Computer Engineering Master's Degree Course (LM-32)

Report #2

UART SW

Team members
S354939 - Pietro Alberto Levo
S358481 - Sandro Marghella
S359182 - Gianluca Riva Governanda
S361451 - Cristina Rizzo

Professor
Claudio Passerone

November 11th, 2025

Introduction

This lab focuses on creating, configuring, and running an embedded system based on the Nios II soft-core processor on the Intel Cyclone V FPGA (DE1-SoC board). The main goal is to build an interface unit that connects the desktop and the FPGA system using the UART (Universal Asynchronous Receiver-Transmitter) protocol. Unlike higher-level systems where the interface is managed by an operating system driver, this project investigates "bit-banging" and a low-level hardware approach.
The project uses General Purpose I/O (GPIO) pins to sample and send the wire line UART signals. This work aims to clarify the relationship between software and hardware, especially how software execution and the timing of physical signals interact.

Educational Objectives

The educational journey begins with a simple "Hello World" system and progresses to more complex tasks. This includes a standalone UART receiver that can work with different baud rates. The lab design allows for system validation and the real-time study of various signals using an oscilloscope. This setup enables the examination of embedded real-time systems where timing and precise control of the processor are crucial. It allows for the direct observation of start/stop bits, parity bits, and propagation delays.

System Configuration and Setup

Implementing on the DE1-SoC board requires a specific setup to connect the ARM-based Hard Processor System (HPS) and the FPGA fabric:

- **Hardware Initial Boot:** Placement of the MicroSD card has to be done before turning the hardware on for the first time. This enables the ARM HPS to route the UART lines to the FPGA.
- **Board Settings:** The manual switch SW(9) is set to 0 to enable software UART mode.
- **System Architecture:** The design uses a Nios II/e soft-core processor integrated through Platform Designer. Since there is no hardware UART block available, the system entirely relies on bit-banging using GPIO pins.
- **Software & BSP Settings:** In the BSP Editor, `stdout` is mapped to `jtag_uart_0` and the `timestamp_timer` to `timer_0`.
- **Implementation Logic:** The C code is designed to detect the Start Bit, pause for 0.5 bit-time to sample at the midpoint of the pulse, and then loop through the data bits using a high-resolution timer.
- **Debugging:** we used an oscilloscope with GPIO header pins to check real-time timing, start/stop bits and propagation delays.

1 Projects

1.1 Project #1

To start the experience, the "Hello world" program is run.

```c
#include <stdio.h>
#include "system.h"
#include "sys/alt_timestamp.h"
#include "altera_avalon_pio_regs.h"

int main()
{
  printf("Hello Sandro, Gianluca, Pietro and Cri!\n");
```

```

    return 0;
}
```

The above program is running on the hardware described in VHDL files and flashed to the FPGA. In order to make Software and Hardware able to work together is necessary the BSP. This work as an Hardware Abstraction Layer which is generated based on the instantiation of the different peripherals and components with other informations, all available on the SOPC file, in this case the *nios_ hps_ system.sopcinfo*.

1.2 Project #4

The project 4's main purpose is to create an UART receiver with software and test it sending characters from PuTTY to the Cyclone V board.
To do so, it been first defined the following costants at the beginning of the code: In this project

```
#define NBIT        8
#define NSTOPBIT    1
#define NOPARITY    0
#define EVENPARITY  1
#define ODDPARITY   2
#define PARITY      NOPARITY
```

Listing 1: Constants

it was not used a parity bit, so each transmission consists of exactly 10 bits: 1 start bit, 8 data bits, and 1 stop bit.

1.2.1 Version #1

In order to create an UART receiver the focus is on the UART's message structure.

```
#define BAUDRATE 300
int main() {
    int ticks_per_sec = alt_timestamp_freq();
    int ticks_per_bit = ticks_per_sec / BAUDRATE;
    int c[NBIT];
    int val;

    printf("UART RX ready.\n");

    while (1) {

        do {
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
        } while (val == 1);

        alt_timestamp_start();
        while (alt_timestamp() < (ticks_per_bit >> 1)) {}

        val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
        if (val != 0) {
            continue;
        }
        alt_timestamp_start();
        int sample_times[NBIT + NSTOPBIT];
        for (int i = 0; i < NBIT + NSTOPBIT; i++)
            sample_times[i] = (i + 1) * ticks_per_bit;

        for (int i = 0; i < NBIT; i++) {
            while (alt_timestamp() < sample_times[i]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[i] = val;        // c[0] = LSB
        }

        while (alt_timestamp() < sample_times[NBIT]) {}
        val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;

        if (val != 1) {
            printf("ERROR: stop bit not valid!\n");
            continue;

```

```c
41         int result = 0;
42         for (int i = 0; i < NBIT; i++)
43             result |= (c[i] << i);
44 
45         printf("Received: %c (0x%02X) (%d)\n", result, result, result);
46     }
47 
48     return 0;
49 }
```

Listing 2: Version_1

The code first waits for the start bit. The default state of the UART line is 1, so the start bit is represented by 0. To ensure that it is not an error, the code reads the line state after half a bit period (see line 21 of 2). If the check passed, the FPGA computes the sampling times and then begins sampling the other 8 data bits of the character encoded in ASCII, as shown in the for loop on line 28.
The 38th line makes sure that the trasmission ends with a stop bit (equal to 1) and then it builds the character (for loop in line 42).

1.3 Version #2

In 2 the code use a for loop to read the 8 char bit trasmitted by UART. The branch instructions used by for loop can slow down execution.
For this reason the 3 uses loop unrolling to improve performance:

```c
int main() {

    int ticks_per_sec = alt_timestamp_freq();
    int ticks_per_bit = ticks_per_sec / BAUDRATE;

    int c[NBIT];
    int val;

    printf("UART RX ready.\n");

    while (1) {

        do {
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
        } while (val == 1);

        alt_timestamp_start();
        while (alt_timestamp() < (ticks_per_bit >> 1)) {}

        // Check on the start
        val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
        if (val != 0) {
            continue;
        }

        alt_timestamp_start();
        int sample_times[NBIT + NSTOPBIT];
        for (int i = 0; i < NBIT + NSTOPBIT; i++)
            sample_times[i] = (i + 1) * ticks_per_bit;


        // 4. reads DATA's bits - loop unrolling

            while (alt_timestamp() < sample_times[0]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[0] = val;

            while (alt_timestamp() < sample_times[1]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[1] = val;

            while (alt_timestamp() < sample_times[2]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[2] = val;

            while (alt_timestamp() < sample_times[3]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[3] = val;

            while (alt_timestamp() < sample_times[4]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[4] = val;

            while (alt_timestamp() < sample_times[5]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[5] = val;

            while (alt_timestamp() < sample_times[6]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[6] = val;

            while (alt_timestamp() < sample_times[7]) {}
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
            c[7] = val;

        while (alt_timestamp() < sample_times[NBIT]) {}
        val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;

        if (val != 1) {
            printf("ERROR: stop bit not valid!\n");
            continue;
        }

        int result = 0;
        for (int i = 0; i < NBIT; i++)
            result |= (c[i] << i);

        printf("Received: %c (0x%02X) (%d) (%d%d%d%d%d%d%d%d)\n", result, result, result, c[7], c[6], c[5], c[4], c[3], c[2], c[1], c[0]);
    }
    return 0;
}
```

Listing 3: Version_2

Thanks to this approach the compiler avoids repeated branch instructions, making the code faster and more efficient than Listing 2.

1.4 Project #5

In this project it is performed the reception and transmission of a char over a 8N1 UART implemented driving GPIOs via software.
This is a time optimized code to enable communication between the PC and the FPGA

```c
int main() {
    int ticks_per_sec = alt_timestamp_freq();
    int ticks_per_bit = ticks_per_sec / BAUDRATE;

    int c[NBIT];
    int val;

    IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, 1);

    printf("UART RX ready.\n");

    while (1) {
        // RX PHASE

        // 1. Wait START bit
        do {
            val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
        } while (val == 1);

        // 2. Start timer and wait half bit time
        alt_timestamp_start();
        while (alt_timestamp() < (ticks_per_bit >> 1)) {}

        // Check start bit
        val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
        if (val != 0) continue;

        // 3. Calculate sampling times (unrolled)
        alt_timestamp_start();
        int sample_times[NBIT + NSTOPBIT];
        for (int i = 0; i < NBIT + NSTOPBIT; i++)
                sample_times[i] = (i + 1) * ticks_per_bit;
```

```c
// 4. Read DATA bits
while (alt_timestamp() < sample_times[0]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
c[0] = val;

while (alt_timestamp() < sample_times[1]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
c[1] = val;

while (alt_timestamp() < sample_times[2]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
c[2] = val;

while (alt_timestamp() < sample_times[3]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
c[3] = val;

while (alt_timestamp() < sample_times[4]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
c[4] = val;

while (alt_timestamp() < sample_times[5]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
c[5] = val;

while (alt_timestamp() < sample_times[6]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
c[6] = val;

while (alt_timestamp() < sample_times[7]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;
c[7] = val;

// 5. Read STOP BIT e verify
while (alt_timestamp() < sample_times[NBIT]) {}
val = IORD_ALTERA_AVALON_PIO_DATA(NIOS_UARTRX_BASE) & 0x01;

if (val != 1) {
    printf("ERROR: invalid stop bit!\n");
    continue;
}

// 6. Build result
int result = 0;
result |= (c[0] << 0);
result |= (c[1] << 1);
result |= (c[2] << 2);
result |= (c[3] << 3);
result |= (c[4] << 4);
result |= (c[5] << 5);
result |= (c[6] << 6);
result |= (c[7] << 7);

printf("Ricevuto: %c (0x%02X) (%d) (%d%d%d%d%d%d%d%d);\n", result, result, result, c[7], c[6], c[5], c[4], c[3], c[2], c[1], c[0])

// TX PHASE
alt_timestamp_start();

// Start bit
IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, 0);
while (alt_timestamp() < ticks_per_bit) {}

// DATA bits
IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, c[0]);
while (alt_timestamp() < ticks_per_bit*2) {}

IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, c[1]);
while (alt_timestamp() < ticks_per_bit*3) {}

IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, c[2]);
while (alt_timestamp() < ticks_per_bit*4) {}

IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, c[3]);
while (alt_timestamp() < ticks_per_bit*5) {}

IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, c[4]);
while (alt_timestamp() < ticks_per_bit*6) {}

IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, c[5]);
while (alt_timestamp() < ticks_per_bit*7) {}

IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, c[6]);
while (alt_timestamp() < ticks_per_bit*8) {}
```

```
        IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, c[7]);
        while (alt_timestamp() < ticks_per_bit*9) {}

        // Stop bit
        IOWR_ALTERA_AVALON_PIO_DATA(NIOS_UARTTX_BASE, 1);
        while (alt_timestamp() < ticks_per_bit*10) {}

    }
    return 0;
}
```

The baud rates values considered were 110, 150, 300, 1200, 2400, 4800, 9600, 19200.
PuTTY [1] failed to establish a UART connection for the lowest values of baud rates. As an alternative, tio [2] was used, which successfully connected. However, the readings were wrong, as evidenced by (1a) and (1b). It is clear how the TX and RX are completely different.
For baud rates above 150 and up to 9600, transmission and reception are correct, as represented by (2a), (2b), (2c), (2d) and (2e).
Since 19200 the sampling becomes unreliable. From (3b) it is easy to understand that:

- The sampling happens in delay, which results in a bit shifting in the transmission communication. This is most likely caused by software overhead due to busy-waiting loops.
- The TX signal is also delayed. At 19200 baud rates the time bit is $\frac{1}{BR} \approx 52\ \mu s$, but the oscilloscope shows each symbol lasts $\approx 60\ \mu s$

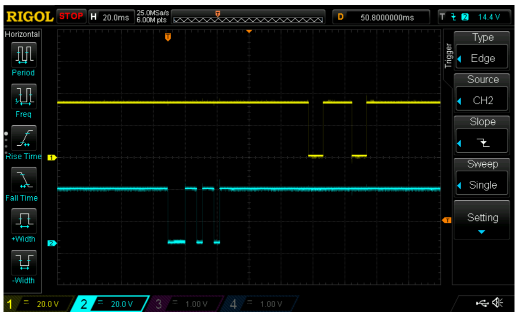
References

[1] Putty. https://www.chiark.greenend.org.uk/~sgtatham/putty/, 2025. Version 0.81.

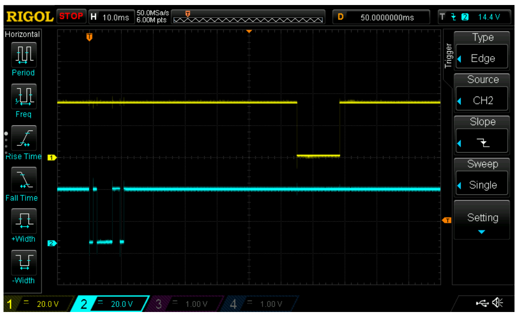
[2] tio: a simple serial device i/o tool. https://github.com/tio/tio, 2025. Accessed: 2025-12-02.

A Appendix

Blue signal: RX line. Yellow signal: TX line.

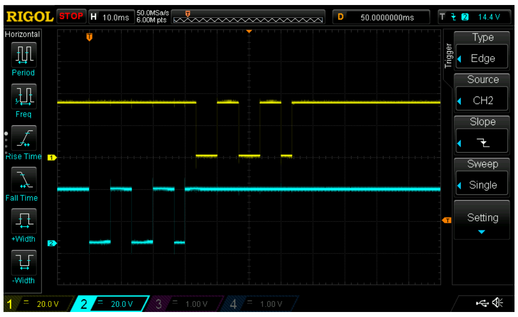

RIGOL STOP H 20.0ms 25.0MSa/s 6.00M pts
D 50.8000000ms
T 2 14.4 V
Type Edge Source CH2 Slope Sweep Single Setting
1 20.0 V 2 20.0 V 3 1.00 V 4 1.00 V

(a) 110 BR

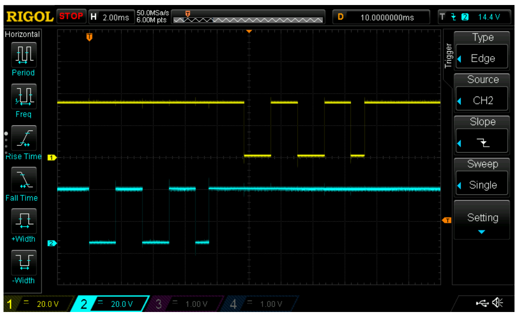

RIGOL STOP H 10.0ms 50.0MSa/s 6.00M pts
D 50.0000000ms
T 2 14.4 V
Type Edge Source CH2 Slope Sweep Single Setting
1 20.0 V 2 20.0 V 3 1.00 V 4 1.00 V

(b) 150 BR

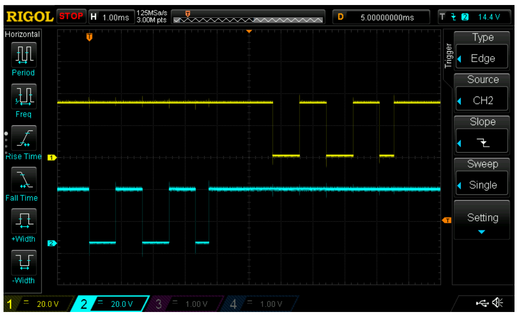
Figure 1


RIGOL STOP H 10.0ms 50.0MSa/s 6.00M pts
D 50.0000000ms
T 2 14.4 V
Type Edge Source CH2 Slope Sweep Single Setting
1 20.0 V 2 20.0 V 3 1.00 V 4 1.00 V

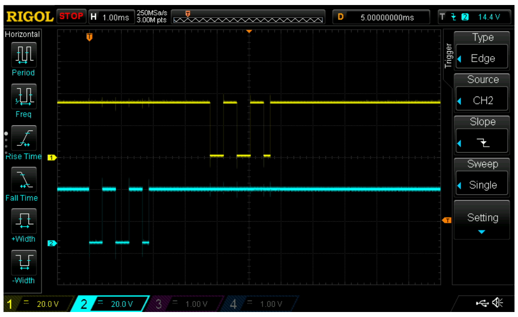
(a) 300 BR


RIGOL STOP H 2.00ms 50.0MSa/s 6.00M pts
D 10.0000000ms
T 2 14.4 V
Type Edge Source CH2 Slope Sweep Single Setting
1 20.0 V 2 20.0 V 3 1.00 V 4 1.00 V

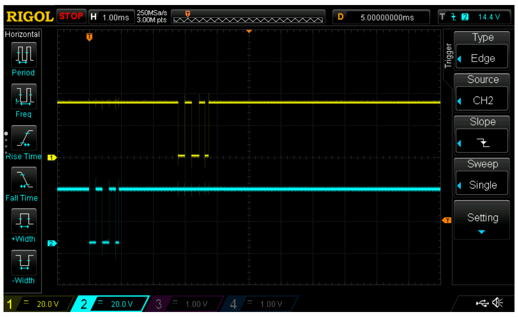
(b) 1200 BR


RIGOL STOP H 1.00ms 125MSa/s 3.00M pts
D 5.00000000ms
T 2 14.4 V
Type Edge Source CH2 Slope Sweep Single Setting
1 20.0 V 2 20.0 V 3 1.00 V 4 1.00 V

(c) 2400 BR

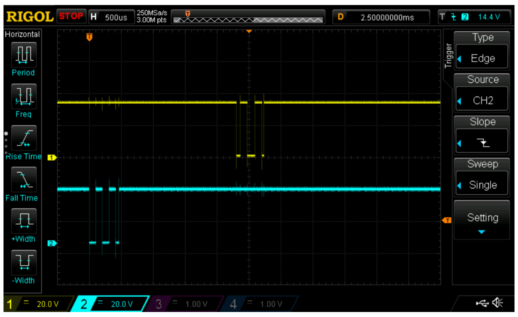

RIGOL STOP H 1.00ms 250MSa/s 3.00M pts
D 5.00000000ms
T 2 14.4 V
Type Edge Source CH2 Slope Sweep Single Setting
1 20.0 V 2 20.0 V 3 1.00 V 4 1.00 V

(d) 4800 BR

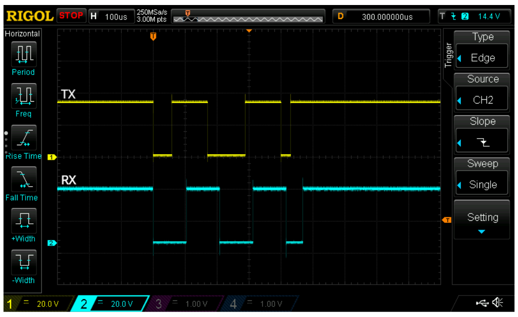

RIGOL STOP H 1.00ms 250MSa/s 3.00M pts
D 5.00000000ms
T 2 14.4 V
Type Edge Source CH2 Slope Sweep Single Setting
1 20.0 V 2 20.0 V 3 1.00 V 4 1.00 V

(e) 9600 BR

Figure 2



(a) 19200 BR



(b) 19200 BR: TX and RX compared

Figure 3