will make them th
   LED1_O <= COUNT_I;
   LED2_O <= DIR_I;
   DATA_O <= counter;

-- ============================================================================
-- The Syncronizer will syncronize all external inputs to the system clock
-- ============================================================================
```

```
  Syncronizer:
  process(clk_50M)
  begin -- process
    if (clk_50M'event and clk_50M='1') then
      count_b <= COUNT_I;
      dir_b <= DIR_I;
    end if;
  end process;

-- ============================================================
-- The UDCounter process will handle the counter state machine in one process
-- ============================================================
  UDCounter:
  process(clk_50M)
    variable next_state   : state_type;
    variable next_counter : std_logic_vector(7 downto 0);
  begin -- process
    if(clk_50M'event and clk_50M='1') then  -- on rising edges of the clk
      next_state := state;                  -- default, we stay in same state
      next_counter := counter;              -- default, counter stays
      case state is
        when S0 =>                          -- Being in S0 indicates that count was previously '0'
          if count_b ='1' then              -- If count is now '1' there has been a rising edge
            next_state := S1;               --   So next state should be S1
            if dir_b = '1' then             --   If the direction is '1' (encoding up)
              next_counter := counter+1;    --     Increment counter
            else                            --   If the direction is not '1'
              next_counter := counter-1;    --     Decrement the counter
            end if;
          end if;
        when S1 =>                          -- Being in S1 indicates that count was previously '1'
          if count_b = '0' then             -- If count is now '0' there has been a falling edge
            next_state := S0;               --   So next state should be S0
          end if;                           -- We don't evaluate the counter on falling edges (but we could)
        when others =>
      end case;
      state <= next_state;                  -- Latch the next state
      counter <= next_counter;              -- Latch the next counter value
    end if;
  end process;

end Behavioral;
```

Listing 6: updown.ucf

```
# This is the .ucf file that defines constraints for the design.
# In the .ucf file, you can specify many diffent constraints,
# for instance which pins your signals should be routed to

# First we define pins for the onboard devices
NET "LED1_O"            LOC = P31;
NET "LED2_O"            LOC = P32;
NET "EXTCLK_I"          LOC = P124;

# Then we define pins for the bottom connector (P1)

NET "DATA_O<0>" LOC = P48;  # Bottom connector pin 12
NET "DATA_O<1>" LOC = P47;  # Bottom connector pin 13
NET "DATA_O<2>" LOC = P55;  # Bottom connector pin 14
NET "DATA_O<3>" LOC = P45;  # Bottom connector pin 15
NET "DATA_O<4>" LOC = P44;  # Bottom connector pin 16
NET "DATA_O<5>" LOC = P43;  # Bottom connector pin 17
NET "DATA_O<6>" LOC = P42;  # Bottom connector pin 18
NET "DATA_O<7>" LOC = P30;  # Bottom connector pin 19
```

```
NET "COUNT_I"     LOC = P49;  # Bottom connector pin 11
NET "DIR_I"       LOC = P50;  # Bottom connector pin 10
```

5.4 Serial output

In many instances, it is practical to be able to output multiple bits, in a serial data stream. This can be used in communication applications, or simply as a debugging option.

There are many ways to specify a machine, that will output a serial bitstream, but we have chosen to show a simple, straight-forward state machine, to implement an asyncronous serial output, with 8 data bits, 1 start bit, and 2 stop bits. The bit rate is set to a fraction of the clock rate, using the recommended clock enable scheme, as seen in the shift register example above.

You may note, that the use of enumerated states will make the source code a little bit clumsy, compared to using e.g. binary encoding, which will enable a much more compact coding style for the next state and output logic. The pros and cons of coding styles can be debated forever, but the enumerated way of desribing states is often prefered, as it allows the compiler to choose the state encoding scheme, which is most effective from an overall perspective.

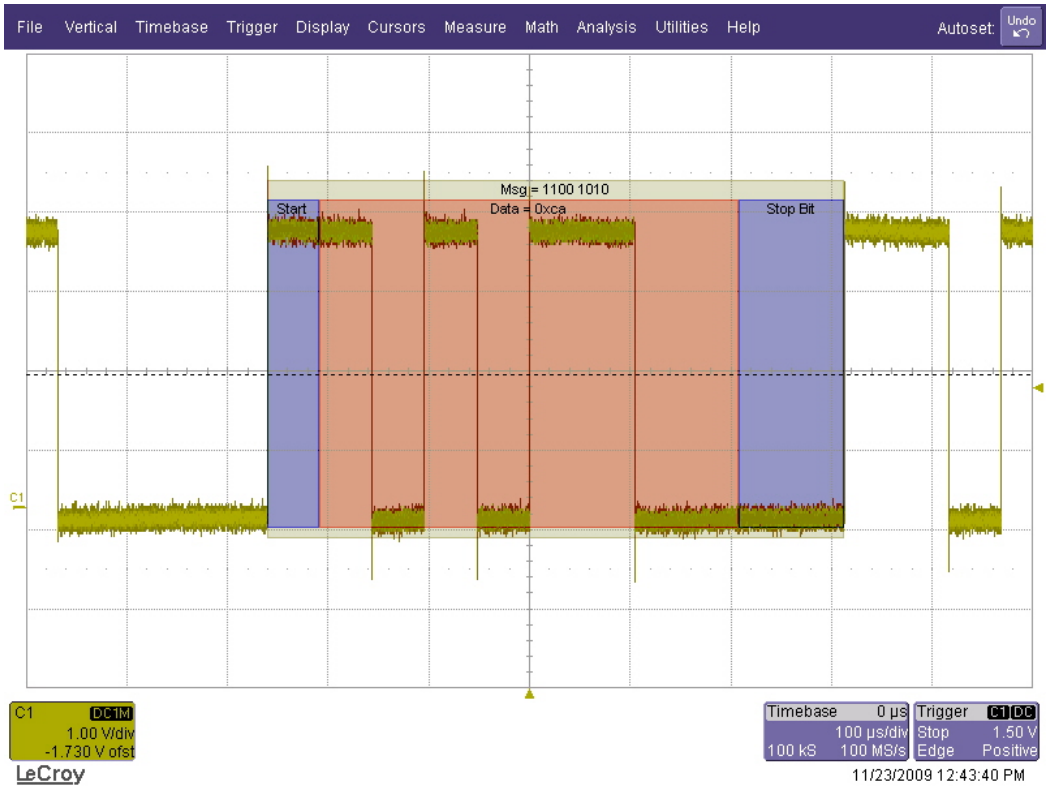


Figure 12: Screenshot of serial output data

The output waveform of the code example is shown in figure 12, with annotations from a serial-decode filter. As you can see, it might be difficult to distinguish the start bit from any other high bit in the signal, so it might be a good idea, to have a break between transmissions, to aid detection of the start bit. This can easily be acomplished by increasing the number of stop bits to e.g. 10.

Listing 7: serial_out.vhd

```
--------------------------------------------------------------------------------
-- Platform:     SDU/TEK/Embedix Spartan-3 50AN experimentation board
-- Application:  8-bit asyncronous serial out @19200bps
----------------------------+----------+--------------------------------------
-- History:     Date        | Author   | Action
-- Created:   2009_11_20    | Anss     | Created
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity SerialOut is
   Port (
     EXTCLK_I      : in  STD_LOGIC;  -- From the 50MHz clock generator
     XMIT_O        : out STD_LOGIC   -- Serial data output
   );
end SerialOut;

architecture Behavioral of SerialOut is

   signal clk_50M   : std_logic;                                  -- we use this signal for clk
   signal xmit      : std_logic;                                  -- xmit signal
   signal data      : std_logic_vector(7 downto 0) := "11001010"; -- The data to output

-- type and signal declarations for xmit state machine
   type so_state_type is (START,B0,B1,B2,B3,B4,B5,B6,B7,S1,S2);
   signal so_state         : so_state_type := START;
   signal nxt_so_state     : so_state_type;
   signal so_clk_enable    : std_logic;
   constant SO_BAUD_PERIOD : integer := 2604;  -- A period of 2604 clocks at 50MHz gives apprx 19200 Hz
   signal so_baud_counter  : integer range 0 to SO_BAUD_PERIOD;


begin
   clk_50M <= EXTCLK_I;                       -- clk_50M is the same as EXTCLK_I (The compiler will make them th
   XMIT_O <= xmit;

-- ================================================================================
-- The SoState process handles the state register of the DFA controlling Serial Output
-- ================================================================================
  SoState:
  process(clk_50M)
  begin -- process
    if (clk_50M'event and clk_50M='1' and so_clk_enable = '1') then
      so_state <= nxt_so_state;
    end if;
  end process;

-- ============================================================================
-- The (unclocked) SoNxt process implements the combinatorial next state logic
-- ============================================================================
  process(so_state)
  begin -- process;
```

```
    case so_state is
      when START => nxt_so_state <= B0;
      when B0    => nxt_so_state <= B1;
      when B1    => nxt_so_state <= B2;
      when B2    => nxt_so_state <= B3;
      when B3    => nxt_so_state <= B4;
      when B4    => nxt_so_state <= B5;
      when B5    => nxt_so_state <= B6;
      when B6    => nxt_so_state <= B7;
      when B7    => nxt_so_state <= S1;
      when S1    => nxt_so_state <= S2;
      when S2    => nxt_so_state <= START;
    end case;
  end process;

-- ======================================================================
-- The (unclocked) SoOut process implements the combinatorial output logic
-- Decode as std UART : LSB first, IDLE = '1' START-BIT = '0', DATA inverted
-- ======================================================================
  SoOut:
  process(so_state)
  begin -- process;
    case so_state is
      when START   => xmit <= '1';
      when B0      => xmit <= not data(0);
      when B1      => xmit <= not data(1);
      when B2      => xmit <= not data(2);
      when B3      => xmit <= not data(3);
      when B4      => xmit <= not data(4);
      when B5      => xmit <= not data(5);
      when B6      => xmit <= not data(6);
      when B7      => xmit <= not data(7);
      when others  => xmit <= '0';
    end case;
  end process;

-- ====================================
-- The SoBaud process implements a clock enable signal for the SoState
-- ====================================
  SoBaud:
  process(clk_50M)
  begin -- process
    if (clk_50M'event and clk_50M = '1') then
      if so_baud_counter < SO_BAUD_PERIOD then
        so_baud_counter <= so_baud_counter+1;
        so_clk_enable <= '0';
      else
        so_baud_counter <= 0;
        so_clk_enable<='1';
      end if;
    end if;
  end process;

end Behavioral;
```

Listing 8: serial_out.ucf

```
# This is the .ucf file that defines constraints for the design.
# In the .ucf file, you can specify many diffent constraints,
# for instance which pins your signals should be routed to

# First we define pins for the onboard devices
NET "EXTCLK_I"          LOC = P124;

# Then we define pins for the bottom connector (P1)
```

```
NET "XMIT_O"      LOC = P60;  # Bottom connector pin 1
```

5.5 Analog input

As FPGA's does not normally have analog inputs, analog data has to be converted into digital form, to be processed by the FPGA. This can be accomplished directly, by chhosing and connecting an external A/D converter to the FPGA, using either a parallel or serial bus to transfer the data.

If the demands for quality is not very high, it is however possible to read analog data using some very simple additional circuitry. The simplest possible example, is to read the value of a variable resistor, by measuring the time it takes to charge a capacitor through the resistor.

Implementing such a scheme requires a square generator, with an output that will be low long enough, to ensure the capacitor is virtually decharged to begin with. Then the output must be high long enough, to ensure the capacitor is charged using the max value of the resistor. A timer will then be able to determine how long it actually takes to charge the capacitor, and thus what the resistance is.

The example shown below, will work with a capacitance of $C \simeq 200nF$ against a resistance $R \in [0\Omega; 100k\Omega]$ The resistance can be a potentiometer, a NTC or PTC thermistor, an LDR, a force sensitive resistor, or any other resistive sensor element.

A similar converter can be made for voltage inputs, by having a fixed resistor, and the using an external analog comparator to signal the FPGA when the capacitor voltage exeeds the input voltage.

Note that such converters are quite unlinear, due to the unlinear nature of the RC charge voltage. If a more linear converter is desired, the RC filter can be replaced by an integrator, using an operational amplifier or similar amplifying element.

Listing 9: analog_in.vhd

```
--------------------------------------------------------------------------------
-- Platform:     SDU/TEK/Embedix Spartan-3 50AN experimentation board
-- Application:  Analog input from Resistive (0-100kR) sensor, using 200nf C
----------------------------+----------+-------------------------------------
-- History:     Date        | Author   | Action
-- Created:   2009_11_20    | Anss     | Created
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity AnalogIn is
    Port (
   EXTCLK_I        : in  STD_LOGIC;                    -- From the 50MHz clock generator
   PULSE_O         : out std_logic;
```

```
        PULSE_I                 : in  std_logic;

   DATA_O          : out STD_LOGIC_VECTOR(7 downto 0)  -- an 8 bit output vector
        );
end AnalogIn;

architecture Behavioral of AnalogIn is

   signal clk_50M              : std_logic;            -- we use this signal for clk
   signal counter       : std_logic_vector(20 downto 0) := (others => '0');
        signal testpuls      : std_logic;
        signal returnpuls    : std_logic;
        signal trig          : std_logic;
   signal data        : std_logic_vector(7 downto 0);

begin
   clk_50M <= EXTCLK_I;        -- clk is the same as EXTCLK_I (The compiler will make them the same wire)
   DATA_O <= data;
        PULSE_O <= testpuls;
        returnpuls <= PULSE_I;

-- ======================================================================
-- The PulsGen process will generate the test pulse with 25% duty cycle
-- ======================================================================
  process(clk_50M)
  begin --process
    if (clk_50M'event and clk_50M='1') then
           if counter(20 downto 19) = "00" then   -- The first 1/4 of the cycle
                  testpuls <= '1';                      -- Set the output high
                  if returnpuls='1' and trig='0' then  -- The first time the input goes high
                    data<=counter(18 downto 11);       --   latch the 8 bits that are active during the test puls
                         trig<='1';                          --   and remember that we have latched - so we ignore
                  end if;
                else                                    -- The remaining 3/4 of the cycle
                  testpuls <= '0';                      --   Keep the output low (decharge the capacitor)
                  trig<='0';                            --   Reset the trig signal
                end if;
          counter <= counter+1;                 -- The counter keeps running
         end if;
  end process;

end Behavioral;
```

Listing 10: analog_in.ucf

```
# This is the .ucf file that defines constraints for the design.
# In the .ucf file, you can specify many diffent constraints,
# for instance which pins your signals should be routed to

# First we define pins for the onboard devices

NET "EXTCLK_I"          LOC = P124;

# Then we define pins for the bottom connector (P1)

NET "DATA_O<0>" LOC = P48;  # Bottom connector pin 12
NET "DATA_O<1>" LOC = P47;  # Bottom connector pin 13
NET "DATA_O<2>" LOC = P55;  # Bottom connector pin 14
NET "DATA_O<3>" LOC = P45;  # Bottom connector pin 15
NET "DATA_O<4>" LOC = P44;  # Bottom connector pin 16
NET "DATA_O<5>" LOC = P43;  # Bottom connector pin 17
NET "DATA_O<6>" LOC = P42;  # Bottom connector pin 18
NET "DATA_O<7>" LOC = P30;  # Bottom connector pin 19

NET "PULSE_O"      LOC = P60;
```

```
NET "PULSE_I"      LOC = P59;
```

5.6 Analog output (with PWM)

Even though FPGA's normally doesn't have D/A convertes, it is possible to create analog output of rather high quality with simple means. There are various schemes to accomplish this, but most are accomplished by low-pass filtering a single digital output, which is modulated in some way.

The most common modulation scheme is the pulse width modulation — PWM —, which is a square wave with fixed frequency, but variable duty cycle. Such a modulation is very easy to implement with a simple counter, a data register and a comperator.

PWM is very popular in applications where a relatively low modulation frequency is desired, such as in power electronics, where the PWM signal drives switching transistors, that deliver power directly to a load. As the transistors have their maximum heat loss during switching, a relatively low switching frequency is often desired.

For non-power applications, where signal bandwidth and aliasing noise is more important, sigma/delta modulation is often a better option than PWM, because the switching frequency can become much higher, and thus easier to remove from the signal with a simple filter.

The example below show a simple PWM modulation with 8 bit encoding, giving 256 distinct duty cycles. The switch frequency is reduced to 25kHz, using an 11 bit counter, but only the top 8 bits are compared to the data register.

The PWM generator can be used to drive switching (power) transistors directly, in order to control e.g. motors. The intertia and self inductance of a motor will even out the switch frequency.

The pwm generator is also well suited to control the apparant intensity of LED's, (and other lamps) as any LED blinking faster than $\simeq 25Hz$ will fool the eye to appear steady.

If a true analog signal is to be created from the PWM waveform, just use a simple RC low-pass filter, with a corner frequency significantly lower than the switching frequency. It is left as an exercise to the reader, to figure out the relationship between filter characteristics and signal to noise ratio.

Listing 11: pwm.vhd

```
--------------------------------------------------------------------------------
-- Platform:     SDU/TEK/Embedix Spartan-3 50AN experimentation board
-- Application:  PWM demo: 8 bit resolution, 25kHz switch frequency
----------------------------+----------+---------------------------------------
-- History:     Date        | Author   | Action
-- Created:   2009_11_20    | Anss     | Created
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity PWM is
    Port (
   EXTCLK_I        : in  STD_LOGIC; -- From the 50MHz clock generator
   PWM_O           : out std_logic
         );
end PWM;

architecture Behavioral of PWM is

   signal clk_50M       : std_logic;    -- we use this signal for clk
   signal counter       : std_logic_vector(10 downto 0) := (others => '0');
   signal pwm           : std_logic;
   signal data          : std_logic_vector(7 downto 0) :="01111111";

begin
   clk_50M <= EXTCLK_I; -- clk is the same as EXTCLK_I (The compiler will make them the same wire)
   PWM_O <= pwm;

-- ================================================================================
-- The PulsGen process will generate the test pulse with 25% duty cycle
-- ================================================================================
  process(clk_50M)
  begin --process
    if (clk_50M'event and clk_50M='1') then
           if counter(10 downto 3) < data then     -- from 0 to the value of data, using only the
                                                   -- top 8 bits of the counter, to reduce frequency
                 pwm<='1';                         --   the pwm output is high
               else                                -- the rest of the cycle
                 pwm<='0';                         --   the pwm output is low
               end if;
               counter<=counter+1;                 -- The counter just runs and runs and runs
        end if;
  end process;

end Behavioral;
```

Listing 12: pwm.ucf

```
# This is the .ucf file that defines constraints for the design.
# In the .ucf file, you can specify many diffent constraints,
# for instance which pins your signals should be routed to

# First we define pins for the onboard devices

NET "EXTCLK_I"          LOC = P124;

# Then we define pins for the bottom connector (P1)

NET "PWM_O"        LOC = P58;   # bottom connector pin 3
```

6 Pitfalls and common problems

A Pin mapping

A.1 bottom connector

The bottom connector is intended for use with either breadboard prototyping systems or prototyping PCB's. In case of a breadboard, this connector needs to be a single line of pins, so we have used a 1×20 pin-header with a spacing of 100mils (2.54mm), designated *P1* in figure **??**. If a prototyping PCB is used instead of a breadboard. the HF properties of the connection can be greatly improved by adding the identical connector *P2*, which is mounted parallel to *P1*, and which carries a connection to the S3XB ground plan on every pin. The two connecotrs will appear as a single 2×20 pinheader, allowing easy connection to a prototyping PCB, or a ribbon cable.

P1	FPGA	function
1	p60	
2	p59	
3	p58	
4	p57	
5	p72	
6	p71	
7	3.3V	Power
8	GND	Reference, Power
9	p54	
10	p50	
11	p49	
12	p48	
13	p47	
14	p55	
15	p45	
16	p44	
17	p43	
18	p42	
19	p30	
20	3.3V	Power

P2	FPGA	function
1	GND	Reference, Power
2	GND	Reference, Power
3	GND	Reference, Power
4	GND	Reference, Power
5	GND	Reference, Power
6	GND	Reference, Power
7	GND	Reference, Power
8	GND	Reference, Power
9	GND	Reference, Power
10	GND	Reference, Power
11	GND	Reference, Power
12	GND	Reference, Power
13	GND	Reference, Power
14	GND	Reference, Power
15	GND	Reference, Power
16	GND	Reference, Power
17	GND	Reference, Power
18	GND	Reference, Power
19	GND	Reference, Power
20	GND	Reference, Power

A.2 Right connector

The right connector is a 2×17 pin header with 100 mil (2.54mm) spacing. It is mechanically compatible to prototyping PCB's and 34-way ribbon cables[2]

P5	FPGA	Function	P5	FPGA	Function
1	p54		2	GND	Reference, Power
3	p55		4	GND	Reference, Power
5	p57		6	GND	Reference, Power
7	p58		8	GND	Reference, Power
9	p59		10	GND	Reference, Power
11	p60		12	GND	Reference, Power
13	p83		14	GND	Reference, Power
15	p85		16	GND	Reference, Power
17	p87		18	GND	Reference, Power
19	p88		20	GND	Reference, Power
21	p90		22	GND	Reference, Power
23	p91		24	GND	Reference, Power
25	p92		26	GND	Reference, Power
27	p93		28	GND	Reference, Power
29	p103		30	GND	Reference, Power
31	p104		32	GND	Reference, Power
33	p105		34	GND	Reference, Power

Table 1: Connections for the Right I/O connector (P5)

A.3 Left connector

The left connector is a 2×17 pin header with 100 mil (2.54mm) spacing. It is mechanically compatible to prototyping PCB's and 34-way ribbon cables.

Note that the pin $1 \ldots 20$ have been configured to be compatible with the connector on an Olimex ARM-7 board, so the ARM-7 board can be connected directly to the first 20 pins of the left connector. To achieve this, it was necessary to avoid connecting pin 3 & 4 to the FPGA, as they are related to the power supply for the ARM board.

[2]Tip: 34 pin ribbon cables were previously used for 3.5 inch floppy disc drives, so your IT administrator may have some lying around

P4	FPGA	Function	P4	FPGA	Function
1	GND	Reference, Power	2	p125	
3		V_{ARM}	4		
5	p127		6	p126	
7	p130		8	p129	
9	p132		10	p131	
11	p4		12	p3	
13	p6		14	p5	
15	p8		16	p7	
17	p12		18	p10	
19	p15		20	p13	
21	p16		22	GND	Reference, Power
23	p18		24	GND	Reference, Power
25	p19		26	GND	Reference, Power
27	p20		28	GND	Reference, Power
29	p21		30	GND	Reference, Power
31	p24		32	GND	Reference, Power
33	p25		34	GND	Reference, Power

Table 2: Connections for the Left I/O connector (P4)

A.4 Olimex connector

In order to provide mechanical stability when mounting the Olimex ARM-7 board on the S3XB, two pins have been added in the middle of the S3XB board. The two pins fit the other connector on the ARM-7 board, inhibiting it from rocking when mounted. The two pins provide two extra connections between FPGA and ARM, with the following configuration:

P6	FPGA	Function
1	p113	
2	p111	

B Schematic

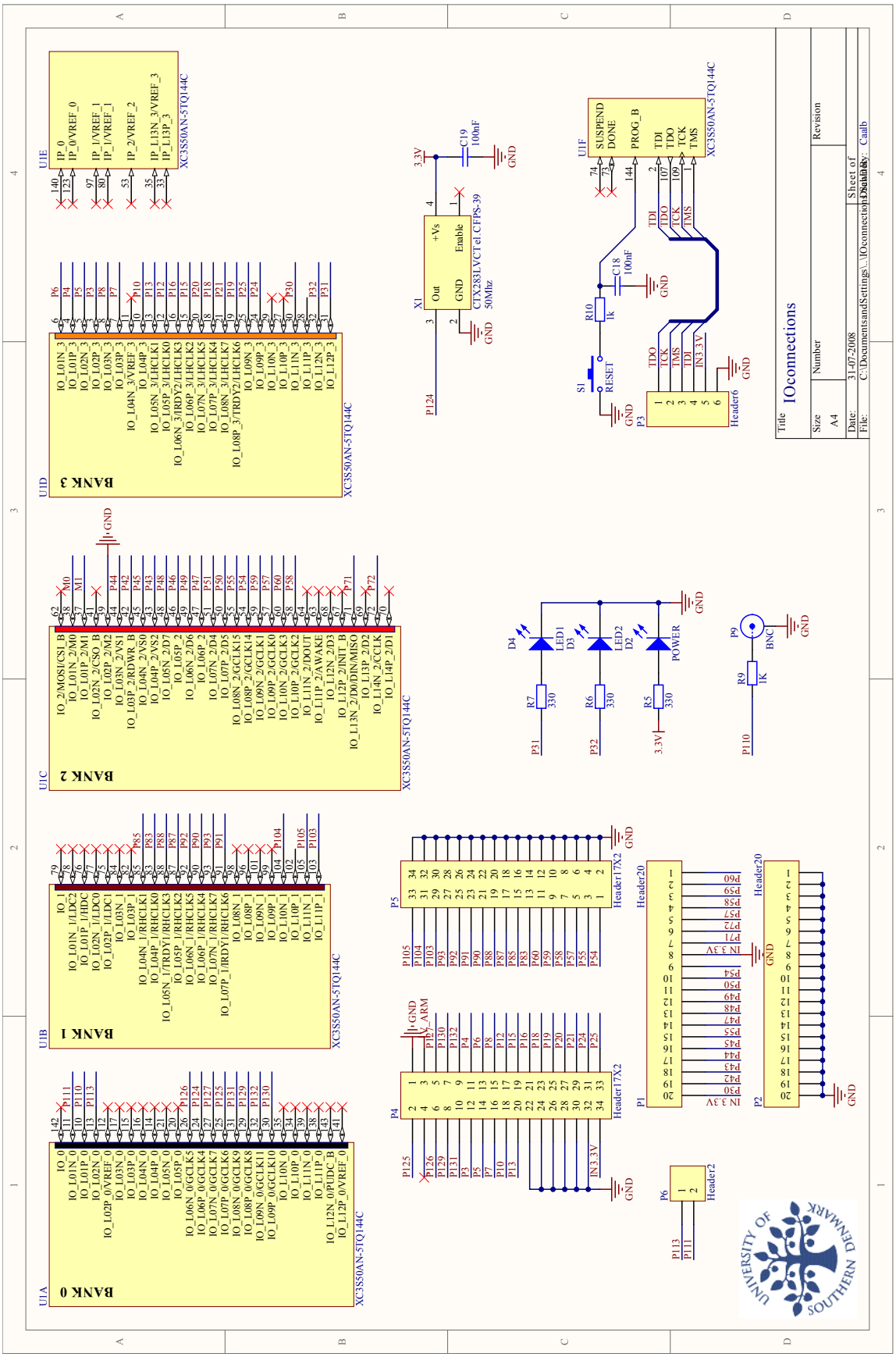


Figure 13: Schematic - page 1 (I/O connections)

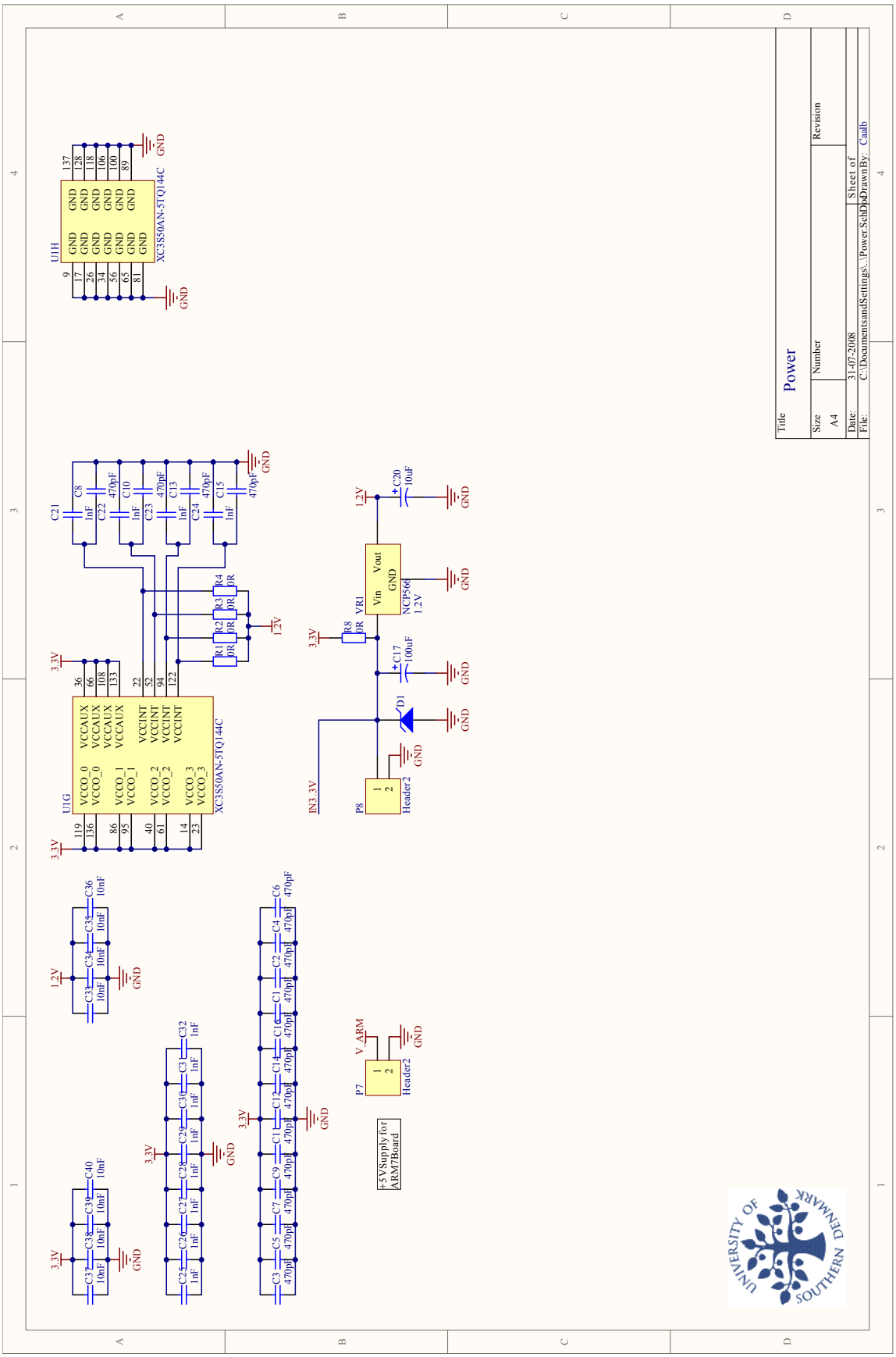
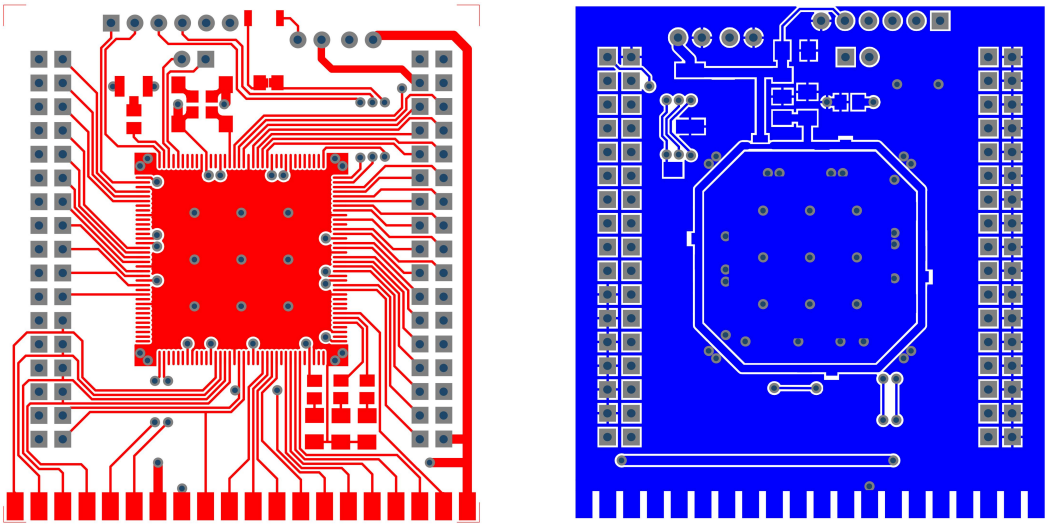


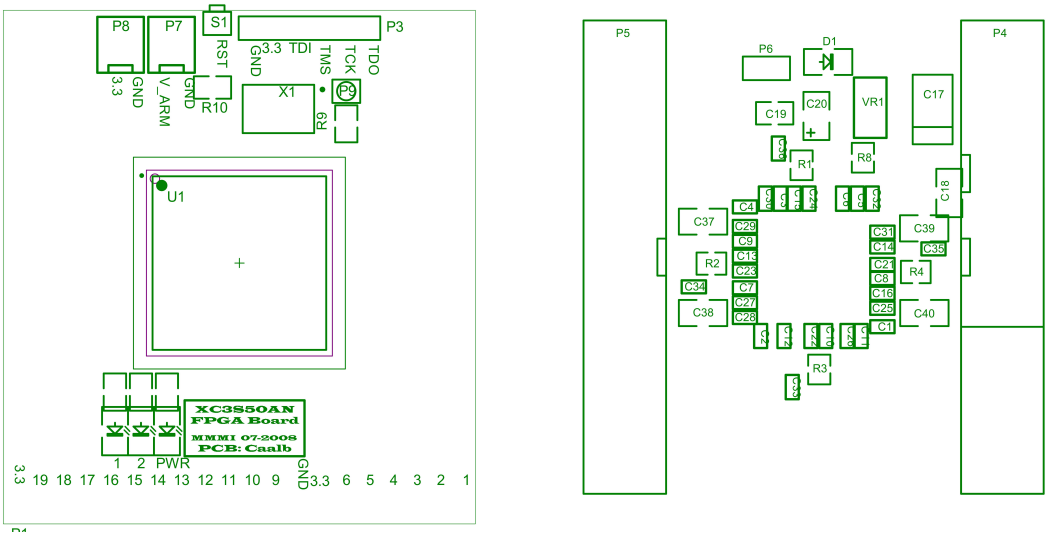
Figure 14: Schematic - page 2 (Power)

C PCB layout



(a) Top (component) side (b) Bottom (solder) side

Figure 15: PCB layout of the S3XB



(a) Top (component) side (b) Bottom (solder) side

Figure 16: Component placement of the S3XB

D Bill of materials

Value	Designator	Footprint	Quantity	Description	Indkøb
100nF	C18	1206	1	!Capacitor	Lager
100nF	C19	0805	1	!Capacitor	Lager
10nF	C33, C34, C35, C36	0603	4	!Capacitor	Digikey: PCC1750CT-ND / Farnell: 722236
10nF	C37, C38, C39, C40	1206	4	!Capacitor	Digikey: 399-1234-1-ND / Farnell: 1414713
1nF	C21, C22, C23, C24, C25, C26, C27, C28, C29, C30, C31, C32	0603	12	!Capacitor	Digikey: PCC1772CT-ND / Farnell: 9406174
470pF	C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16	0603	16	!Capacitor	Digikey: PCC1950CT-ND / Farnell: 722157
100uF	C17	Tantal 100uF 10V (lager)	1	!Polarized Capacitor (Radial)	Lager
10uF	C20	1206Pol	1	!Polarized Capacitor (Radial)	Lager
0R	R1, R2, R3, R4, R8	J0805	5	!Resistor	Digikey: P0.0ACT-ND / Farnell: 1469846
1k	R10	0805 v.solder_blob	1	!Resistor	Lager
1K	R9	0805	1	!Resistor	Lager
330	R5, R6, R7	0805	3	!Resistor	Lager
50Mhz	X1	CB3LV-3C/CFPS-39	1	3.3V, surface mount oscillator	Digikey: CTX283LVCT-ND / Farnell: 1276652
	P6	HDR1X2	1	Header, 2-Pin	Lager
	P7, P8	HDR2x1 M LÅS	2	Header, 2-Pin M LÅS	Lager
	P3	HDR1X6	1	Header, 6-Pin	Lager
	P4	HDR2X10/17FLAD Combi	1	Header, 17-Pin, Dual row	Lager
	P5	HDR2X17FLAD Special	1	Header, 17-Pin, Dual row	Lager
	P1	HDR1X20 Kant Special	1	Header, 20-Pin	Lager
	P2	HDR1X20 Kant	1	Header, 20-Pin	Lager
	D2	LED1206	1	LED	Lager
	D3	LED1206	1	LED	Lager
	D4	LED1206	1	LED	Lager
	P9	Miniature Coaxial Connectors	1	Miniature Coaxial Connectors	Digikey: H9161-ND / Farnell: 3908021
	U1	TQ144_L	1	Spartan-3AN Non-Volatile	Digikey: 122-1555-ND
	S1	B3U-3000P	1	Switch	Digikey: SW1022CT-ND / Farnell: 1333655
3.3V	VR1	SOT-223	1	Volt Reg NCP566	Digikey: / Farnell: 1460689
3.5V	D1	d1206	1	Zener Diode	Digikey: ZMM5227BDICT-ND / Farnell: 8735450

Figure 17: Bill of materials

References