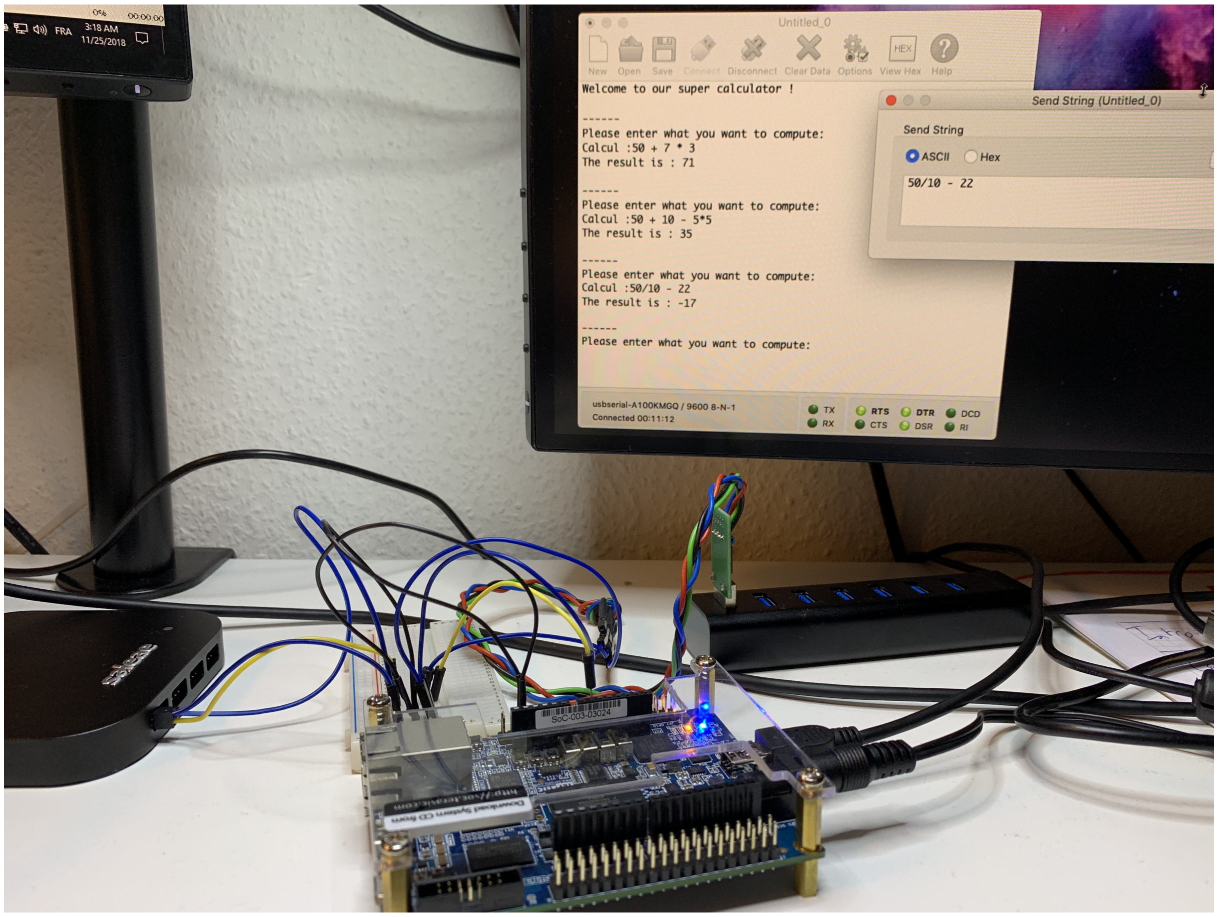
Laboratory 2 - Embedded Systems

Maxime Marchionno, Nicolas Peslerbe

November 26, 2018



```
Welcome to our super calculator !

------
Please enter what you want to compute:
Calcul :50 + 7 * 3
The result is : 71

------
Please enter what you want to compute:
Calcul :50 + 10 - 5*5
The result is : 35

------
Please enter what you want to compute:
Calcul :50/10 - 22
The result is : -17

------
Please enter what you want to compute:
```

Send String (Untitled_0)

Send String

[✓ ASCII] Hex

```
50/10 - 22
```

usbserial-A100KMGQ / 9600 8-N-1
Connected 00:11:12

Contents

Abstract

In this document, our goal is to report about the second Embedded Systems laboratory. The goal of this laboratory was to implement a programmable interface from scratch, including software and hardware parts.

1 Introduction

First of all, we decided to choose the UART programmable interface as goal for this laboratory. Following a very rigorous way to do, which includes a non negligible part of the work on test benches creation to check modules, we were able, step-by-step, to build a fully working UART programmable interface. In the following sections we will first report about the system working principle and after about some technical details, finally, we will provide some analysis. This system was invented without picking up any idea from existing ones, only some general documentation (Mainly the one from the laboratory document) was read. It probably differs a lot from existing UART interfaces. For the laboratory, we used a DE0-Nano-SoC development board which uses a Cyclone V FPGA, the programmable interface was added to a NIOS II processor using Avalon bus.

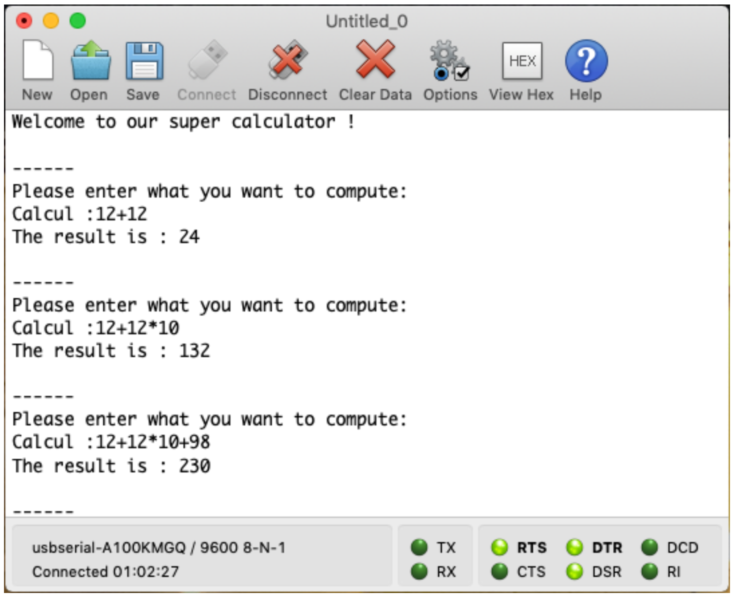
2 Main elements

2.1 The calculator

One of the part of our work was to create a relatively basic calculator. It's quite independent from the UART because we implemented the UART interface with drivers (uartDrivers.h) which allows to read or write on the uart using some predefined methods. The calculator working principle is the following one :

- Wait to have some data available
- Read received data from UART
- Print the calculation in the Nios console
- Decompose the received data to identify number and signs
- Perform a first computation of all divisions / multiplications from left to right
- Perform a second computation of all additions / substractions from left to right
- Write the calculation result over UART and print it in the Nios console

As you read, our calculator can support multiple in line calculus from the distant device and perform it respecting priorities (it unfortunately doesn't support parenthesis or floating point numbers).



```
Welcome to our super calculator !

------
Please enter what you want to compute:
Calcul :12+12
The result is : 24

------
Please enter what you want to compute:
Calcul :12+12*10
The result is : 132

------
Please enter what you want to compute:
Calcul :12+12*10+98
The result is : 230

------
```

usbserial-A100KMGQ / 9600 8-N-1
Connected 01:02:27

Figure 1: Nios calculator from a distant point of view (UART connected computer)

2.2 The UART interface

As we said before, some generic drivers was implemented to use the UART programmable interface, it means that this module can be reuse in order to perform other tasks. We defined the following functions as interface with our programmable interface:

```
int readUart(char * values, int numberMaxToRead);
```

Read data currently in the UART buffer and in reception. Two conditions can stop reading:
- The buffer becomes empty and no data have been received for some ms.
- The `numberMaxToRead` limit was reached.

The function returns the number of bytes read.

```c
void writeUart(char* toWrite, int numberOfCharacters);
int writeString(char* toWrite);
```

Write some data in the write buffer of the programmable interface. The data will be send immediately if the buffer is empty or after other data reaming in it. You can call the `writeUART` function precising the number of bytes to write or the `writeString` method if your string contains the '\0' character to precise it's end.

```c
int canRead();
```

Return one if the reception buffer of the UART connection has some data in it, 0 otherwise.

```c
int setupUart(int wordLength, int rate, int parity);
```

Method to call to set up the UART connection, `wordLengh` is unfortunately a not available option because it needed a lot of changes in the implementation to be able to work. `Rate` allows you to change the send/receive bitrate with UART standards and `parity` to enable the parity bit. By default (if you don't call the setup function), settings are : 9600 bit per second, 8 bits words ans no parity ending bit.

3 Implementation details

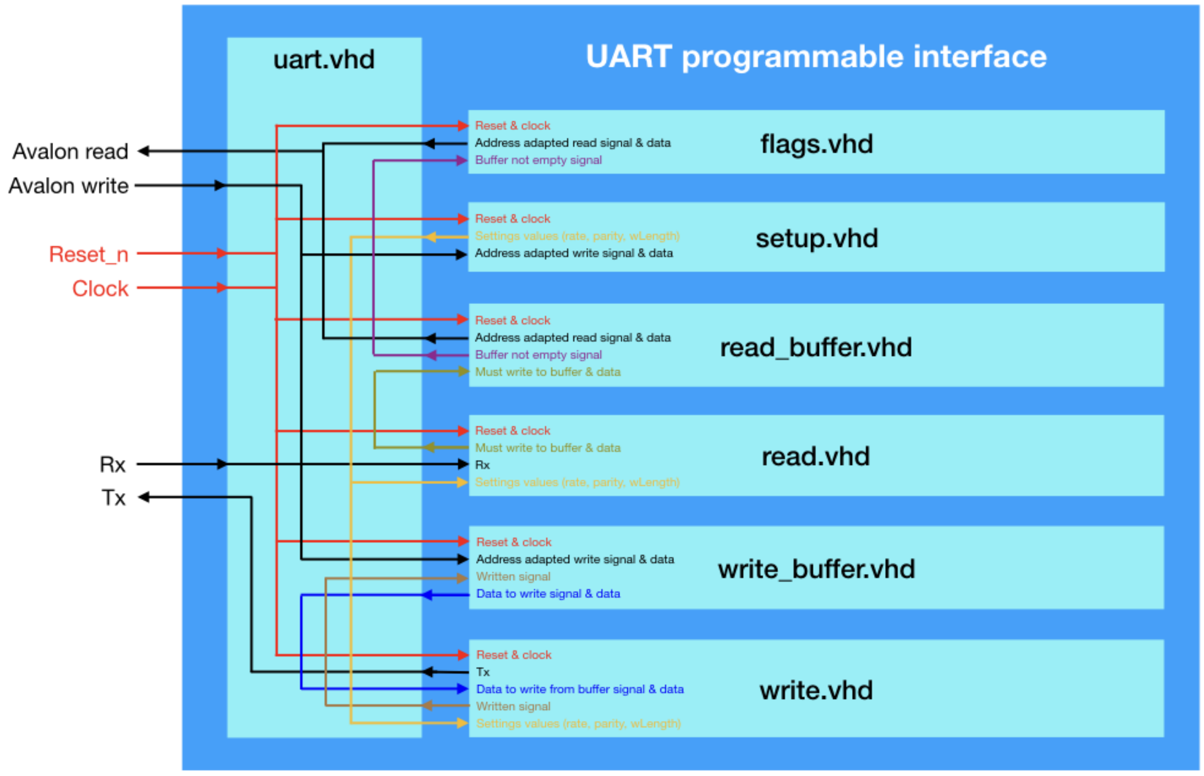

uart.vhd
UART programmable interface
Avalon read
Avalon write
Reset_n
Clock
Rx
Tx
Reset & clock
Address adapted read signal & data
Buffer not empty signal
flags.vhd
Reset & clock
Settings values (rate, parity, wLength)
Address adapted write signal & data
setup.vhd
Reset & clock
Address adapted read signal & data
Buffer not empty signal
Must write to buffer & data
read_buffer.vhd
Reset & clock
Must write to buffer & data
Rx
Settings values (rate, parity, wLength)
read.vhd
Reset & clock
Address adapted write signal & data
Written signal
Data to write signal & data
write_buffer.vhd
Reset & clock
Tx
Data to write from buffer signal & data
Written signal
Settings values (rate, parity, wLength)
write.vhd

Figure 2: Scheme of the uart programmable interface with submodules

3.1 General structure

Our interface needs only 4 distinct addresses to perform all its tasks, which means that we use 2 bits for addressing on the Avalon bus. The avalon bus adresses are shifted by two bits because data are coded on 32 bits words with an address for each byte, and we decided to use one different word for each functionality. The addresses mapping to communicate with UART PI the following one:

- 1st address (BASE + 0) for setup write
- 2nd address (BASE + 4) to read data from read buffer
- 3rd address (BASE + 8) to write data in the write buffer
- 4th address (BASE + 12) to read UART module state flags

As you can see on figure 2, the UART module is divided in many subparts. Using the read/write signals and the address, the `uart.vhd` module send an adapted read/write signal to submodules.

3.2 Flags

The `flags.vhd` submodule have been create to get some information about the UART interface. The only task it perform currently is to return 1 if the read buffer contains unread data, 0 otherwise. We could imagine in a future version to get information about the number of data currently buffered for example.

3.3 Setup

The `setup.vhd` submodule is a small 8 bits memory we can update at every moment writing on the corresponding address, it is used each time we re-initialize counter values of UART read/write functions (to define bit duration), or to add parity bit to a written byte for example.

3.4 Circular buffers

The circular buffers role are to buffer data coming from the program (using the `writeUart` function) or from the UART Rx port, so there is two buffers in our programmable interface, one for read data and the other one for write, each has an internal memory of 1024 bytes, so uses 10 bits addresses.
On figure 3 you can see an example of sending some data. Steps are the following:

- A writing signal and some data are coming from the `uart.vhd` parent module, the buffer save this data and increment the write address.
- The `not_empty` signal changes state because the read an write addresses are different. The `write.vhd` module catches the change, and begin to send the buffered data of the `read_address`.
- When data was sent, the `write.vhd` module send back a signal to increment the `read_address`. If `not_empty` signal stay true, it continues to send data.

The working principle of the read buffer is approximately the same, when a complete byte was received from Rx (in `read.vhd`), it sends a signal to the buffer in order to record the byte it and increment its write address. If read and write addresses are different and a read signal is coming from avalon bus, it write data on the bus and increments the read address.

3.5 Read

The read function is starting when we get a falling edge on the Rx channel. It uses an internal counter to wait predefined time : 1.5 * bit time for first bit reception, 1 * bit time for others, and it catches value at the end each wait, if it expects a bit, write it to a temporary vector. At the end of the read (8 bits received, and parity bit if active), it send a signal to the buffer to save data and return in a waiting state.

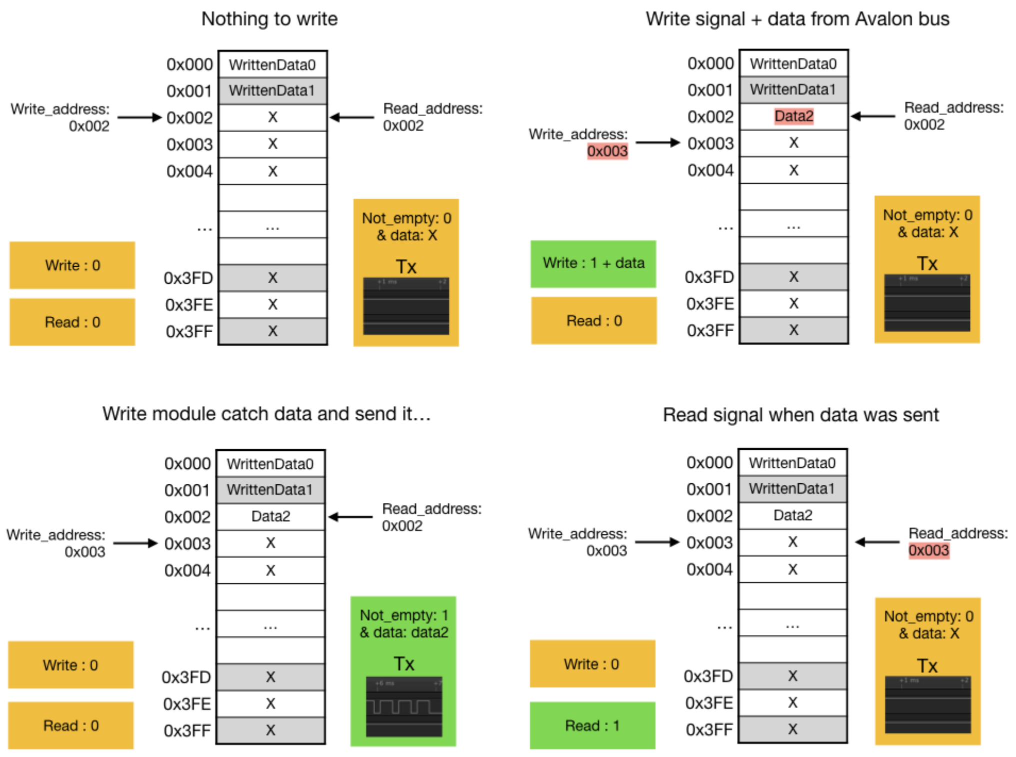


Figure 3: Write buffer example, some data coming from avalon bus to be written on UART

3.6 Write

The write submodule works the same as the read one, except that it write values from the buffer to the UART. Each time the counter reaches stop value it changes the written value (a kind of state machine) and come back to a all time high signal at the end of the process.

4 Analysis

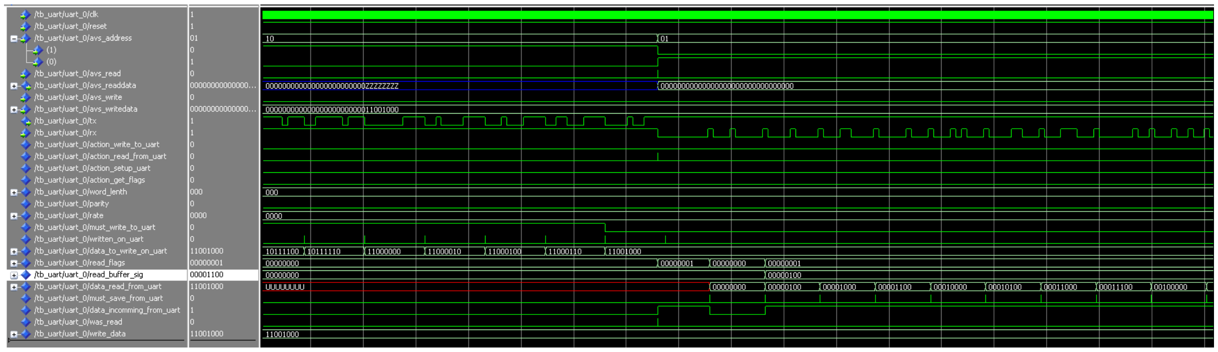
As mentioned at the beginning of this laboratory report, we processed step by step, we first created and tested everything virtually, by doing some test benches (The most complete is `tb_uart.vhd`). A preview can be seen on figure 4, we can see the `written_on_uart` signal which indicates when a byte was wrote on Tx or `must_save_from_uart` which indicates the end of the reception of a byte on Rx. Tests simulates read, write and flags using a fake avalon bus and check return values.

At the moment every bug was fixed we came to the real implementation on the FPGA. You can see final signals on figures 5 and 6 the calculus sent by the PC and the answer from the FPGA. Timing of the FPGA UART generated signal is as good as the one from PC, some analog signal analysis showed us that the signal is also very clean.

5 Conclusion

Finally, we were able to create a working UART connection using the FPGA and a Nios processor. The creation of the interface virtually was very easy to do and debug with modelsim was fast. Unfortunately, it wasn't the case for the second part of the laboratory, the tap logic analyzer is not as easy to use and compilation time can lead to hours waste. The three main issues we had was the inversion of the reset when we plugged our interface to the avalon bus, the address shift (of two bits) not taken in account (so nothing was write except setup) and the printf bug due to the overriding of the read/write system methods by our own methods in the UART drivers.

This exercise was a really good one, it led us to a best comprehension of hardware, and, again, required form us an all time greater need of attention and rigorousness, which is very very important to become highly skilled in low level development.



```
/tb_uart/uart_0/clk                      1
/tb_uart/uart_0/reset                    1
/tb_uart/uart_0/avs_address              01
  (1)                                    0
  (0)                                    1
/tb_uart/uart_0/avs_read                 0
/tb_uart/uart_0/avs_readdata             00000000000000...
/tb_uart/uart_0/avs_write                0
/tb_uart/uart_0/avs_writedata            00000000000000...
/tb_uart/uart_0/tx                       1
/tb_uart/uart_0/rx                       1
/tb_uart/uart_0/action_write_to_uart     0
/tb_uart/uart_0/action_read_from_uart    0
/tb_uart/uart_0/action_setup_uart        0
/tb_uart/uart_0/action_get_flags         0
/tb_uart/uart_0/word_lenth               000
/tb_uart/uart_0/parity                   0
/tb_uart/uart_0/rate                     0000
/tb_uart/uart_0/must_write_to_uart       0
/tb_uart/uart_0/written_on_uart          0
/tb_uart/uart_0/data_to_write_on_uart    11001000
/tb_uart/uart_0/read_flags               00000001
/tb_uart/uart_0/read_buffer_sig          00001100
/tb_uart/uart_0/data_read_from_uart      11001000
/tb_uart/uart_0/must_save_from_uart      0
/tb_uart/uart_0/data_incomming_from_uart 1
/tb_uart/uart_0/was_read                 0
/tb_uart/uart_0/write_data               11001000
```

Figure 4: Modelsim simulation preview, only submodule link signals

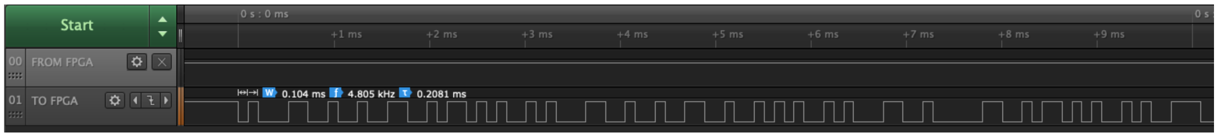


Figure 5: Signal sent by the distant device (PC) to the FPGA, over UART

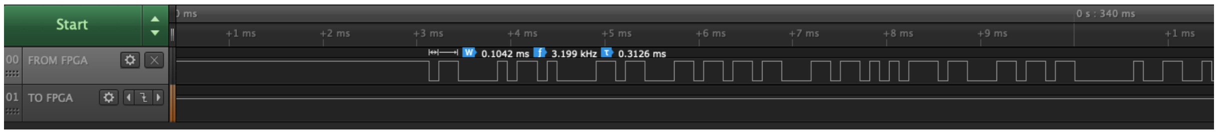


Figure 6: Answer sent by the FPGA to the distant device (PC), over UART