UMONS
Université de Mons

Faculté Polytechnique

Tutorial: driving the ADC of the FPGA board Altera DE1-SoC (rev. D)

Course: Hardware and Software Platforms

Master (first year) in electrical engineering, specialism 'Multimedia and Telecommunications'

Nikola JOVICIC
Quentin TEDESCHI

POLYTECH MONS

Professor : Mr. Carlos VALDERRAMA

June 2019

pôle hainuyer

Contents

Introduction

This tutorial will present the driving of the ADC integrated in the FPGA Altera DE1-SoC rev. D. The ADC model used in this project is a AD7928, from Analog Devices. It is a 8 channel, 12 bit resolution device working at a frequency of 12.5 MHz.

The section "Useful information and requirements" is important to read, because it establishes the basis of this tutorial. It gives the user the link to the Github repository, the documentation, codes and specifies the requirements to be able to do this tutorial. There is also a section "getting started" that will quickly explain you how to start the project and what files you should use and in which case.

The first part will explain how the ADC works and what are its main parameters. The second part will present the driver of this peripheral and the waveform simulations to show what results should be obtained. The last part will show a practical implementation that allows the user to choose one of the eight channels of the ADC.

Chapter 1

Useful information and requirements

1.1 Github repository

This project has a Github repo that can be found at the following URL: `https://github.com/njlora/Altera-DE1SoC-ADC`

This repository contains all the documentation (to understand how the ADC works) and the project files that have been used in Altera Quartus II v15 (driver, applications and testbenches).

1.2 Documentation and codes

The documents that need to be read to understand this tutorial are located on the repo at *Altera-DE1SoC-ADC/documentation/*. There is one .PDF file that presents the ADC itself (we recommend that you read and understand this part) and the user manual of the FPGA board (used in this project to understand how the 7-segments LEDs work).

The codes of the driver, the application and testbenches is also on the Git repo at *Altera-DE1SoC-ADC/hdl_codes/*. The driver's file is "adv_adc.v" and the application file is "Bouton.vhd". Testbenches are will always have "TB" in their names.

1.3 Software

This tutorial has been made using Altera Quartus II v15. This version of the software has already a driver for the ADC in the *IP catalog*. Lower versions of Altera Quartus II do not have *IP catalog*. Authors of this tutorial cannot guarantee that other versions of Altera Quartus II will work with the provided files.

1.4 Requirements

This tutorial assumses that you know how to use the basic functionalities Altera Quartus II (create a new project, add files to project, compile, simulate with "modelsim", ...). The authors of this tutorial that provide the files and the explanations are not responsible for any damage caused to the material by their codes or explanations.

1.5 Getting started

Launch Altera Quartus II on your computer and create a new project (give it a name that you can recognise later - we have named ours "ADC_DE1-SoC"). Choose a path on your computer where you can easily find the project afterwards. Be careful to select the appropriate model of board that you own when the program prompts you to chose it. After this, you are ready to proceed. The files that you should have for your project are :

- adv_adc.v : the Verilog driver of the ADC,
- adv_adc_TB.vhd : the VHDL testbench of the driver,
- Bouton.vhd : the file of the application that will be implemented with the driver,
- TB_final.vhd : the testbench that regroups the driver and the application and simulates their behaviour when they are connected together.

This tutorial will always explain the theoretical background and the functioning of the components before doing anything practical. For the simulations, the tutorial will indicate which files must be compiled and what results should be observed. This will help you when you will perform your own simulations and check that everything is correct.

We advise you to never flash files on the FPGA board if the simulation results are not coherent or if you get errors in the waveforms (U->unknow values, X-> short-circuit, ...). If you do not follow this precaution, we, the authors of this tutorial, cannot be held responsible for the damage that you might cause to your board.

Chapter 2

Description of ADC

The ADC is integrated on the FPGA, so we can directly connect our analog source to it. To do so, the ADC is equipped with a 2x5 pin header, 2 of them are for the voltage (5V) and ground, the other 8 are simply 8 different channels. The ADC exchanges 4 signals

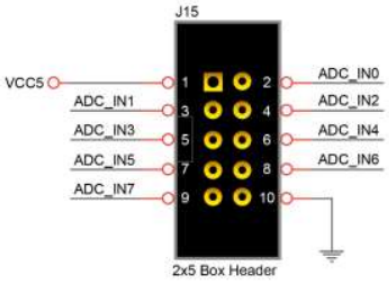

Figure 2.1: The 10 pins of the ADC (source: user manual)

with the FPGA itself: "DIN", "DOUT", "CS_N" and "SCLK". The driver of the card must handle some of them: "DIN", "CS_N" and "SCLK", and "DOUT" will be the result send by the ADC. "DIN" contain all of the configurations for the ADC:

-Voltage Range (2,5V or 5V)

-Coding (Binary or complement)

-Power Management

-Channel Selection

Other configurations are possible, but they are too advanced for this tutorial, so we won't go further in the details. Figure 2.2 displays the inputs and outputs of the ADC.

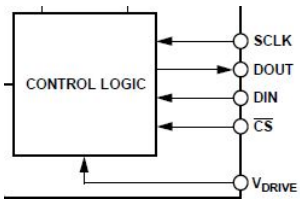

Figure 2.2: Functional diagram of the ADC (source: ADC datasheet)

Chapter 3

Driver of the ADC

3.1 Presentation of the driver

As said in the first part, the driver of the ADC has been acquired through the library *IP catalog* of the software Quartus. It is located in the file "adv_adc.v" (Verilog file). However, if you wish to acquire the driver by yourself, you need to go to *Tools -> IP catalog* and type "ADC" in the search bar. Double-click on the result that appears (only one will show) and a window will pop-up. Under the tab "Block symbol" you will see the representation of the ADC driver (figure 3.1). Please bear in in mind that our driver is slightly modified and thus, doesn't correspond exactly to what you will generate (modifications will be explained later in this tutorial).

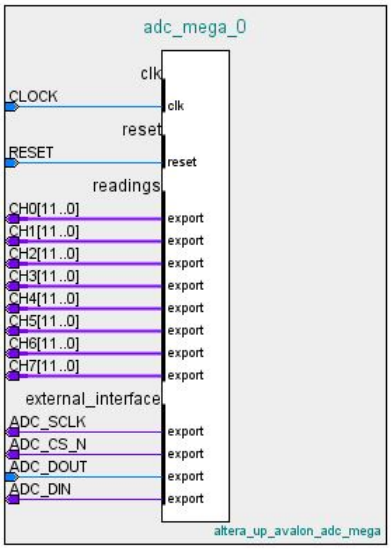

Figure 3.1: Symbolic representation of the ADC driver (with the inputs and outputs)

Figure 3.1 shows the inputs and the outputs of the ADC driver that will control the ADC on the FPGA board. "CLOCK" is the 50 MHz clock on the board and "ADC_SCLK" is the output of the driver and the input clock of the ADC (here, 12.5 MHz). The divison is done inside the driver. "RESET" is a simple reset button that will put back the driver in its initial state.

"ADC_CS_N" is the "not chip select" that goes to the ADC and command it to start the conversion of the analog signal to a digital one. The conversion process consists in creating the "DOUT" vector of 15 bits. Its representation can be seen on figure 3.2, where the first 3 bits are the address of the channel where the information is sent and the last 12 bits are the input signal. In the driver, "ADC_DOUT" is a 1-bit line that will

receive the elements of "DOUT" vector. It may appear confusing that "ADC_DOUT" is an input, but remember that it goes out of the ADC. Here, the name is given from the ADC point-of-view (not from the driver's point-of-view).

"ADC_DIN" is the 1-bit line that goes to the control register of the ADC, which is 12-bits long. This line sends bits in order to control the behaviour of the ADC by setting the values in the register. Again, the name is given according to the point-of-view of the ADC.

The 8 output channels of the driver are the readings on each channel of the ADC (from 0 to 7). These outputs don't go to the ADC.

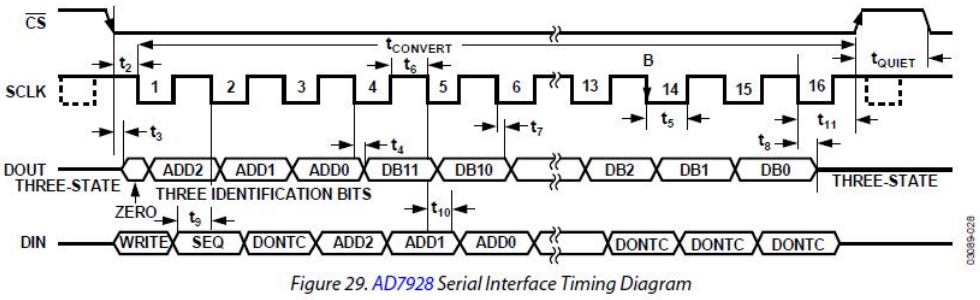

Figure 3.2: Conversion process inside the ADC (CS_N = 0) (source: ADC datasheet)

3.2 Modification of the original driver (from IP catalog)

The driver used in this project is located in the file "adv_adc.v" (a Verilog file). It has been slightly modified to allow us to implement our application. The original driver "sweeps" over each channel of the ADC (0-7). We wanted to implement an application that would allow the user to select the channel over which the ADC should send the data. We have added a variable "chan", that is passed as an input, and replaced "NUM_CH[2:0]" on lines 173 and 176 of "adv_adc.v" to select the channel that we want and not sweep over all channels.

3.3 How does the driver work ?

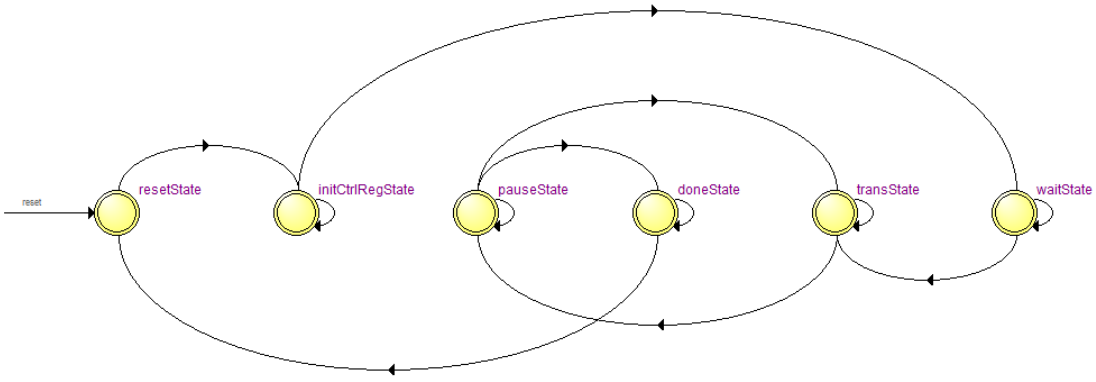

Figure 3.3: State machine of the driver

The driver's work can be explained by its state machine, shown in figure 3.3. The section in the code "adv_adc.v" related to the following explanation is "NextState Selection Logic".

When the RESET button is pressed, the driver is put into "resetState". In this state, CS_N (not chip select) is at 1 (which means that the ADC is not converting the analog signal to digital - no DOUT vector) and DONE is at 0 (the ADC has not "swept" all the channels). At the next positive clock edge (the one of 50 MHz), the driver will pass into "initCtrlRegState". In this state, CS_N is at 0 and the ADC peripheral starts to convert the analog signal (building the DOUT vector) and DONE is still at 0. If 15 clock counts of ADC_SCLK have passed (which means that DOUT vector is ready) and ADC_SCLK is at 0, then the driver will pass into "waitState". Otherwise, it stays in "initCtrlRegState".

In "waitState", CS_N goes back to 1 and DONE remains at 0. When the driver is in this state, it means that the ADC peripheral is ready to send a DOUT vector on a specific channel (3 bits of address and 12 bits of digital signal), bit by bit to "ADC_DOUT" of the driver. If GO is at 1 (ready to send the bits of DOUT to the driver), the next state is "transState". In this state, the ADC peripheral transmits the bits of the DOUT vector to "ADC_DOUT" of the driver. CS_N and DONE are both at 0, meaning that a new conversion process can start (for a new incoming signal). After 15 clock counts of ADC_SCLK, all the 15 bits have been sent to the driver and the new incoming signal has been converted to digital values. The next state is "pauseState".

The "pauseState" (CS_N = 1, DONE = 0) is important for the ADC, because there is a time condition, called quiet time, between each conversion (from analog to digital) that must be respected. The minimum quiet time should be 50 ns. As long as this condition is not fullfilled, the driver remains in the "pauseState". This condition only concerns conversions and not transmission of the converted values to the driver. After this delay, the next state will be "transState" again but a different channel is used.

When all channels have been "swept" (the ADC has sent DOUT on channel 0 upto channel 7), the driver goes into "doneState" (CS_N = 1, DONE = 1). As long as GO equals 1, the driver remains in this state. When GO equals 0, the driver goes back to the first state "resetState". From there, it will begin the whole process again.

3.4 Testbench and simulations

The following section will present the results of the testbench "adv_adc_TB", which aims to show the normal functioning of the driver. The testbench will simulate a conversion process by modifying the bits of DOUT at each SCLK count (clock of the ADC). The bits of DOUT are modified randomly. We have also simulated channel change to see if our modification of the driver is taken into account. To perform the simulation, you must compile the files "adv_adc.v" and "adv_adc_TB.vhd" and then run "modelsim".

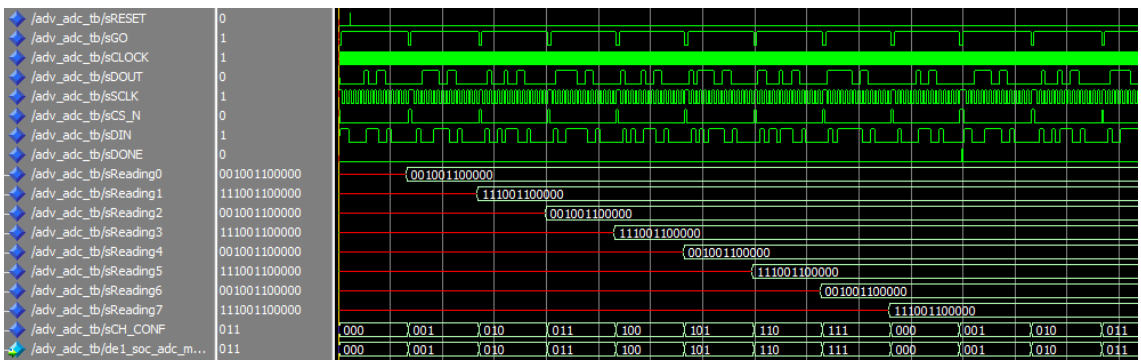

Figure 3.4: Simulation of the driver

The first thing we can check on figure 3.4 is that the signal "sDOUT", associated with "DOUT", is randomly modified (as wanted). We can also check on the signal "sCH_CONF", associated with "chan" (our channel selection variable), that the address of the channel is changing.

We can notice that the change of address is not considered immediately and that the driver waits for the next CS_N = 0 to consider the new value of "sCH_CONF" to change channel. This is normal, because the initial state of the driver is "resetState" and no values of channel have been assigned yet. At the following CS_N = 0, the driver takes the previous channel address (ex: 000) and puts "sDOUT" on it.

We can also notice that while CS_N = 0, GO = 1. It means that the ADC sends the actual values of DOUT on a channel while processing the new incoming signal. This confirms the behaviour explained in the section "How does the driver work ?".

In addition, the selected channel is in accordance with reading channel (000 -> Reading0, 001 -> Reading1, etc ...).

Finally, we can see that when all the channels have been swept, DONE = 1. It goes back to 0 when the process starts over again (starting with address 000).

Chapter 4

Application of the driver

4.1 Presentation of the application

Now that we have modified the driver to act like we want it, we need to control our variables and to connect all of them with the actual physical components. The file associated with our application is "Bouton.vhd". Our FPGA is equipped with 4 buttons and 10 switches. We decided to have a button to activate the "Reset", and to use 3 switches for the channel selection and one for the "Go". A LED is associated with each switch, so we decided to turn them on when the switch is on. Switches 9 to 7 are used for the channel selection ("1" when switch on and "0" when switch off, so the range is from 111 to 000). To further improve the users' experience, we have decided to use one of the 7-segment display to show the channel number (on the far left), allowing the user to bypass the binary conversion. We decided to place the "Reset" on Key 0, and to light the LED 1 when this button is pressed, for visual reasons.

The programming of the 7-segment is found in the user manual of the FPGA board. There is a total of 6 displays on the board. Their ID is HEXn (where n ranges from 0 to 5). We use HEX5 to display the number of the channel. The order of the bits in the vector HEX5 and the corresponding segment are shown on figure 4.1. The file "Bouton.vhd" has a process "Hex" where the 7-segment display is modified. It is important to point out that for the 7-segment display, bit "0" means light ON and bit "1" means light OFF.

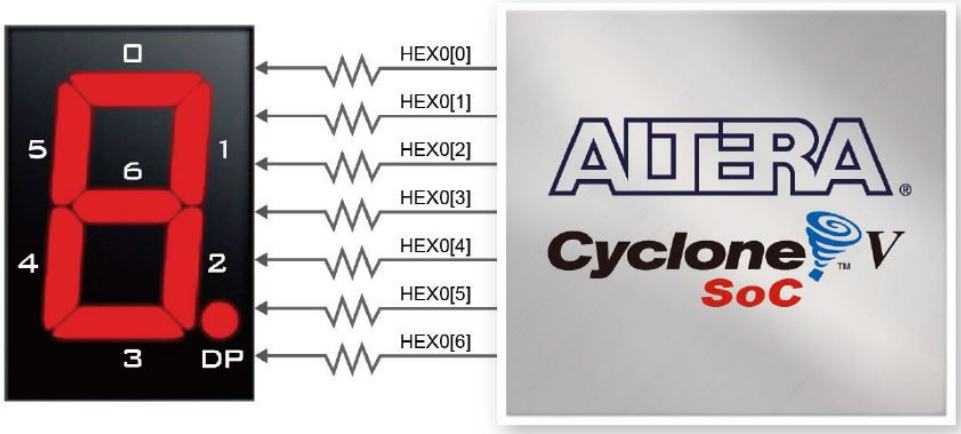

Figure 4.1: 7-segment display on the FPGA board (source : user manual)

4.2 Testbenches simulations

The first thing we need to check on this simulation is that the physical component of the ADC and the FPGA and our signals are reacting the same way. To facilitate the lecture, they are each side by side. We can see that all the buttons and switches have the same waveform as their physical counterpart, so it simply means that our code connected them just as we wished. To perform the simulation, you must compile the files "adv_adc.v", "Bouton.vhd" and "TB_final.vhd" and then run "modelsim".

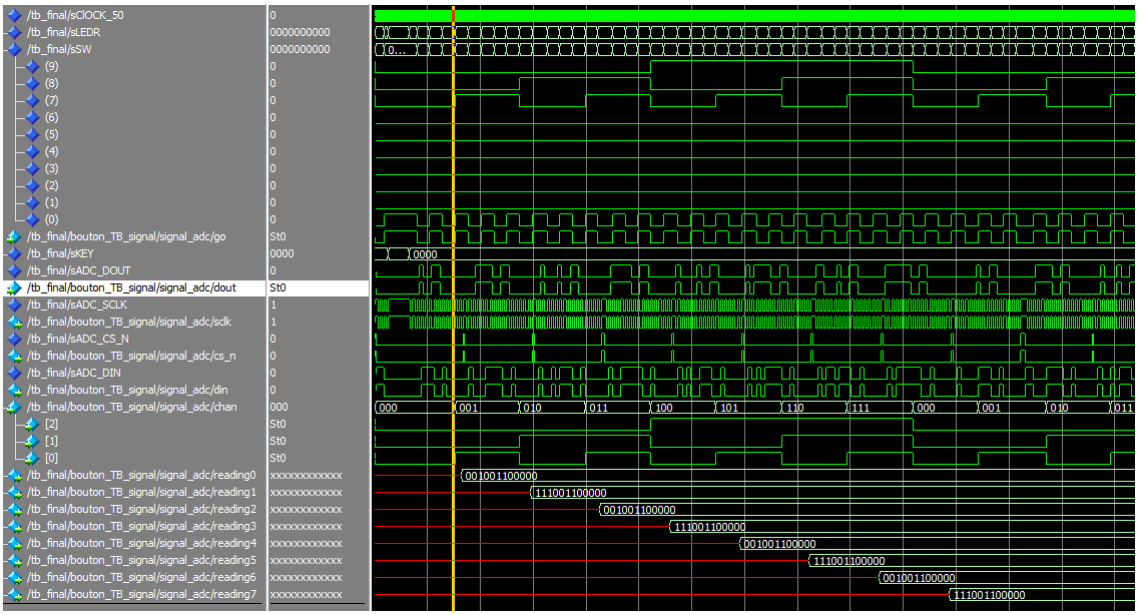

Figure 4.2: Simulation of the application

The very first thing we want to check with the simulation is that the switches work and change the channel. On figure 4.2, the signal "sSW", associated with the switches, shows that the values of "sSW"(9 to 7) is in accordance with "chan". This means that the switches properly change the channel, just like we wanted. We also find the same behaviour as in the previous simulation (the channel change occurs on the next "CS_N = 0") and confirm that the channel and the "readings" are coherent (000->reading0, 001->reading1, ...). We can also verify that "GO" is properly activated by "sSW"(0) and is in accordance with the signal "go".

For convenience reasons, we have put the simulation of the LEDs on the top of the switches in another figure (4.3). It shows us that LEDs (signal "sLEDR") turn ON at the same time as the switches are at "1". This proves that LEDs and switches are linked together and work properly.

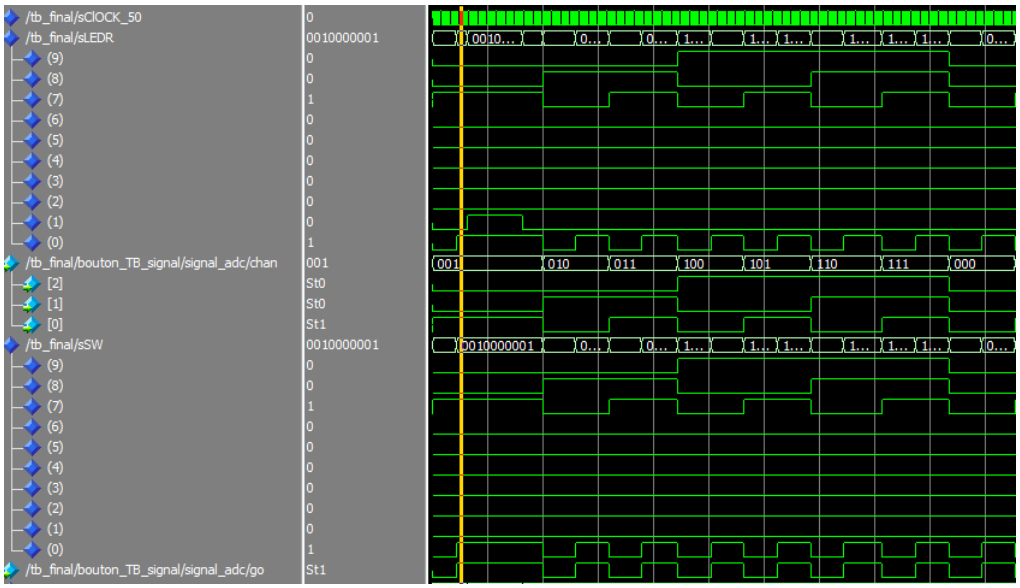

Figure 4.3: Simulation of the application (LEDs)

Chapter 5

Testing the application on the board

5.1 Flashing the files on the board

After finishing and checking the simulations, you can put the driver and the application on the FPGA board. To do so, we advise that you first remove the testbenches from the project and that you only keep "adv_adc.v" and "Bouton.vhd". When you have done that, re-compile the project.

When the compilation is done, you can take the Altera FPGA board, plug in the AC adapter into power and connect the USB cable from the computer to the "USB Blaster II" connector. Do not forget to press the red button on the side to turn the board ON. With Altera Quartus II, the drivers of the FPGA are already installed and your computer should have recognised the board (the board is recognised only if it is turned ON). The next step is to flash the .sof file on your board. SOF stands for "SRAM Object File" and it is a binary file that comes from the compilation of the driver and the application. You can usually find this file in the folder "output_files".

The flashing procedure is rather simple. Once the board has been connected to your computer and turned ON, go in the tab *Tools -> Programmer*. This will open a new window.

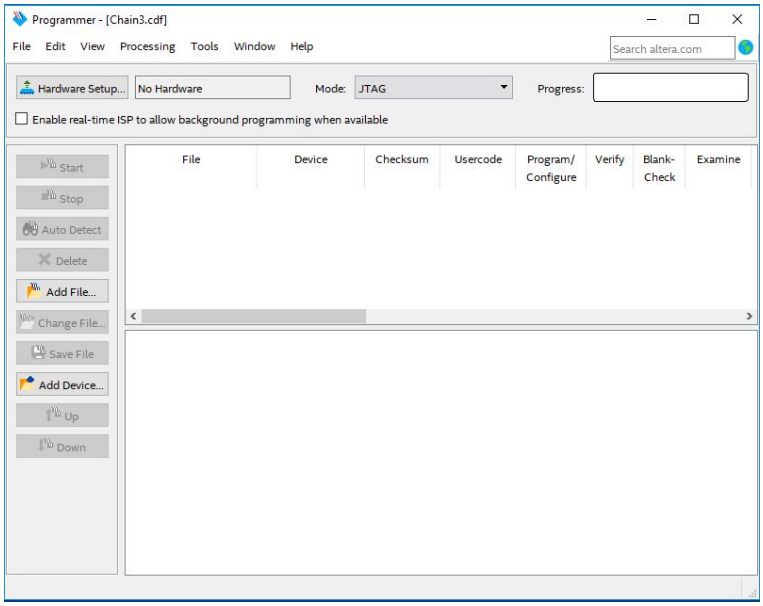

Figure 5.1: Blank windows of Programmer

It might be possible that your hardware has not been detected by Quartus (figure 5.1). To add your board, click on the tab "Hardware Setup..." and another window should appear (figure 5.2).

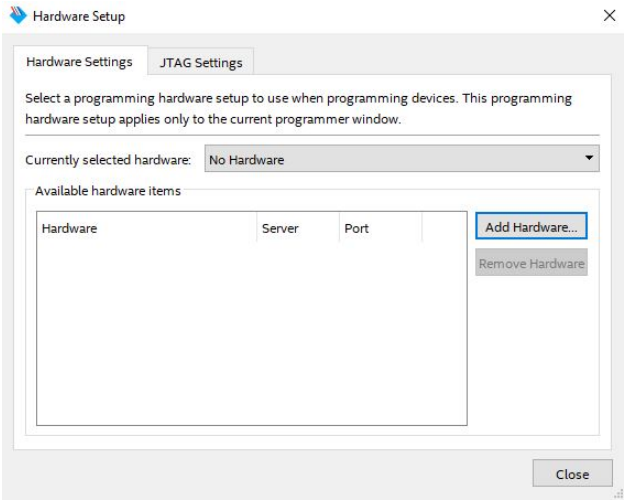

Figure 5.2: Adding the board to Programmer

Click on "No Hardware" on the right of "Currently selected hardware" and select "DE-SoC [USB-x]" (x is a number that might be different for you). Then, click on "Add Hardware ..." and then close this window. You should now have the board recognised properly. Be sure to select mode "JTAG". After this, click on "Auto Detect" tab with the binoculars symbol. The program will ask you to select the device (in our case, it is 5CSEMA5). Check on your board's Altera chip the device model to avoid issues. You should obtain a window similar to figure 5.3, with two elements in the chain.

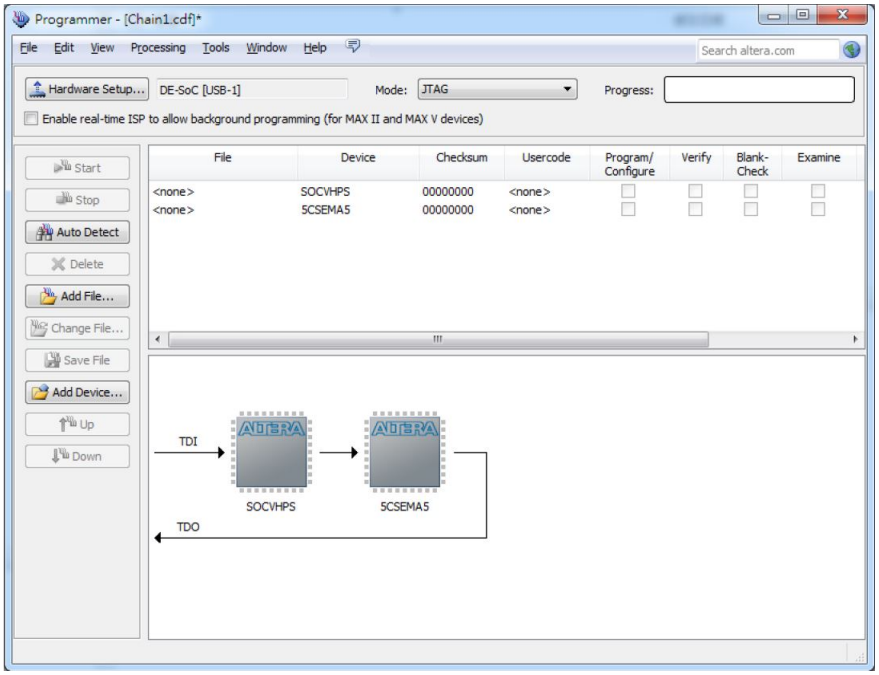

Figure 5.3: Chain of components (source : user manual)

Now, you have to flash the .sof file to the device 5CSEMA5. As shown on figure 5.4, right-click on the device 5CSEMA5 and select "Change File". This will prompt a new window that will ask for the .sof file. It is located in your project directory, inside the

folder "output_files". The name of this file is the same as your project (in our case, it is ADC_DE-SoC.sof). Select the file and click on "Open".

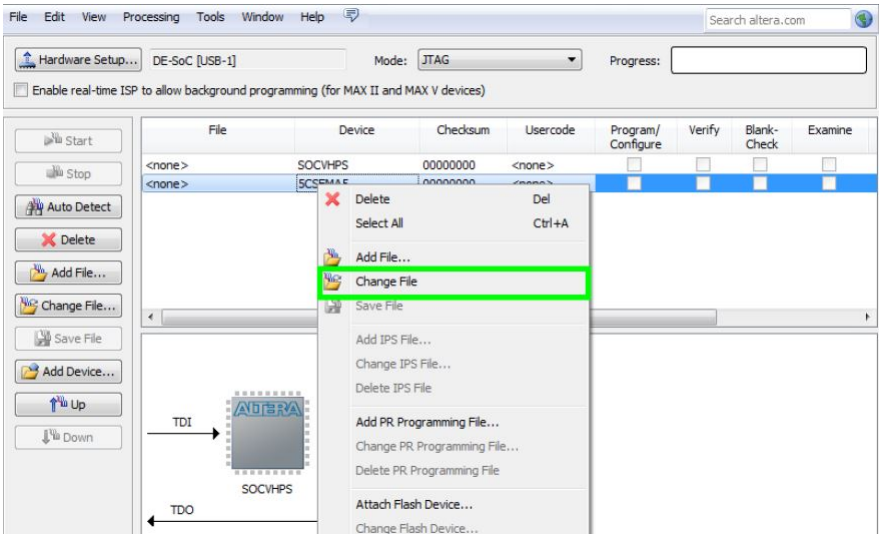

Figure 5.4: Adding .sof file to Programmer (source : user manual)

You are now ready to flash the FPGA board. Make sure that your chain has only 2 elements (no more than 2 and no duplicates are allowed !) and that your .sof file is correctly selected for the device 5CSEMA5. You can then tick the box "Program/Configure" for the device 5CSEMA5 (don't do it for the other one !) and click on "Start" (figure 5.5). The progress bar at the top will indicate when the process is complete.

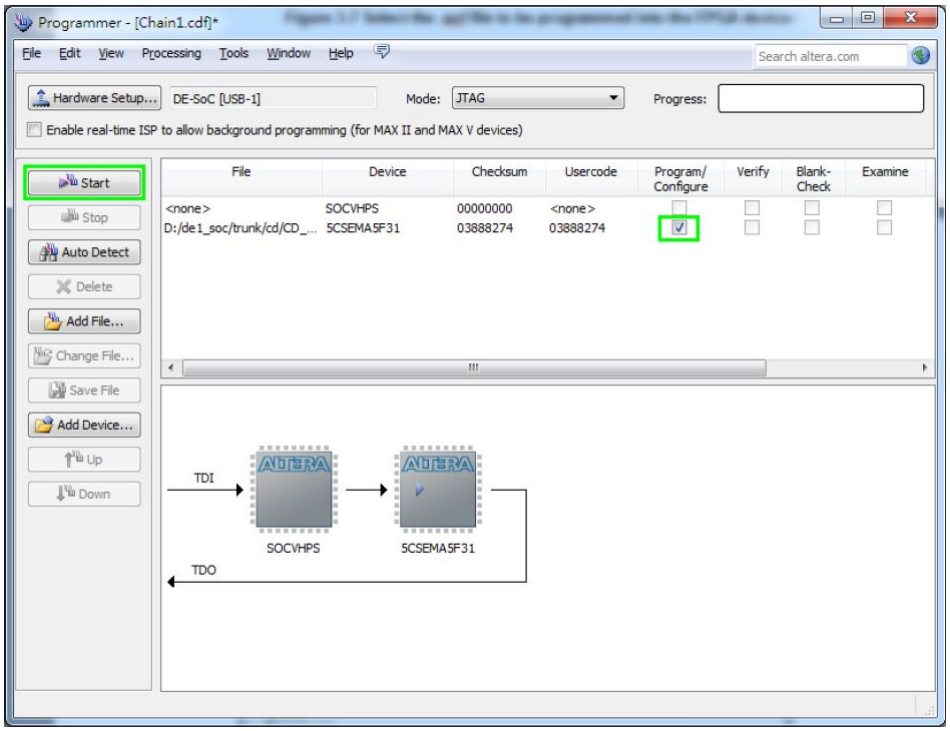

Figure 5.5: Ready to flash the FPGA board (source : user manual)

5.2 Practical demonstration

Depending on the initial position of the switches on your board, you will see a number from 0 to 7 on the far left 7-segment display. Use the first 3 switches on the left to select the channel where you want the ADC to send data. You will also see that the LEDs at the top of the switches will light up when the switches are up. Switch 0 (on the far right) is for the "GO" and the far right button (labelled "KEY0") is the reset. When pressing this button, the LED1 will light up. You can check our practical demonstration in the following video : `https://youtu.be/aX6UkGxf2ac`

Conclusion

This tutorial has presented a driver and an application for the ADC of the FPGA Altera DE1-SoC (rev. D). We have explained the theoretical background of each component to help you understand how they work. We have presented simulations to show you what results you should expect when you will run test by yourself. Finally, we have showed our practical implementation in a short video

We remind you that all of the explanations here come from our personal knowledge, experiments and from .PDF files you can find on the Github repo (user manual and datasheet of ADC).