Final project

Eletronic Systems - SELE
Marco Ramos, up201504735
Rafael Almeida, up201503139
Faculty of Sciences of the University of Porto (FCUP) / Faculty of Engineering of the University of Porto (FEUP)
15th of January of 2019

I. Introduction

In this report, we explain how the two tasks from this final project were accomplished, while also detailing the different types of communication and techniques used for this effect.

The first task of this final project consisted of interacting with a ChipKIT Uno32 board (PIC32MX320F128H microcontroller) using only its JTAG port. We interact with the port using an Arduino Uno Board which then communicates through UART with the computer where we insert commands corresponding to the different implemented interactions: getting the ID code of the microcontroller, turning on the internal led LD5, turning the led off and getting the state of a button connected to one of Chipkit's pins.

For the second task, we will establish a two-wire/I2C communication with a TMP175 temperature sensing device and an Arduino Uno board, which will also communicate through UART with the computer and with a python script running in this computer, it was possible to plot the temperature in real time with a resolution of 0,0625 ^{o}C. We also designed a board with ATmega328p MCU (Arduino Uno's MCU) and with the TMP175 using EasyEDA, which could then be manufactured and run the produced firmware.

II. Task 1

A. PIC32's JTAG port

The JTAG port for boundary scan testing consists of 4 pins: TMS (Test Mode select), TDI (Test Data Input), TDO (Test Data Output) and TCK (Test Clock). The location of these pins on the board can be found consulting the board's reference manual [2].

We define the Arduino's pins connected to TMS, TDI and TCK as outputs and the pin connected to the TDO as a digital input.

Communication with the board is done exclusively through these pins.

Setting the TMS pin high or low and completing clock cycles (setting the clock high, waiting 1ms and then low, waiting 1ms again), enables us to navigate the TAP states, which can be observed in the fig. 1.

The BST infrastructure is controlled by shifting in instructions to the instruction register (IR) which then controls what the data register (DR) consists of.

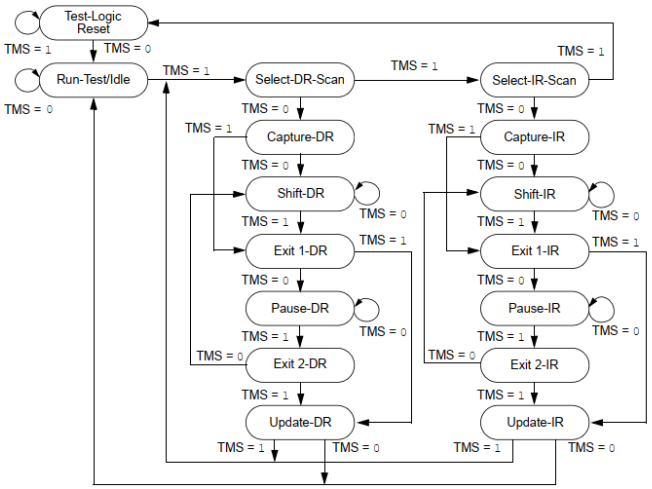

Fig. 1. TAP controller state diagram.

The instructions and test vectors are introduced into the TDI pin and we read the values shifted out by the boundary scan chain in the TDO.

Information on the different instructions available, the size of the instruction register and the various data registers are found on the microcontroller's BSDL (Boundary-Scan Description Language) file, which is usually made available by the MCU's manufacturer, as was indeed the case for this one. It can be found in reference[3].

B. The general method and getting the ID code

The instructions consist of 5 bits and the instruction for the ID code to be made available, in the data register, is (00001), as specified in the BSDL file.

We developed simple functions to facilitate the navigation of the TAP states: a function for completing the cycle of the clock as described above (we set the TCK output pin as high, delay 1ms, set the pin low, delay another 1ms), a function for completing "x" cycles with the TMS set high and another for completing "x" cycles with the TMS set low.

To send an instruction, we can first make sure we know in which state we're located initially, by doing a soft reset: we set TMS high and complete 5 clock cycles. Regardless of the initial point, this takes us to the "Test-Logic Reset" state.

So, after a soft reset, we set TMS=0 and complete 1 clock cycle (getting us to the "Run-Test/Idle" state), then set TMS=1 and complete 2 clock cycles (getting us to the "Select-IR Scan"

state) and finally set TMS=0 and complete another clock cycle (arriving at the "Shifting-IR" state).

The LSB (1) is the first to be shifted. For this, we set the TDI high, according to the wanted instruction (00001), read the value present in the TDO by storing it in a variable which we then print in the computer, and then, with TMS still set low, complete a clock cycle, successfully shifting in the first value for the instruction and returning to the "Shift-IR" state. The process is repeated for the next 3 values. The last one is shifted by instead setting TMS high before completing the clock cycle, which leaves us in the "Exit 1-IR" state.

For getting the ID itself, we proceed to the "Shift-DR" state through the same process (set TMS to the adequate value and complete clock cycles to change TAP state).

This time, we have to shift 31 values before exiting the "Shift-DR" state, as the ID code consists of 32 bits.

This time, we don't worry about the values of TDI and instead only store the TDO values in 32 bit variable, which is then printed and correctly contains the ID code of the MCU.

C. Turning the internal led LD5 on and off

For setting values in the pins of the MCU, we give the "EXTEST" instruction (00110). This makes the data register consist of the boundary scan cells of the MCU.

The instruction and the number of cells present in the register, is also information present in the BSDL file.

Since there are 148 cells, we created a 19-element array of bytes (152 bits in total) for storing the 148 values which are shifted out and also to store the values we want to shift in.

Following the procedure described in the previous section, we must only know in what position the led lies. The BSDL file has the 148 cells labeled from 0 to 147. Consulting the Chipkit's reference manual, we find out the led LD5 is the pin labeled "RF0". There are 3 cells corresponding to this pin. One for its input, n=18, another for its output, n=19 and another labeled "control", n=20, for setting the wanted mode.

Since we want to control the state of the led, which is fed by the output of the cell, we control it by setting the output cell in n=19 to the wanted state (high for turning the led on and low for turning it off) and setting the control bit (n=20) high.

The rest of the values in the 148 bit test vector array are simply 0.

The first value to be shifted is that corresponding to n=0, so, to turn on the led, we set TDI high before the 20th and 21st shifts (n=19 and n=20) and for turning it off, we only set the TDI high before the 21st shift, while for the rest of the shifts it's set low.

D. Getting the state of the button

The button is connected to pin 29 of the ChipKit board. This corresponds to pin 63 of the PIC32, labeled "RE3". Consulting the BSDL file again, this pin also has 3 corresponding cells: for its input, output and control, n=3, n=4 and n=5, respectively.

We again, give the instruction for "EXTEST" and proceed to shift 148 values of the data register, this time not worrying about what we set in the TDI (all zeros) and simply storing the values read in the TDO.

Knowing the state of the button consists of knowing the input of pin. Since this input information, corresponds to n=3, we know this will be the 4th value read in the TDO (after 3 shifts).

E. Important remarks

Perhaps the most noteworthy aspects of this project, are ones which one might easily disregard as details, but these were indeed the biggest obstacles to completing the task with ease, since they are mere details, they are sometimes disregarded in informative texts about general concepts of boundary scan testing. Sometimes these are disregarded in the text itself and other times disregarded by users which are simply looking for an introductory outlook of the subject.

These were the following:

- The shifting of the registers (both IR and DR) occur upon *exiting* the "Shift" state. This is, for shifting "n" bits, we enter the state an initial time, then come back to it "n-1" times as the shifting occurs and the last shift occurs when we leave onto the "Exit-1" state.
- For shifting in a value, we set the TDI to the wanted state and *then* proceed to shift the register by exiting the "Shift" state. One could think, that the first value which is "shifted out" is only read in the TDO *after* shifting, as we did, however, this first value which is actually never "shifted out" as it is actually available to read directly before shifting. When the first shift is done, this first output value is lost and the value made available in the TDO is actually the second value. This means the TDI is set and the TDO is read both before completing a clock cycle. This was detected since we repeatedly got the wrong instruction capture (the value which is shifted out upon shifting in an instruction in the IR-register): we were skipping the first value (1), since we thought it had to be shifted out, and obtained only 0's. The error was found when we analysed the signals and found TDO to be set high, before the shifting occurs.
- Picturing the boundary scan chain starting at the TDI and ending in the TDO, one can picture the pin labeled as "0" in the BSDL file is the first, following the TDI and the "147" pin will be the last one, being read by the TDO. This would mean to shift a test vector that sets pin 147 as "y" and pin 0 as "x", the value "y" is the first to be shifted and "x" is the last. However, upon failing to turn on the led, we found the opposite to be true. The last value to be shifted is the value wanted for pin 147, which means the order in which we shift coincides with the number of the pins (0 is shifted first, followed by 1, etc.). If we consider the test vector to be a 148 number, the first to be shifted (n=0) must be the LSB, which coincides with what happens for the instructions, but opposes what one pictures the boundary scan chain and counts from left to right.

III. Task 2

A. Controlling the TMP175 sensor

The TMP175 device is a temperature sensor. Programming it from scratch, to make it output a value of the temperature is relativity easy, due to its data sheet[1] clarity.

The pin configuration of this sensor can be observed in the following figure. The SDA and SCL pins are the standard I2C protocol communication pins, the V+ is the input power voltage, which needs to be 3.3 Volts, GND is the ground, in this case shared with the Arduino. Finally, the A0, A1 and A2 pins will be used to setup the address for the TMP175. Also the ALERT pin is for extra functions of this sensor, like for example, getting a signal when the sensor is at a higher temperature as the define as T_{high}, but this feature will not be used in this work.

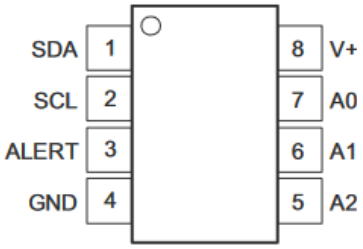

Fig. 2. Pin configuration of the TMP175[1].

The frequency we chose for the SCL signal is 100 kHz, which lies between the lower and upper limit of the normal mode for this sensor, 0.001 to 0.4 MHz (1-400 kHz).

The address we chose with A0, A1 and A2 pins was the easiest configuration to setup, connecting every pin to GND (A0 = A1 = A2 = 0). Making the TMP175 slave adress 1001000 (0x48) with 7 bits, due to I2C communication being done with 8 data bits, the 8th bit will be a Read or Write bit if it is 1 or 0, respectively. This makes the address byte that we will need to send before write instructions 10010000 (0x90) and the address byte for reading 10010001 (0x91).

To communicate with the sensor and make it output its current temperature, we first need to configure it for this effect. This is done by sending into the TMP175, an address byte to write (0x90), then we send a byte for altering the pointer register byte to the configuration register which is 00000001 (0x01). This means the last two bits on this pointer become 01, these are the P1 and P0 bits of the register's pointer byte, which can be observed in the next figure.

P7	P6	P5	P4	P3	P2	P1	P0
0	0	0	0	0	0	Register Bits	

P1	P0	TYPE	REGISTER
0	0	R only, default	Temperature register
0	1	R/W	Configuration register
1	0	R/W	T_{LOW} register
1	1	R/W	T_{HIGH} register

Fig. 3. Pointer Register Byte and the possible values for P1 and P0 bits[1].

Right after sending that byte, we write the configuration byte which is what the sensor is expecting. In this case we want to configure the sensor to output the temperature with 12 bits, 8 integers plus 4 bits for decimals, making it precise to 0,0625 ^{o}C. For this, we need to send the configuration register byte. The configuration register byte has 8 bits for different functions. Because we want to simply extract the value of temperature, we will only use 2 bits out of this byte, resulting on the rest of the bits to be 0, due to the rest of those configuration bits, not being necessary to this work.

D7	D6	D5	D4	D3	D2	D1	D0
OS	R1	R0	F1	F0	POL	TM	SD

Fig. 4. Configuration Register Byte[1].

The only bits that will have any values, will be the R1 and R0, which in this case, we will want to have 12 bits of resolution, or in other words, 0.0625 ^{o}C of precision. By utilizing the best resolution, we make a slight compromise on the speed of operation through an increase in the conversion time, which is 220 ms on average for 12 bits resolution. This time, will be dealt by adding a delay of 250 ms on the loop section of the code for the sensor, on the Arduino.

R1	R0	RESOLUTION	CONVERSION TIME (Typical)
0	0	9 bits (0.5°C)	27.5 ms
0	1	10 bits (0.25°C)	55 ms
1	0	11 bits (0.125°C)	110 ms
1	1	12 bits (0.0625°C)	220 ms

Fig. 5. Configuration Register Byte[1].

After this configuration, we will need to reconfigure the sensor back to the temperature register, which makes it able to output its temperature value and receive an input of "Read". The output of the temperature register will be done with 2 bytes, the first one will have the most significant bits and will have the integers values. The second byte will have the four most significant bits equal to the decimal values and the last four always equal to 0.

So, to get a temperature read, we will need to configure back the pointer to the temperature register, sending first a Write address byte (0x90) and right after the temperature register byte (0x00). After this procedure the sensor is ready to output temperature values with 12 bits resolution.

After this reconfiguration, we will only need to send a Read address byte and read 2 bytes right after to get the full 12 bits of temperature information. The next figure illustrates this process very clearly.

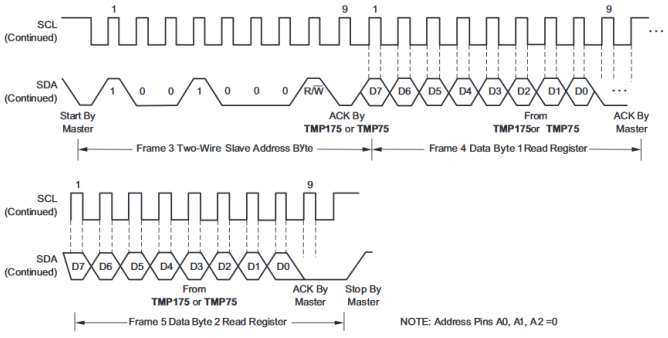

Fig. 6. Read format for the TMP175 sensor[1].

B. Acquiring temperature information with a computer connected to the Arduino

After setting up the Arduino to receive temperature readings, we wanted to receive these temperature bytes on the computer through a USB connection with the Arduino. We tried early on to use PuTTY, but this software does not have friendly user interfaces so we switched to Python. This change was facilitated by the ease of access to received data through serial ports with Python and also because of the amazing plot and numerical libraries available like Matplotlib and Numpy.

The code we did for this had in first place a short section to start and open the serial connection with the Arduino, with UART at 9600 baud rate, using Serial library. After that part we have an output decoding function (decode(A1), A1 is the 4 bits we read from the Arduino), because the data came as hexadecimal bits, we need to convert the temperature bits to decimal values. The third part of the code is a function (plotsetup(M,N)) where we load values into a list, to make the first plot of this process. After loading all of these data we are ready to update the plot, that's where the fourth part of the code is needed, this fourth step is the last function we will need, this function (plotupdate(M,N)), updates the plot with N new values on a plot with M points.

In resume, for a temperature read to start, we just need to put values of M and N into the last line of code, "plotupdate(M,N)" and run the code. This will result, on a updating plot with a total time range of about 0.25 M seconds, for example, if we choose M=200 we will get 50 seconds of time range and with a N=5 we will update every 5 new points, which will be every 1.25 seconds. Also the decrease in the N value will make the plot time less precise, this is due to the time it will take to refresh each plot (about 3 ms) and its recommend to use and N equal or higher than 5.

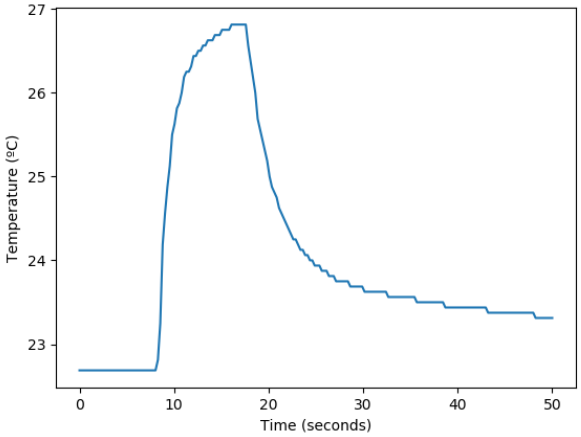

Fig. 7. Example of temperature reading plot with a 50 seconds total time.

The example we can observe in this last figure is a reading with (M,N) = (200,5), resulting in the total time for the plot being 50 seconds. This plot shows in the beginning, a constant temperature reading and after some seconds, we place our finger above the sensor increasing its temperature and thus the value read. Around 10 seconds later, we take off the finger from the sensor and see the temperature getting lower. This behaviour we observed, has an approximately exponential dependence with time and temperature difference from the sensor's temperature versus the surrounding ambient temperature, which is precisely what Newton's law of cooling states.

References

[1] http://www.ti.com/product/TMP175
[2] http://ww1.microchip.com/downloads/en/DeviceDoc/chipKIT-Uno32-RevC_rm.pdf
[3] http://ww1.microchip.com/downloads/en/DeviceDoc/PIC32MX320F128H.bsdl