FPGA Design of a Controller for a CAN Controller

Robby Andersson

LiTH-ISY-EX-ET-0191-2003

Linköping 2002-11-20

FPGA Design of a Controller for a CAN Controller

Examensarbete utfört i Elektroniksystem,
Institutionen för Systemteknik,
Linköpings Tekniska Högskola

Av

Robby Andersson

LiTH-ISY-EX-ET-0191-2003

Handledare: Per Gustafson

Examinator: Kent Palmkvist
Linköping 2002-11-20

Avdelning, Institution
Division, Department

Institutionen för Systemteknik
581 83 LINKÖPING

Datum
Date
2003-02-07

Språk	
Language	
	Svenska / Swedish
X	Engelska / English

Rapporttyp	
Report category	
	Licentiatavhandling
X	Examensarbete
	C-uppsats
	D-uppsats
	Övrig rapport

ISBN

ISRN LITH-ISY-EX-ET-0191-2003

Serietitel och serienummer Title of series, numbering

ISSN

URL för elektronisk version
http: / / www.ep.liu.se / exjobb / isy / 2003 / 191 /

Titel
Title

FPGA design av en kontrollenhet för en CAN-kontrollenhet.

FPGA design of a controller for a CAN controller.

Författare
Author

Robby Andersson

Sammanfattning
Abstract

This diploma work describes how an FPGA is designed to control a CAN controller. It describes the different tools used when working with Actel’s design tools and the sequence of work applied. It gives a short overview of a multiplexer, the CAN bus, an analog / digital-converter and some more information on the actual FPGA. It also brings up the design process of the FPGA, planning, coding, simulating, testing and finally programming the FPGA. The different parts implemented in the FPGA are a shift-register and two state- machines that are connected with each other. They work together to control the SJA1000 CAN controller made by Philips. They also receive data from the analog / digital-converter that they forward onwards to the CAN controller that forward the data on the CAN bus.

Nyckelord
Keyword
FPGA, design, CAN, controller, kontrollenhet

Abstract

This diploma work describes how an FPGA is designed to control a CAN controller. It describes the different tools used when working with Actel's design tools and the sequence of work applied. It gives a short overview of a multiplexer, the CAN bus, an analog/digital-converter and some more information on the actual FPGA. It also brings up the design process of the FPGA, planning, coding, simulating, testing and finally programming the FPGA. The different parts implemented in the FPGA are a shift-register and two state-machines that are connected with each other. They work together to control the SJA1000 CAN controller made by Philips. They also receive data from the analog/digital-converter that they forward onwards to the CAN controller that forward the data on the CAN bus.

1 Introduction

At Gutec AB, Linköping, a project was initiated. The goal of this project will remain a secret since it involves patent-rights and other companies.

This diploma work describes programming an FPGA by using VHDL. The FPGA was placed on an already finished circuit board. This meant that the signals to and from the FPGA were known. The FPGA were to work as a host controller to a bus-controller connected to a transceiver for a CAN-bus system, identify itself on that bus-system, send and receive data on the bus, accept data from a A/D-converter, light a diode depending on the data received from the bus and the A/D-converter and, lastly, control a multiplexer.

Several tools had been made available to make this possible. Those tools involved a desktop to manage the files and write the code, a synthesizer tool to create files needed for the programming unit that creates the useable FPGA.

When the FPGA began to take form a small but useful tool was acquired to make it possible to study a finished FPGA signals using a computer and a small unit as an interface between. The company Actel sells these tools. The normal way to design an FPGA today is to create the code necessary to program the FPGA. Before programming, the generated code from a synthesizer tool is run in a simulator tool to see if it will work in theory. Theory and practice does not always match but it's still a good indication if it will work in practice. Then the generated files are used to program the FPGA in a programming device.

1.1 About the report

This report is written for people that have some basic knowledge of electronics. This means that details will not be described on how things work that people with this knowledge should know.

1.2 Project Description

The Diploma work's main assignment was to produce a working FPGA using the hardware-descriptive language VHDL and Actel Desktop.
The work consists of:

- Plan how the FPGA should work with the CAN-bus.
- Producing the code.
- Testing the code in the simulator tool.
- Synthesizing the code necessary to program a working FPGA.
- Testing the circuit board with the FPGA on so that it works with the CAN-bus.

The methods used to account for the above points will be:

1. Thoughts on the FPGA in chapter 3.
2. How the code was written, tested and implemented in chapter 4.
3. The resulting FPGA's function on the CAN-bus in chapter 5.

The Diploma work will be concluded in chapter 5.

2 Concepts

2.1 Signal-levels

Unless otherwise mentioned all signals in this report follow CMOS-levels, +5 Volts, where high is +5 and low is ground.

2.2 VHDL

VHDL is an abbreviation of Very high-speed integrated circuit Hardware Description Language. It is a standardized language used to specify, verify, and design electronics. VHDL was developed by the US department of defense at the beginning of the eighties and was made a standard for modeling and simulating. The translation of VHDL-code to a net-list for e.g. an FPGA is called synthesis. The synthesizing process is not made into a standard thus it is the synthesizer tool that decides what VHDL-code constructs that are supported for synthesis. VHDL is an object-based language. It is a big and general hardware-descriptive language, which gives several opportunities to describe the same behavior with different language-designs. The design itself is made up of components consisting of two parts.

Entity	In and output signals of the component
Architecture	Behavior of the entity, described by different abstraction models.

The component can be described by different abstraction-levels and structural descriptions.

The abstraction-levels can describe the same function but with different levels of detail. When using VHDL it is possible to mix different abstraction-levels simply by connecting the different components that are using different levels. The different levels used in practical electronic design are:

- The Behavior-model
- RTL-Model (Register Transfer Level)
- Gate-level

The behavior model is used at an early stage as a specification on how the circuit is supposed to work, thus it is easy to read and can also be used for documentation.

The RTL-model describes the behavior in asynchrone and synchrone state-machines, bus-structures, operators, registers, multiplexers, ALU and many more structures. These exist in different language-designs that can be synthesized if the synthesis-tool supports it.

Gate level is the lowest abstraction level used for synthesis. At this level a gate-net is written or the design is described using Boolean algebra. This level gives the most control over synthesis and optimizing of circuit-area.

Structural descriptions or design-hierarchies are used in VHDL to hide details. The details are hid in a “black box”, thus only the signals to the “box” can be seen. It is the designer that decides how many hierarchies to use in the design.

Using both language-abstractions and structural descriptions the result will give a decreasing level of detail toward the top of the hierarchy.

When testing the code written in VHDL to verify the correct behavior of a component a test-bench is written. This test-bench is also VHDL-code that generates stimuli to the component. By using the test-bench on the component’s different abstraction-levels it is possible to verify the correct function at the different levels.

More explanations of the language VHDL will be shown in later chapters.

2.3 VERILOG

Verilog is a competing hardware design language to VHDL. It is standardized and developed by an American company that was bought by another company named Cadence. It was created before VHDL and thus is more established, foremost in the USA. In comparison to VHDL, Verilog only exist at a low abstraction-level. Verilog is similar to VHDL but instead of entity and architecture it uses modules. Verilog is more close to hardware, more specific to ASIC and not as general as VHDL.

2.4 FPGA

FPGA is an abbreviation for Field Programmable Gate Array. It is a sub-group to Application Specific Integrated Circuits, ASIC. ASICs are designed at transistor-level. The transistors with connections are made by a design-process involving building layers using e.g. diffusion. A simplification of this is using standard-cells. These are e.g. adders or other functions that are connected to produce the function desired. There are other design-processes but they will not be described here.

What ASICs and the different design-processes of these have in common is that once produced, they can’t be changed.

An FPGA is a gate-matrix in which the connections between and within the cells are programmed. An FPGA then gains the same benefits as an ASIC but the silicon used is larger in the FPGA than in the ASIC. This means the area of silicon isn’t optimized and the speed of the FPGA is slower than for an ASIC.

Another solution to design problems is to use standard-circuits and microprocessors. These are cheap since they are produced in large series. They can be programmed by using a high-definition language, e.g. C. It makes it easy to change or create new code.

An FPGA share both the ASIC and the microprocessor’s benefits. It can be used to fast design application-specific designs and the circuits programmed can easily be changed on the circuit board with a newly programmed FPGA. The drawback is the cost of the FPGA compared to an ASIC in large quantities.

The FPGA functions can be programmed at different levels. One can go directly inside the cells or use higher-level components such as e.g. NAND, adders, registers, flip-flops and more. There are several manufacturers of FPGA. Actel are one, Xilinx another.

2.4.1 Actel FPGA

The FPGA used for this project is an FPGA from Actel, belonging to the 42MX family. This family has FPGAs with the number of gates ranging from 2000 to 36 000. The largest model has SRAM. They come in different packages depending on model, some compatible with other models of the same family, some not. It is designed with different logic modules.

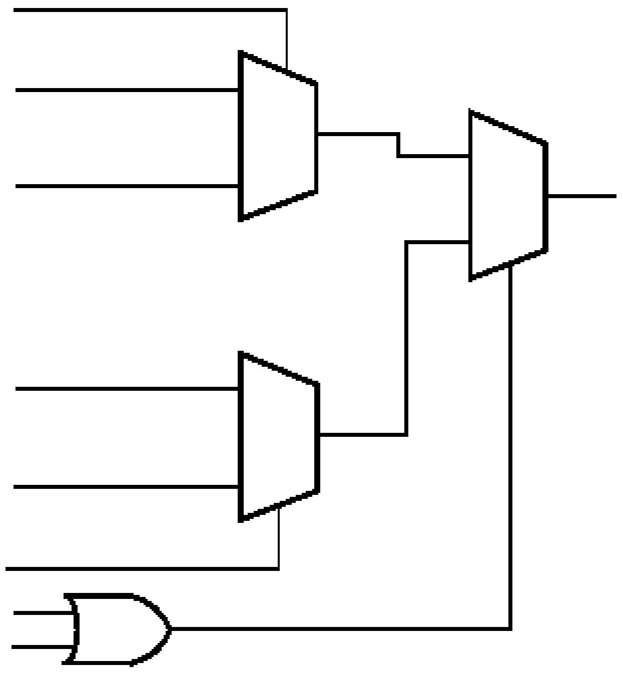


Figure 2.1 Logic module.

The C-module implements logic, the S-module implements the same combinatorial logic as the C-module while adding a sequential element and the D-modules allow the A42MX24 to perform wide-decode functions comparable to CPLDs and PAL devices. The FPGA is programmed by linking these modules together to produce the intended result. The package used in this project was PL84, this means 84 pins, model name A42MX24. It has 1890 modules, 964 sequential, 912 combinatorial and 24 decode.

2.5 CAN-bus

CAN is a serial bus system especially suited for networking "intelligent" devices as well as sensors and actuators within a system or sub-system. CAN is a bus system with multi-master capabilities, that is, all CAN nodes are able to transmit data and several CAN nodes can request the bus simultaneously.

The bit rate depends on the length of the bus and on the hardware used but it can be as high as 1 Mbit/s. Further information can be obtained from numerous sources, see [2].

2.5.1 CAN-bus in this system

In this project a circuit made by Philips Semiconductors were to be used and the bit rate will be 125 kbit/s. It is called SJA1000 and is a CAN controller. It is connected to a transceiver, which in turn is connected to the actual bus. Transceiver is a word derived from “transmit” and “receive”. It is a unit that creates the correct signal levels both for the controller and the actual bus.

The idea is to build a small network of nodes controlled by a mother-node. The FPGA and its circuit board with a CAN-controller is one of several similar boards of these small nodes. The mother-node would be a computer controlling a CAN-interface thus making it possible to send and receive data on the bus controlling the nodes. The mother-node will run alone and at start up it will send out a message with a special format. Any node that has just been connected and started up will react on this message and send back its unique ID with a message and then switch to a work mode. This message will be sent out at regular intervals to check if new nodes have been connected to the bus or if a node has somehow lost its power and gone through its power-up sequence. It is easy to determine the lost power case by detecting that a node that has already reported in reports in again.

2.6 Multiplexer

A multiplexer is a unit that accepts several different in-signals but only sends out one or a few of these depending on a control signal. Maxim designs the multiplexer that is used in this project. It is model MAX393. It is a simple multiplexer made by analog switches with four in-signals and four out-signals controlled by control signals.

In the project only two in-signals, two out-signals and a single control signal will be used, or rather, were supposed to be used. At the beginning of this project the FPGA were supposed to control the multiplexer by shifting between the two in-signals with a single control signal. However, at a later point in the project it was discovered that one of the in-signals were faulty and therefor unusable. This meant that the multiplexer could be set into a single mode, making it redundant. The remaining input signal could just as well be directly connected to the FPGA. This of course made it easier to program the FPGA.

2.7 Analog/Digital-converter

An analog/digital-converter does what the name implies; it converts an analog signal into a digital signal. Burr-Brown manufactures the model used in this project. It is a serial 16-bit sampling A/D-converter. The range on the analog input signal is 0 - 4 V; this range is then converted internally onto a 16-bit range with the MSB as the sign-bit.

This will be demonstrated by an example taken from the data-sheet produced by Burr-Brown (note that it is two's complement number representation relative the middle level):

+Full Scale-1LSB	0111 1111 1111 1111	4 V
Mid Scale	0000 0000 0000 0000	2 V
Mid Scale –1 LSB	1111 1111 1111 1111	~1.99 V
-Full Scale	1000 0000 0000 0000	0 V

The converter can be set to work in a few different ways. Only the way it was used in this project will be reported. The converter is set to use its internal clock; this means a single signal is enough to control it. This signal is called $\overline{CONV}$ in the data-sheet. When the converter detects a falling edge on $\overline{CONV}$ it begins a conversion of the current analog input signal. This in turn generates some output signals. See figure 2.2. These signals, BUSY, DATACLK, and DATA, will be used in the FPGA. See below.

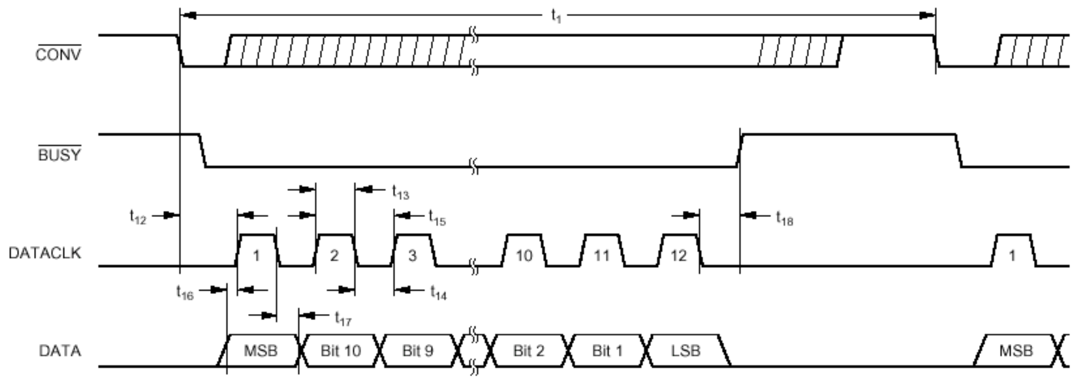


Figure 2.2 Serial Data Timing, Internal Clock.

2.8 SJA1000 CAN Controller

This is by far the most complex unit of the circuits the FPGA has to interface to. It needs to be initialized to work properly. The data-sheet and application note on the controller is extensive. This means that for those readers that wish to study the controller more in detail needs to download the data-sheet and application-notes from Philips. See [4] for further details.

2.8.1 Signals

The controller uses several signals plus an eight-bit wide bus to communicate with the host-controller, that is the FPGA. Inside the controller there are several registers that are used to control its behavior. The FPGA sets up these registers during an initialization-process and then reads/writes certain registers at run-time.

2.8.2 Initialization

There are two ways to initialize the controller, the hardware reset and the software reset. During power-up of the whole circuit board that includes the CAN-controller and the host-controller (the FPGA), among other signals, a reset-pulse is created. It is used to set these two circuits in a known initial state. After power-up the FPGA begins by setting up the "mode register" of the controller and then polls it to see if it is ready to be initialized. Once it receives a satisfactory answer it proceeds by setting up several other registers and then enters a "decision mode" where it waits for further instructions, e.g. to send a message on the CAN-bus. The host-controller may also send a software reset request that initializes the software reset. It is used to re-configure the controller during operation.

2.8.3 Sending or receiving a message

Sending a message on the CAN-bus, or receiving, is completed by reading/writing certain registers in the controller. This process can be handled in two ways, either by interrupts or by polling control. In this project was the use of polling selected. In the case of sending a transmission, it was decided to just send it, not even checking whether or not it was okay to start a message. This choice was based on the knowledge that at the beginning of the project the bus-system would not be heavily used and also slow. This meant that any message being transmitted would be sent long before any subsequent message would want to be sent.

2.9 Development Tools

Below is a summary of the different development tools that were used.

2.9.1 Actel Desktop

The Desktop is used to create projects and manage simulation and synthesis. In these projects all the information is stored. It contains tools for handling files such as schematics, VHDL-files, simulation-files and other files. It consists of several different parts that are described below. The Desktop was both a curse and a blessing, but mostly a blessing luckily. It had a tendency to crash sometimes, it did not forget old file-links in some cases and it refused to get rid of files generated by an internal tool, even if those files were obsolete.

2.9.2 Design View

Design View includes a VHDL-compiler, the possibility to choose an editor such as the Windows notepad or emacs for writing the code in, a tool to generate simulation-stimuli by a graphical interface that generates complete VHDL-code for the simulation tool Veribest VHDL simulator, and much more. It is the base from where other tools needed to finish the FPGA are started, such as the simulator or synthesizer. These can also be started stand alone but it is straightforward to start them from Design View.

In Design View a project are started by setting a few details. These details can be changed later, e.g. VHDL flavor and libraries to be used. In the project the VHDL code is written, schematics drawn, and the hierarchy in the project is managed by an interface. All this makes it easy to complete the project to its final shape.

2.9.3 Veribest VHDL Simulator

This tool made it possible to verify and test the VHDL code that had been produced. It contains a compiler. It offers the opportunity to look at any signal used in the VHDL code, both the code before synthesizing and the code generated after. It produces curves that can be viewed in detail making it possible to get a good overall view if things are working as they are supposed to. It also generates useful messages about events such as e.g. spikes on signals.

2.9.4 Synplicity Synplify

This tool is used to synthesize the code and produce a net-list that can be used by a routing tool, which in turn produces the files necessary for the programming tool. It contains a compiler and has several settings to help produce the best result. It produces code that can be used in the simulator to check if the synthesis has produced expected result.

2.9.5 Place and Route

This is invoked in Design View and it is used to produce the files needed for the programming tool. It can produce files needed for tools such as Silicon Explorer, to look at timing-characteristics and what pins certain out-signals should use in the FPGA if that is required. Fixed pin placement can create problems with optimization inside the FPGA since it cannot choose pins according to the best routing. This was unfortunate but something that had to be accepted.

2.9.6 Silicon Sculptor

This is the actual programming tool, the tool that produces the finished FPGA. It takes files produced by the place and route process and then programs the FPGA in a programming device. It is simple to use with a menu-system and feedback on the progress.

2.9.7 Silicon Explorer

This small but useful tool is used to examine the signals in an already programmed FPGA. To be able to use it certain flags must be set when creating the files necessary to program the FPGA. This of course takes up space inside the FPGA but it is very nice to have this opportunity to verify that everything is working as expected. It uses a small box connected to the computers parallel port and the actual pins on the FPGA. On the computer signals can then be viewed similar to how signals would look like when using an oscilloscope. It can also be used to control signals inside the FPGA.

3 Function of the FPGA

The FPGA would have to work with two circuits with inputs, the CAN-controller and the A/D-converter. At the beginning of the project it should have worked with a multiplexer too but for reasons explained earlier it would not be necessary.

The FPGA would need to be reset to start in a known mode much like any other piece of electronics at power-up. This reset signal was generated by the circuit board the FPGA is placed on. It is pulse that starts low, stays high for 20 ms and then goes low again, long enough to reset anything on the circuit board that needs to be reset.

Any signals that control the CAN-controller would go through the FPGA. This means the clock and reset of the controller would be routed through the FPGA.

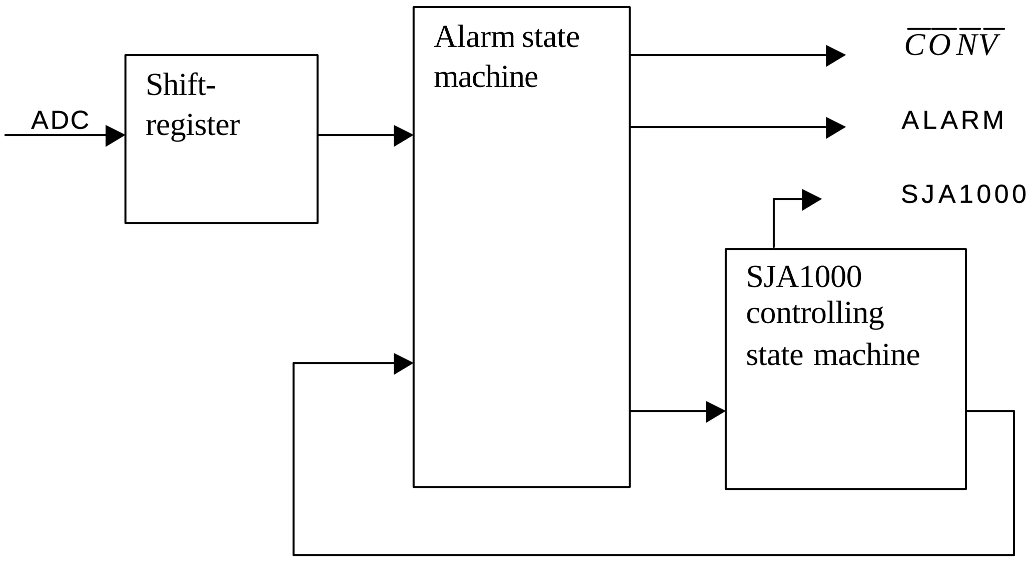


Figure 3.1 Figure of FPGA with signals connected to it.

The FPGA would do the following:

- Send the value obtained from the ADC when told to do so on the CAN-bus.
- Change the threshold between comparing of the value obtained from the ADC and a value sent on the CAN-bus.
- Set a signal high for five seconds if the ADC value is higher than the threshold and keep it high as long as the ADC value is higher.

3.1 Clock

A crystal working at 24 MHz generates the clock-signal for the FPGA. This clock-signal is also routed through the FPGA and used to clock the CAN-controller.

3.2 A/D-Converter

To make this unit start a conversion a falling edge signal must be sent to it. The idea was to make it start a conversion with a frequency of 1500 Hz. The converter can complete a conversion at 20 µs maximum according to its data sheet. A frequency of 1500 Hz means a period of about 666 µs so the conversion will be finished well in time for the next one.

This means the FPGA has to generate the signal that starts the conversion, the $\overline{CO}\overline{NV}$ signal, at about 1500 Hz. A 14-bit counter working at the FPGAs clock-frequency will generate this signal. This means it will not be exactly 1500 Hz but close enough, about 1465 Hz. Once a conversion is initiated the result must be taken care of by the FPGA. The A/D-converter generates three signals in response to the falling-edge $\overline{CO}\overline{NV}$ signal. These three signals can be viewed in figure 2.2, chapter 2.7.

It was decided to use a 16-bit serial-in/parallel-out register clocked by the A/D-converters DATACLK signal. It was enabled by the A/D-converters BUSY signal and its DATA signal was connected to the serial-in of the register. It was reset by the general reset signal connected to the FPGA.

3.3 SJA1000

The CAN-Controller needed to be reset and the FPGA would do this as its host-controller. To initialize it, it was decided to create a state-machine. It would also need a clock and according to its data-sheet it could handle 24 MHz, so the same clock as the FPGA would clock it. It also needed to be set in a special mode according to its data sheet. It was put it in Motorola mode, the other mode being Intel. This meant the Motorola mode read/write-cycle diagram would be used. Thus the SJA1000s MODE signal would be set to low.

Which one of these two different modes that should be chosen depends on what kind of host-controller that is used to control the SJA1000. The normal way to control the CAN-controller is probably by using a microprocessor by a brand of above-mentioned companies. Since the SJA1000 would be controlled by the FPGA it really didn’t matter which of these two modes were used, and the Motorola mode was chosen before Intel simply because it looked easier to work with.

Another signal that needed to be set was the $\overline{CS}$ signal. It is simply a signal that controls whether or not the SJA1000 that receives this signal should be active or not. This is only necessary to control if there are several CAN-controllers on the same circuit board connected to the same bus. In this project it is only needed be set to low to make it active since it would be the only CAN-controller the FPGA would control.

The FPGA would read and write from and to the SJA1000 a lot. This meant the read/write process would be repeated a lot, so it was decided to design the FPGA so the SJA1000 easily could be accessed by different data. As mentioned earlier in chapter 2.8 the SJA1000 is the link between its host-controller and the CAN-bus. All it does is to relay information to and from the bus, thus, it is up to the FPGA to handle whatever the SJA1000 should. Further descriptions on how to initialize and handle the

SJA1000 will be described in chapter 3.4. See figure 3.2 for a description of the SJA1000 read/write cycle.

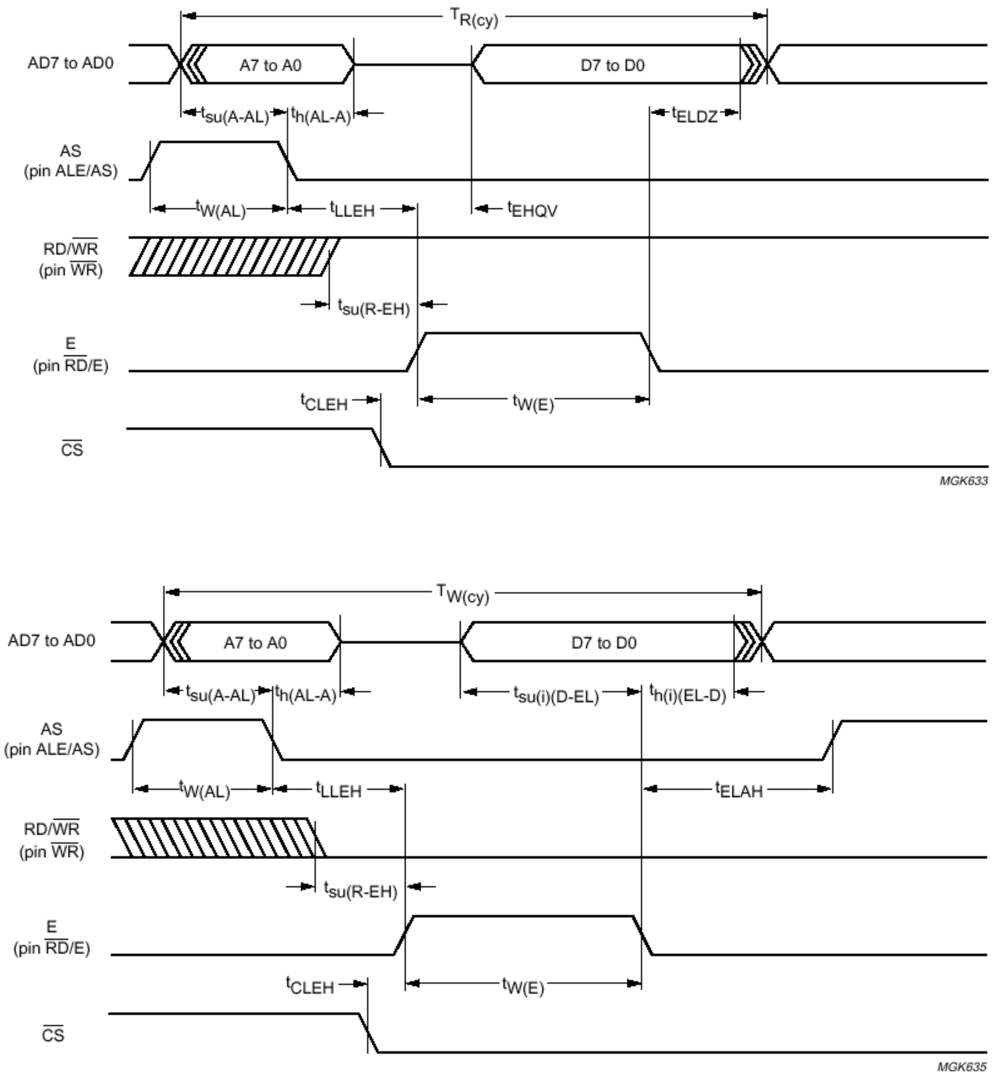


Figure 3.2 Read and write of SJA1000 in Motorola Mode.

3.4 FPGA

This is the central piece of this project; it handles and controls everything in it. Below is a picture of its internal structure.

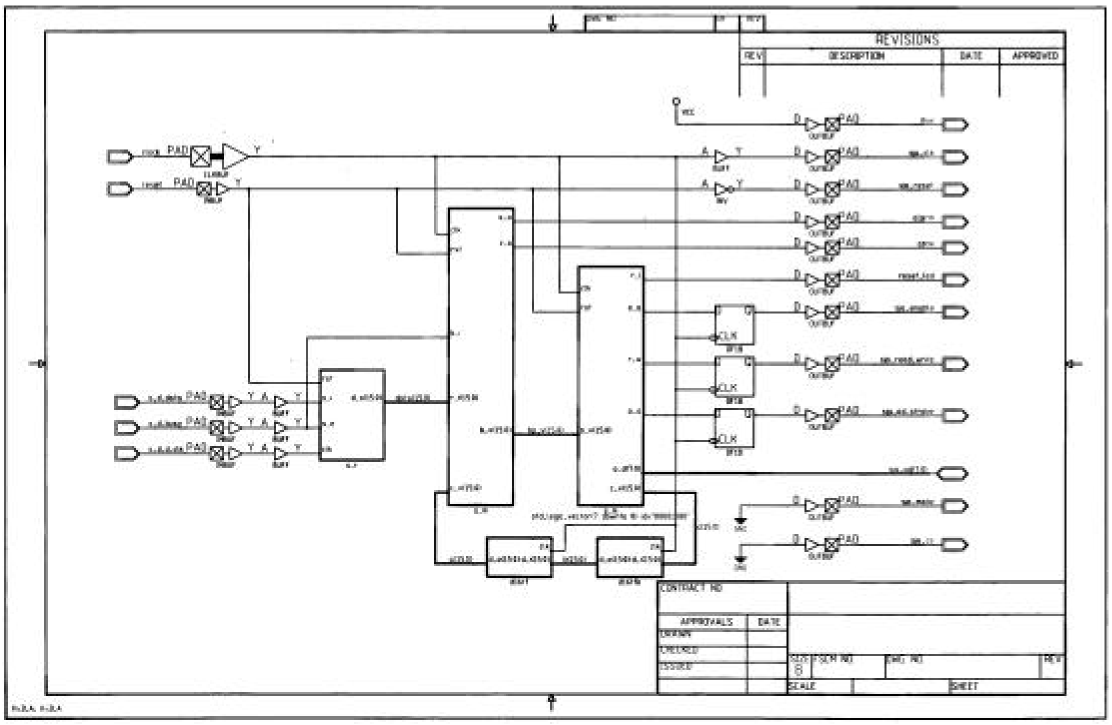


Figure 3.3 Figure of the insides of the FPGA. See larger version in appendix B.

As can be seen there are five in-signals, eleven out-signals and one bi-directional bus signal. It was decided to split it up into three different segments, a shift-register serial-in/parallel-out that receives data from the ADC, a state-machine that initiates the ADC conversion cycle and send out an alarm signal, and the SJA1000 control state-machine. This last segment would control everything that has to do with the SJA1000. The two rectangular blocks at the bottom of the design are D-flip-flops.

3.4.1 Shift-register

This is a simple serial-in/parallel-out shift-register with a reset signal, a serial-input signal, a shift enable signal, a clock signal and a 16 bit-wide parallel output signal. When a reset signal is obtained at power-up it will put the parallel output signal to a binary “1000 0000 0000 0000”, the lowest possible input value from the ADC. See 3.2 for a description of how it works together with the ADC.

3.4.2 Alarm state-machine

This state-machine has four tasks. First, it should generate the $\overline{CONV}$ signal for the ADC. The most significant bit of a simple 14-bit wide counter that counts every clock-cycle generates a signal working at almost 1465 Hz. This counter is integrated into the state-machine.

Second, it should load data from the shift-register when the shift-register is not receiving any data from the ADC. Checking the BUSY signal sent from the ADC does this. If it is high then no data is being transmitted from the ADC to the shift-register and its contents can be read.

Third, it should compare the value obtained from the shift-register with a value it gets from the SJA1000 control state-machine. If the value obtained from the shift-register is higher than the comparison-value it will send out an alarm-signal as long as this occurs while new values are read from the shift-register. It will keep this alarm-signal high for five seconds after a satisfactory comparison has been achieved once again.

Fourth, it will keep the value it has obtained from the shift-register so the SJA1000 control state-machine can read it when it wants to. This is related to the SJA1000 control state-machine. It needs this value when it is told to send it on the CAN-bus to the mother-node, see below.

3.4.3 SJA1000 control state-machine

This state-machine initializes the SJA1000 after power-up and then reacts to what the SJA1000 receives on the CAN-bus. It would be large since there are a lot of registers that needs to be set. It should have a part where data can be written to the SJA1000 and another part where data sent from the SJA1000 to the FPGA can be read. It should have its own unique identification number. This number do not need to be large since there will not be many nodes on the CAN-bus. Five bits is enough giving a range of 0 to 31. After the overall reset pulse has passed and operation has begun it should start by writing to the Mode-register of the SJA1000 thus forcing the SJA1000 into a reset-mode, read the same register to conclude that it really is in reset-mode and then proceed by setting up several different registers. See figure 3.4 for an overview of the initialization process of the SJA1000.

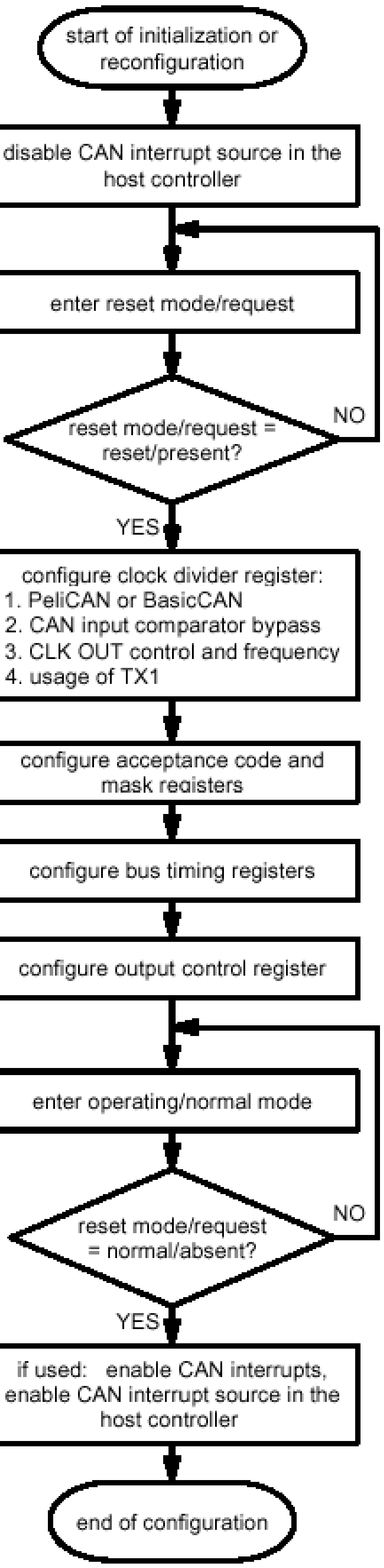

start of initialization or reconfiguration
disable CAN interrupt source in the host controller
enter reset mode/request
reset mode/request = reset/present?
NO
YES
configure clock divider register:
1. PeliCAN or BasicCAN
2. CAN input comparator bypass
3. CLK OUT control and frequency
4. usage of TX1
configure acceptance code and mask registers
configure bus timing registers
configure output control register
enter operating/normal mode
reset mode/request = normal/absent?
NO
YES
if used: enable CAN interrupts, enable CAN interrupt source in the host controller
end of configuration

Figure 3.4 Init-process taken from application note, see [4].

As can be seen there are a few registers that need to be set. Most of them are only related to the operation of the SJA1000, such as which mode it should be in, bus timing, filtering mode and output control are others.

The Acceptance-mask- and Acceptance-code-registers should be set up so the SJA1000 work as described in section 2.5.1. The Acceptance-code-register tells the SJA1000 which messages it should listen for and the Acceptance-mask-register tells it which bits in the Acceptance-code-register are really relevant. This means that the mask-register controls the code-register.

In this project the SJA1000 would work in PeliCAN mode using a single long filter to filter incoming messages of Extended Frame Format, EFF.

Below is an example of how the registers work to filter messages. This actually describes how the registers are to be set up first after power-up.

ACR0	ACR1	ACR2	ACR3
1000 0000	0000 0000	0000 0000	0000 0XXX
AMR2	AMR1	AMR2	AMR3
0000 0000	0000 0000	0000 0000	0000 0111
Accepted messages:			
1000 0000	0000 0000	0000 0000	0000 0xxx

("X" = irrelevant, "x" = don't care)

It can be seen that if the AMR bits are set to "1" the ACR bits do not matter. This means the SJA1000 can be set to listen to messages that look like the ACR but have certain bits not matter, thus making it possible to let through certain messages that looks almost like the ACR but with some few differing bits.

In this example the three last bits in ACR3 and AMR3 are set to "1", it shows how these two registers work, but the Data Sheet of the SJA1000 shows that these three last bits are not part of the actual received message. The example is still correct. The two last bits are reserved and the third is used for something called Remote Transmit Request, something that will not be used. According to the Data Sheet these three bits should be set to "1" in the AMR3.

This means that the SJA1000 will only listen to messages that look like this: "1000 0000 0000 0000 0000 0000 0000 0"

This is the special format message mentioned in part 2.5.1. When the SJA1000 receives this the first time it goes through a soft reset initiated by the FPGA that rewrites the Acceptance-code- and Acceptance-mask-registers again. This time to:

ACR0	ACR1	ACR2	ACR3
0000 0000	XXX0 0000	0000 0000	IIII IXXX
AMR2	AMR1	AMR2	AMR3
0000 0000	1110 0000	0000 0000	0000 0111
Accepted messages:			
0000 0000	xxx0 0000	0000 0000	IIII Ixxx

("X" = irrelevant, "x" = don't care, "I" = identity bit)

The three bits in AMR1 will be used to send specific messages on the CAN-bus to the node that the FPGA can react to, making it possible to send eight different messages to it that it can react to. The identity bits are used to filter messages to the specific node only.

Only two messages out of the eight possible would be used, leaving six extra for future designs. These two messages would be a request to send the latest obtained data from the ADC and to set a new compare value for the alarm-state-machine.

A read or write to the SJA1000 is done as described in figure 3.2, so this behavior would need to be created. Since a lot of reading and writing would go on an extra register was created to keep track of which state to jump to after each read or write request to the SJA1000. This means a part of the state-machine would handle reading from and another writing to, both ending with a jump to several different states depending on which state it came from, which is handled by the extra register.

4 Implementation of the FPGA

It is easy to start a project when using the Desktop, just enter a few parameters in a wizard and begin. This was done by specifying the path to where the project should be saved with all its files, the name of the project, the VHDL flavor, and what technology family that would be used. The –93 flavor seemed the best choice since it was the latest. The technology would be the Actel 42MX. That chosen it was time to start creating the design.

Since the building blocks of the FPGA had been sketched and it was known what they would look like, a bottom-up approach seemed appropriate to use. This meant the building blocks would be coded in VHDL first and then connected in a schematic. By doing it this way the VHDL code blocks could be simulated individually and then when the whole design were put together it would be known that at least the building blocks would work.

4.1 State-machine building blocks

It was decided to code the state machines using the Moore approach two-process FSM. It means that no output logic of them would be connected to their input logic, that's generally a safer way of designing. It was also decided to make the reset of them asynchronous. If asynchronous resets can be used, that work independently from the clock, use them. They establish the initial state, which makes logic simulation easier and puts the entire design in a known state. Writing the code was easy using Design View, just open a VHDL text window and enter the code. Checks can then be run on it as the coding proceeds to see if the syntax is correct. The state-machines are not special in any way compared to other machines of their type. The SJA1000 control state machine has an extra register that keep track of where it should jump after a read or write to SJA1000 thus making it a bit different.

4.2 Testing the VHDL code

This was easy to do since a tool for creating VHDL files for the simulator tool was available. Right-click on the VHDL file and a waveform creator window would open. The signals in the entity of the VHDL code would be shown in the window. It is then easy to set the clock input signal and other input signals for the VHDL entity that needs to be created. When satisfaction is reached with the result, save, and a stimuli file is generated. Then assign the generated file to the VHDL file that needs to be to simulated and open the simulator. The generated file contains VHDL code and if preferred it can be written from scratch. The tool is good for making a skeleton code. Then enter what is needed as simulation proceeds.

There are two methods of testing, pre-synthesis and post-synthesis. The stimuli file generated and manipulated could be used for both simulations, thus simplifying testing by not forcing creation of new files. Pre-synthesis is easier to simulate since it is easier for the computer to calculate the signals behavior. The pre-synthesis method was used heavily until the behavioral of the code were correct. When this was the case it was synthesized and tested to make sure it worked afterwards synthesis. Post-synthesis simulation is useful to detect e.g. logic glitches or plain wrong behavior that the synthesis tool somehow has created. In the two simulations signals can be tracked. This means signals in and out from different parts of the design can be examined, e.g. an OR-gate somewhere in it or just the signals of a state-machine. It is easy to choose which signals to examine by clicking on menus.

4.3 Synthesis of VHDL code

When the behavior of the VHDL code was accepted it was run through the synthesis tool. This was easy to do, just point at the file needs to be synthesized, right-click and choose synthesis. That invokes Synplify with all files necessary for it to work connected.

In Synplify some parameters need to be set such as what clock frequency the design should use, what technology and other such things. Unfortunately the version of Synplify was the light-version. Many of the more useful tools were not available, like a viewer of how the tool actually synthesized things at gate-level or use of constraint files for special rules of the synthesis. Two things that were useful were the Resource-sharing and Symbolic FSM compiler. When used the synthesis tool tried to optimize and reuse logic for the different parts in the FPGA, saving resources. The Symbolic FSM Compiler function in Synplify automatically extracts state machines in symbolic form and re-encodes them for optimally performance and area implementation. The Compiler also performs reach ability analysis and eliminates unreachable states. When all initial criteria have been set, let the tool run and do the work. When it is finished it creates a net-list plus VHDL code. The VHDL code is used to run post-synthesis simulation and a place and route tool uses the net-list.

4.4 Schematics

When the VHDL code and the blocks were completed, symbol blocks were generated for them so they could be used in a schematic implementation. Generating symbols was easy, right-click on the VHDL file and click “Generate block symbol”. The Design View generates a symbols with the entity’s in- and out-signals, ready for use in a schematic.

The schematic drawing is done in Design View. Instead of creating a VHDL file a schematic is created. Connected to this schematic is the technology used. This means it is possible to go down to the lowest level and find e.g. an OR-gate and put it on the schematic. This can used to draw both simple and complex designs, or to use VHDL code, like it was used in this project, to connect blocks by their symbols and then AND two of their signals.

In the schematic pad connections needs to be specified so that the synthesis tool know which signals is connected to which pad of the FPGA. Each pad is connected to one of the FPGA’s physical pins, thus if the best possible clock fan-out is wanted the clock is connected to a clock pad. By doing this the clock net already drawn inside the FPGA is utilized. See [3] for more information regarding the FPGA. It is also necessary that all signals are drawn to the right kind of pad and buffered correctly.

The design was drawn as planned, connecting the blocks and checked for errors. Any errors were corrected and the whole schematic was then simulated pre-synthesis. When it was established that the behavior was all right it was synthesized and a post-synthesis simulation was run. The output signals of the SJA1000 control state machine post-synthesis had glitches so three D-flip-flops were added to make sure there would be no glitches. Finally satisfied with the post-synthesis simulation things could now move on to the last stage, the place and route of the FPGA.

4.5 Place and Route

This is the last stage of the design process of an Actel FPGA. When the simulations are correct and synthesis results acceptable open this tool in Design View by right clicking and choosing “Place and route”. This can only be done once a synthesis has been done. It will open up a tool that takes the net-list generated from the synthesis tool and uses it to create a file for the programming device. This was fairly straightforward. Open the tool, set the FPGA technology used, the temperature range and which specific model. Then run a “compile”. If there are any reports of warnings or errors, check and correct them and run the place and route again until satisfied. Once the compile reports no errors or warnings it is optional to use pin edit to define which signals go to which pin on the FPGA. This had to be chosen since the card the FPGA were to be soldered on was already finished. That can hamper the router tool since the pin might be uncomfortably placed but there was nothing to do about it. Luckily the design fit well inside the FPGA, about 30% of the resources (in an A42MX24) was occupied. When the routing was finished there was only one more thing to do and that was to generate the file for the programming device.

This tool also had the option of generating a “probe-file” that could be used in Silicon Explorer. For it to be useful the routing would have to reserve special pins on the FPGA that would be used for this purpose. This can also hamper the routing if it is a dense design that almost does not fit. The design fit well so a probe file could be used and was. See [3] for the FPGA concerning these special pins.

4.6 Programming the FPGA

The FPGA could be programmed using a program called Silicon Sculptor and a device connected to the printer port of a PC. Put the FPGA in the device, load the file generated by the Place and Route tool and set a few criteria and it is ready to be programmed. Some things needs to be specified like what technology about to be programmed and where from to load the file. In the program a check can be performed on the programming device to establish that it works properly, and this was done every time to make sure the programming would go well.

4.7 Silicon Explorer

This little tool is handy for checking why a finished FPGA does not work as intended. It is a small box powered by and connected to the reserved pins for this purpose on the FPGA. The COM port of a PC controls it. With the probe file generated by the Place and Route tool it is possible to find all the signals in the FPGA that is connected with the net-list. By using an oscilloscope the signals can then be examined or what state a state-machine is in can be determined. This was used to find out what state machine didn't work and check the code and synthesis for that one. As stated before, theory and practice do not always get along. The simulation showed it should work post-synthesis and it did not. Silicon Explorer helped narrow the search, find the problem, and fix it.

4.8 Summary

Below is a figure of the sequence of work that was applied until a satisfactory result was obtained.

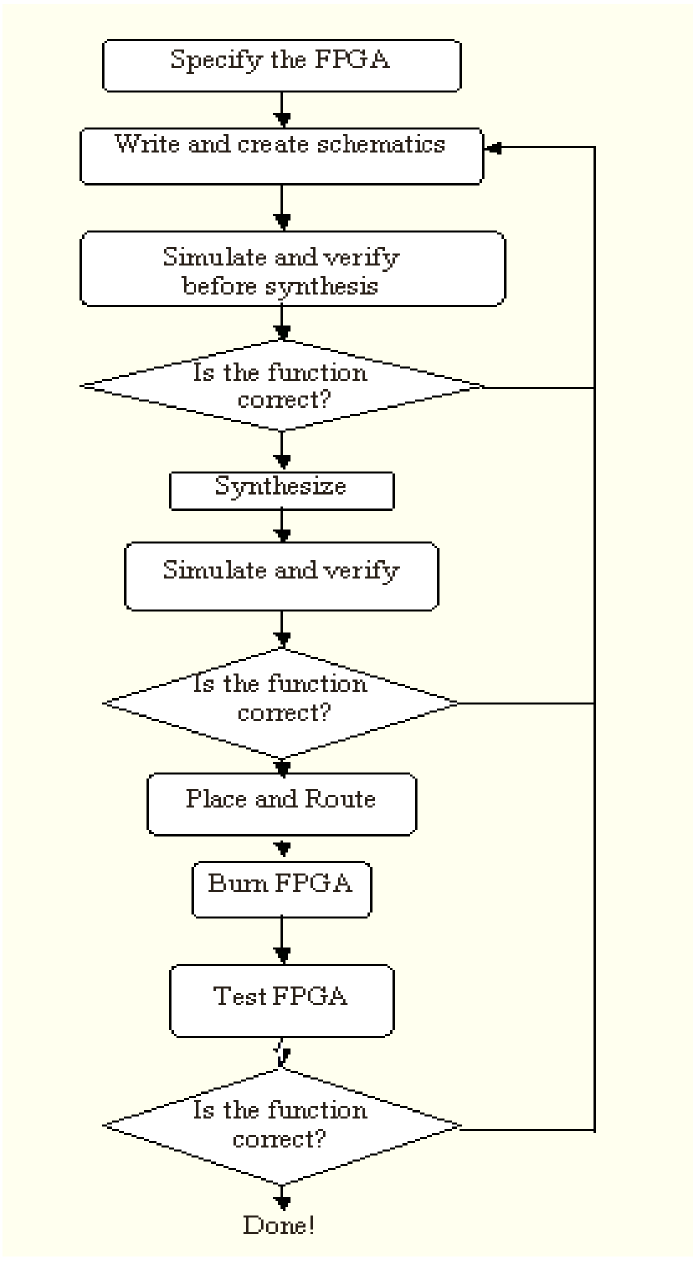


Figure 4.1 Working sequence.

5 Results

The FPGA worked, it initializes the SJA1000 and can correctly receive messages to change the threshold, send data back on the CAN-bus, and reacts to a value higher than the threshold from the ADC as specified. Setting up a CAN-bus, connecting the node with the FPGA and observing the traffic using a CAN-program on a PC was used to test this.

5.1 Enhancements

- There could be a better way to handle the reading and writing of the SJA1000.
- In its current implementation transmission polling is not done, perhaps this need to be fixed.

5.2 Conclusion

This diploma work shows how an FPGA can be designed using tools from Actel. It shows several important parts of the design process applied, a process used by many engineers around the world. Specify the function of the FPGA, write the code, simulate, synthesize, simulate, place and route, go back and fix bugs and repeat until the goal of a working design is reached. All the parts of the design process are here and it is fair to mention a lot was learned, for example, how to plan the work, how to program and how to find something that is wrong and fix it. Working with the tools from Actel didn't differ much from working with any other tools encountered before. There were minor bugs and it sometimes crashed but that can happen because of a lot of reasons.

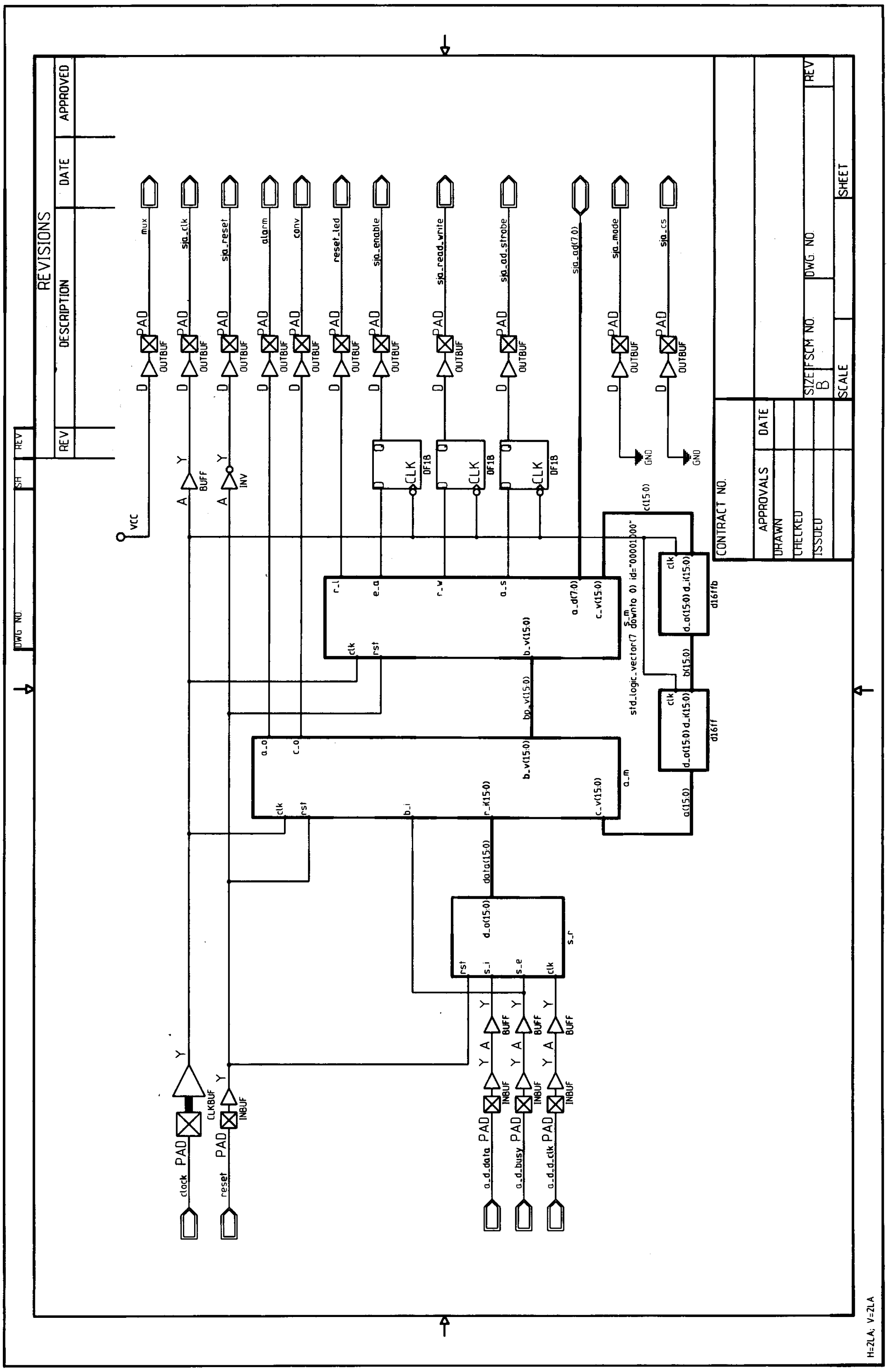
Reference

1. Skahill, Kevin. VHDL For Programmable Logic. Addison-Wesley 1996.
2. http://www.kvaser.com/ CAN-bus solutions.
3. http://www.actel.com/ FPGA plus software links.
4. http://www-us.semiconductors.philips.com/cgi-bin/pldb/pip/sja1000/ SJA1000 product information plus links to Data sheet and Application Note.

Appendix A - Abbreviations

ACR	Acceptance-Code-Register
A/D Converter, ADC	Analog/Digital Converter
ALU	Arithmetic Logic Unit.
AMR	Acceptance-Mask-Register
ASIC	Application Specific Integrated Circuit.
C	A programming language.
CAN-bus	Controller Area Network, a serial bus.
CMOS	A logic family in electronics.
CPLD	Complex Programmable Array Logic
D-flip-flop	A Bi-stable register where data is stored when synchronized by a control signal, a clock.
EFF	Extended Frame Format, a mode in the SJA1000.
FPGA	Field Programmable Gate Array.
FSM	Finite State Machine
Latch	The out-signal is equal to the in-signal when a control signal is set.
LSB	Least Significant Bit
MSB	Most Significant Bit
PAL	Programmable Array Logic
PC	Personal Computer
Register	Physical storage-medium for data controlled by electrical signals.
Routing	Translates the net-list from synthesis to the FPGA specific programming-file usable by the programming tools.

Appendix B - Schematic



REVISIONS
REV
DESCRIPTION
DATE
APPROVED
mux
sja_clk
sja_reset
alarm
conv
reset_led
sja_enable
sja_read_write
sja_ad_strobe
sja_ad(7:0)
sja_mode
sja_cs
PAD
OUTBUF
D
DF1B
CLK
GND
VCC
BUFF
INV
A
Y
r_l
e_a
r_w
a_s
a_d(7:0)
c_v(15:0)
clk
rst
b_v(15:0)
s_m
std_logic_vector(7 downto 0) id="00001000"
c(15:0)
b(15:0)
a(15:0)
d16ffb
d16ff
d_o(15:0)
d_i(15:0)
bo_v(15:0)
a_o
c_o
b_i
r_k(15:0)
a_m
data(15:0)
s_i
s_e
s_r
clock
reset
CLKBUF
INBUF
a_d_data
a_d_busy
a_d_clk
CONTRACT NO.
APPROVALS
DRAWN
CHECKED
ISSUED
SIZE
FSCM NO.
DWG NO.
B
SCALE
SHEET
SH
H=2LA; V=2LA