PREPARED FOR SUBMISSION TO JINST

UART controlled stopwatch

C. Donazzolo and G. Gavello

Università degli Studi di Torino

E-mail: chiara.donazzolo@edu.unito.it, gaia.gavello@edu.unito.it

ABSTRACT:
The purpose of this paper is to describe a possible way of implementing a UART controlled stopwatch on FPGA. Its contents will reflect the typical design flow starting from Verilog sources up to hardware implementation passing through the required verification tests.

Contents

1 Introduction

The scope of this project is to implement a stopwatch on an FPGA board.
The FPGA is programmed by writing source code in Verilog, which is then be simulated, synthesized and implemented using Xilinx's Vivado Design Suite. The timing analysis function is used to obtain a timing clean design.

The stopwatch is meant to be controlled by the user through typing characters on a computer keyboard, therefore it's necessary to detail a communication protocol between the board and the computer. More specifically, the UART protocol is used.
The user will be able to stop, start and reset the stopwatch, and will also be able to get a timestamp upon pressing a specific key. The user's inputs are collected using a PuTTY console, which is also used to show the requested timestamps.

A single digit seven segment display and a four digits one are used to display the time. The single digit is used to display the minutes (up to 9) while the four digits are used to display the hundredths of seconds (up to 5999).

The following paper will detail each step of the project.

2 Realization

The stopwatch is implemented using a central module, *StopWatch*, responsible for receiving data from the computer through the *uartRx* module, regulating the counter in the *StopWatchCounter* module accordingly. The results are displayed on a seven-segment display using the *FourDigits-Display* module and eventually transmitted back to the computer through the *uartTx* module. Each module will now be detailed.

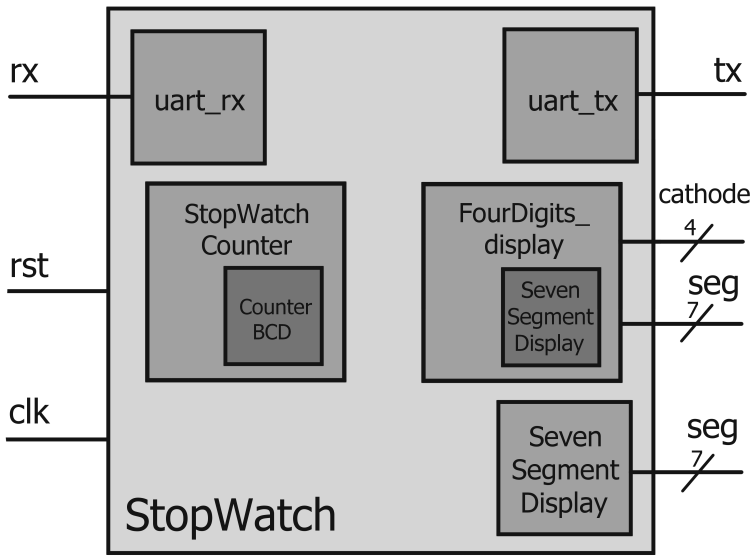


Figure 1: UART protocol

2.1 CounterBCD

A parametrized binary coded decimal counter from 0 to the parameter. This module receives as inputs a clock signal, a reset signal and an enable signal, and gives as outputs the 4-bits long value of the counter and a signal for the carry. When the maximum is reached, the carry signal is set to high until the next iteration, in which the counter is forced to roll over.

2.2 StopWatchCounter

This module uses cascading *CounterBCD* modules to count up to 9 minutes and 59.99 seconds.
It receives a clock signal, a reset signal and an enable signal as inputs and it gives a 4-bits long output for each of the digits of the counter, plus an overflow signal that is asserted as the counter rolls over after having reached the maximum.
The input clock, which is assumed will be a 10MHz provided by a PLL, is the clock input of each *CounterBCD* instance and is used as input for a *TickCounter* module to generate a signal once every 0.01 seconds. The result of the logic AND between the tick signal and the enable signal received as input is used as the enable for the first counter, which gives the value of the hundredth of seconds

digit. The carry flag of this module is the enable signal for the next counter and so forth. When the carry signal for the last counter turns high so does the overflow signal.
Each counter goes up to 9 except for the one that gives us the value of the tens of seconds, which goes up to 5.

2.3 uartRx

The communication between the FPGA and the computer follows the UART protocol, provided that both parties involved settle on the same frequency of communication, also known as baud rate. The UART protocol requires a low bit signaling the start of the communication, 8 data bits and a high bit signaling the end of the communication. The signal then remains high until the next starting bit.

Figure 2: UART protocol

The UART receiver takes as inputs a clock and a reset signal, a tick signal that acts as an enabler and a 1-bit signal that carries the data sequentially. The module then reconstructs the input back into an 8-bits long output and it also gives as output a signal that turns high when the reception has been completed.
To avoid a metastable situation that might occur if we try to read the value of the input bit as it's still changing, the frequency of the input enable signal is actually 16 times the baud rate. This technique is called oversampling: as will be detailed later, this subdivision is used to read the value of the data no further than 1/16th from the middle point of the bit transmission, making sure that the signal will be stable.

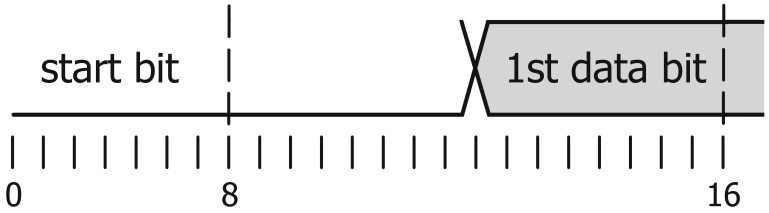


Figure 3: Oversampling

We used a finite state machine to code this module.
There are four possible states: IDLE, START, DATA and STOP.

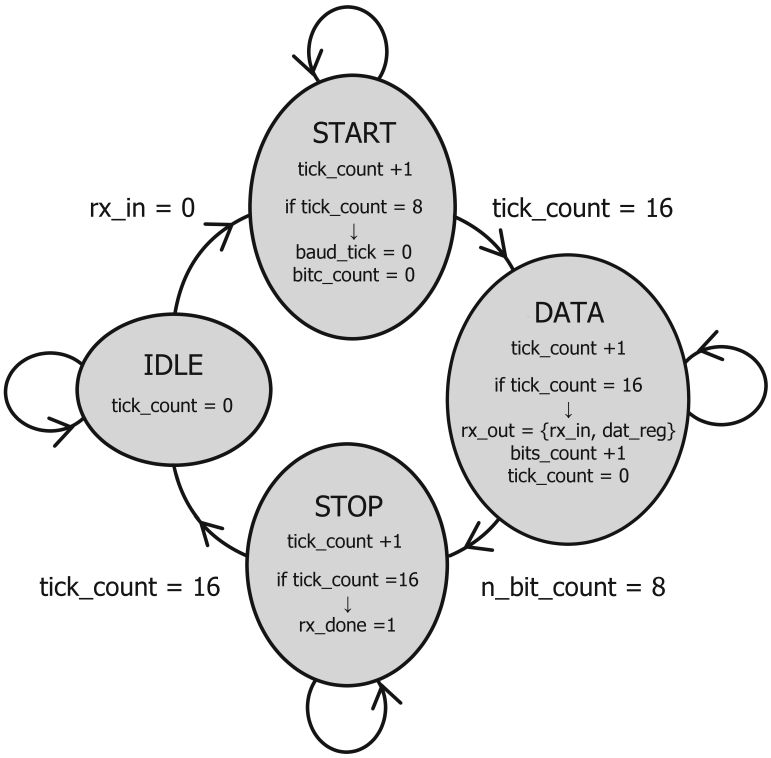


Figure 4: FSM for *uartRx*

The IDLE state is left when the low start bit is received: the next state is set as START and the tick counter is set to 0.
In the START state we wait for the middle point of the transmission of the start bit: only when the tick counter reaches 8 the next state is set as DATA. The tick counter is reset and a counter for the number of received bits is set to 0.
In the DATA state we wait for the tick counter to reach 16, and since it was reset in the middle of the transmission of the start bit this will happen when the middle of the first data bit is reached. When that happens, we sample the value of the current data bit and we concatenate it to an 8-bits data register which will be our output. The tick counter is then reset so that after 16 counts the next data bit will be sampled. Each time this happens the received bits counter is incremented, and when it reaches 8 the next state is set to STOP.
In the STOP state we wait once again for the tick counter to reach 16. When that happens, the signal indicating that the reception is done is set to high and we go back to the IDLE state.
If the reset signal is high the state is set to IDLE, the data register is set to all zeros and the counters are reset. For further details see ref. [1].

2.4 uartTx

The UART transmitter takes as input a clock signal, a reset signal, a tick signal acting as an enabler, a start signal that when high will initiate the transmission and an 8-bit long signal that we want to transmit as sequential bits. The output will consequently be a single bit signal, plus a signal that

turns high when the transmission is done.
In this case the oversampling process isn't needed, but we will use the same tick signal with a frequency of 16 times the baud rate and shift a bit out every 16 ticks.
This module was also realized using a finite state machine with the same 4 states as the receiver.

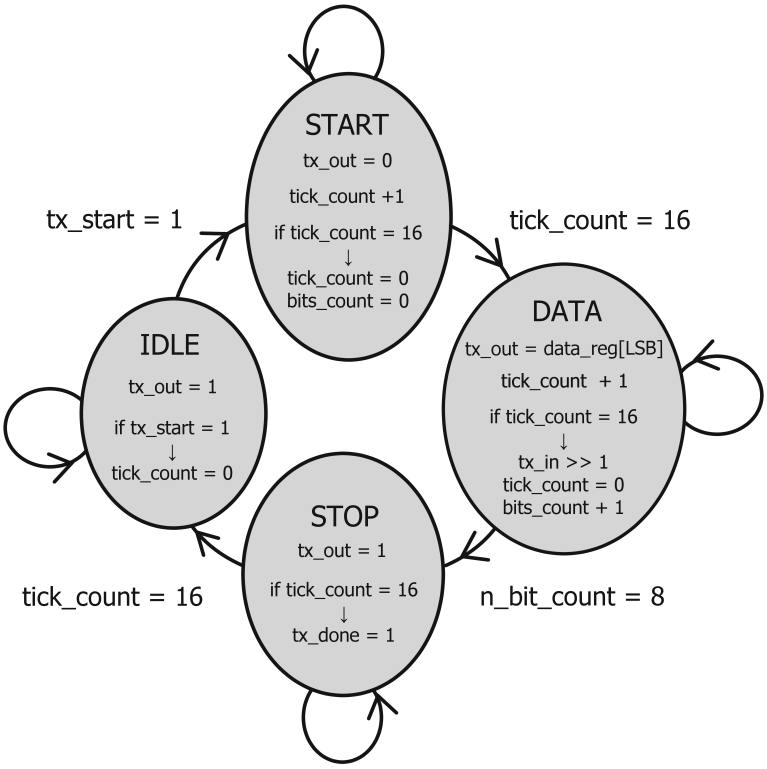


Figure 5: FsM for *uartTx*

In the IDLE state the output line is kept high until the *start* signal is asserted. When that happens, the next state is set to START, the tick counter is set to 0 and we load the 8-bits input in an 8-bits data register that will be used in the transmission.
In the START state a low bit is given as output and after counting 16 ticks the next state is set to DATA, the tick counter is reset and a counter used for the number of transmitted bits is set to 0.
In the DATA state the output of the module is set to be the least significative bit of the data register. When the tick counter reaches 16 the data register is shifted right, the tick counter reset and the transmitted bits counter is incremented. When this counter reaches 8 the next state is set to STOP.
In the STOP state a high bit is transmitted and after 16 ticks the signal indicating the completion of the transmission is raised and we go back to the IDLE state.
As was the case for the *uartRx* module, if the reset signal is high the state is set to IDLE, the data register is set to all zeros and the counters are reset. The output is also set to high.

2.5 SevenSegmentDecoder

Given a 4-bit value between 0 and 9 as input, the output of this module is the value that each segment of a seven segment display must have to represent the number. The display in our possesion was a common cathode one so we coded the value of the segments accordingly.

2.6 FourDigitsDisplay

A module for controlling a 4 digits seven segment display, it receives as inputs a clock signal and the four 4-bits long binary repesentations of the digits we want to display and gives as outputs the values of the seven segments and a 4-bits long signal that is used to control which one of the digits is turned on.
A 2-bits counter with a frequency of about 98kHz is be used to decide which digit will be turned on and what value will be displayed. The value will then be passed on as an input to a *SevenSegmentDecoder* module.
The display used was a common cathode one, so the signal selecting which digit is turned on uses a one cold encoding accordingly. For further details see ref. [2].

2.7 StopWatch

This is the main module of the project. It receives as inputs a clock signal, a reset signal and a sequential bits signal from the computer, while it gives as outputs a sequential bit signal back to the computer and the signals needed to drive a single digit and a 4 digits seven segment display. The input clock is fed into an IP-compiled *PLL* module. Each module instantiated here is then driven by the 10MHz PLL output clock and uses the logic OR between the pll-locked signal and the input reset as a syncronous reset signal.

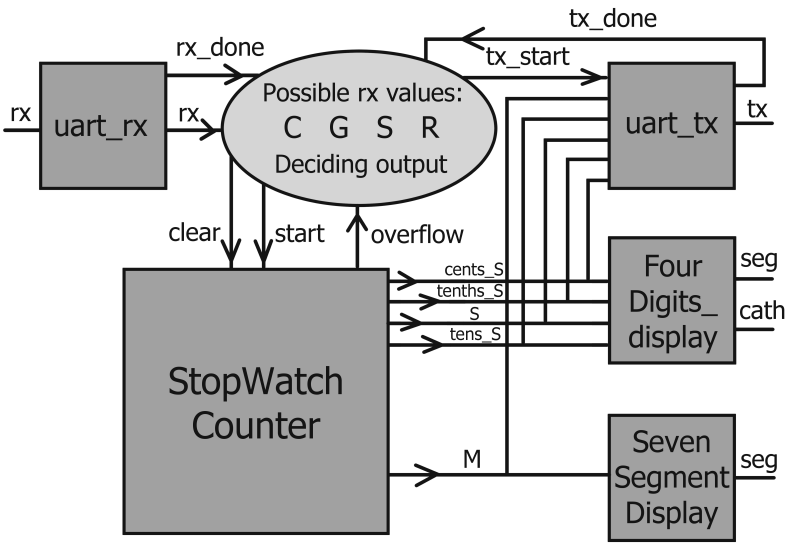


Figure 6: Generalized *StopWatch* module scheme

An 8-bits ascii character is sent as a sequential input and it is handled by an *uartRx* module, which gives as output the original 8-bits value. We then use a finite state machine to determine how to respond to the input. In the FSM there are two possible states, IDLE and TRANSMIT.

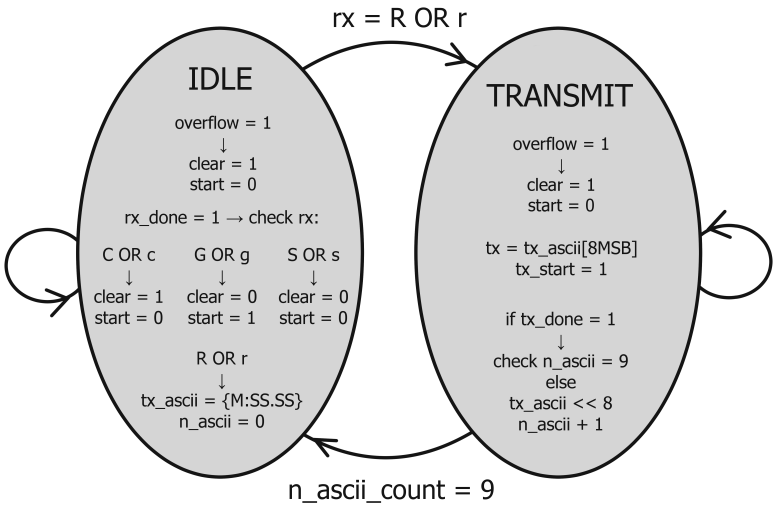


Figure 7: FSM for *Stopwatch*

In the IDLE state once the signal that indicates the reception is done is high we identify which character has been sent by the user. If it's *C (clear)*, *S (stop)* or *G (go)* the future state of the *clear* and *start* variables is decided. These variables are the reset and enable inputs of the *StopWatchCounter* instance respectively. If the R (*read*) character is received each digit of the current time output of the *StopWatchCounter* is converted to and 8-bits ascii and is stored as a concatenation in the M:SS.SS format, with two added characters in the end for displaying purposes, in a designed register called *txAscii*. A 3-bits counter for the number of ascii characters that have been trasmitted is set to zero and the next state is set to TRANSMIT.
In the TRANSMIT state the eight most significative bits of *txAscii* are set as the 8-bits input of the *uartTx* module and the enable of said module is set to high. Once the signal indicating that the transmission has been completed is high if the 9 ascii characters have already been transmitted the next state is set back to IDLE, otherwise *txAscii* is shifted left and the counter for the number of transmitted character is incremented by 1.
Each state presents a starting condition that sets *clear* to high and *start* to low if the overflow output of the *StopWatchCounter* is high, meaning that the counter will be reset and stopped when the maximum is reached. If this happens in the TRANSMIT state the transmission will be completed and the state will then return to IDLE, waiting for the user's input before starting the stopwatch again.
If the reset input signal is high the state is set to IDLE, the *clear* signal is high and the *start* signal is low.

An *uartTx* module is used to handle transmissions to the computer. It uses the same clock and reset signals as the rest of the modules, while the enable signal and the 8-bits signal to be transmitted are given by the TRANSMIT state of the FSM.
Both the *uartTx* and the *uartRx* modules have a baud tick of 9600*16Hz which is generated using a *TickCounterRst* module.

A *StopWatchCounter* module is instantiated using the clock, *clear* and *start* signals as mentioned before. The overflow output signal is used to reset the counter after the maximum is reached as previously stated. The time output, other than being stored in *txAscii* if needed, are given as inputs to the modules used to drive the seven segment display: a *SevenSegmentDecoder* module is called for the minutes digit, which will be displayed on a single digit display, while a *FourDigitsDisplay* module is called for the other digits.

3 Simulations

Testing the implemented modules on known signals is a key point to check whether the code behaves as expected or not. Therefore, a step-by step approach is adopted and each key feature is simulated separately.

Since the *StopWatchCounter* module is the core of the device, the first test is intended to verify if the counter works properly or not. In this case, the *TickCounter* is set a thousand times faster than what is required in such a way to see the *over* flag activating in a reasonable time. A combination of high/low *start* and *reset* is also tested at the very beginning of the simulation. Merely for graphical reasons, the clock is not added to the waveforms window.

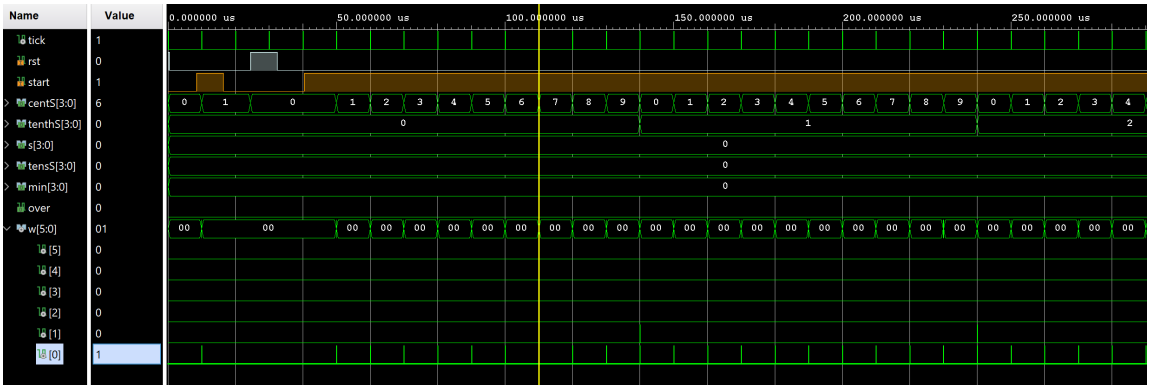


Figure 8: Counting *StopWatchCounter*

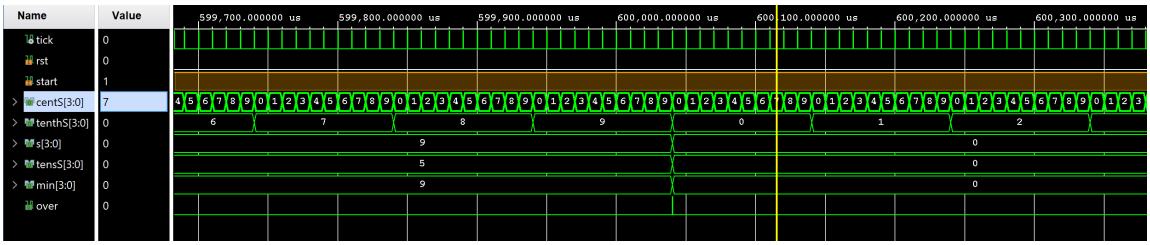


Figure 9: Active *over* flag

At this point, the attention can be focused on the *uartRx* module. Depending on the ASCII character received, a different task is performed. Therefore, simulating some UART serialized ASCII characters is a crucial part of the testing procedure. Assuming a 9600 Hz baud rate, every 104160 ns a bit of the simulated signal is assigned to the *rx* input until the sequence is completed. The tested ASCII characters are 47, 53, 43 which respectively stand for *G (go)*, *S (stop)* and *C (clear)*.

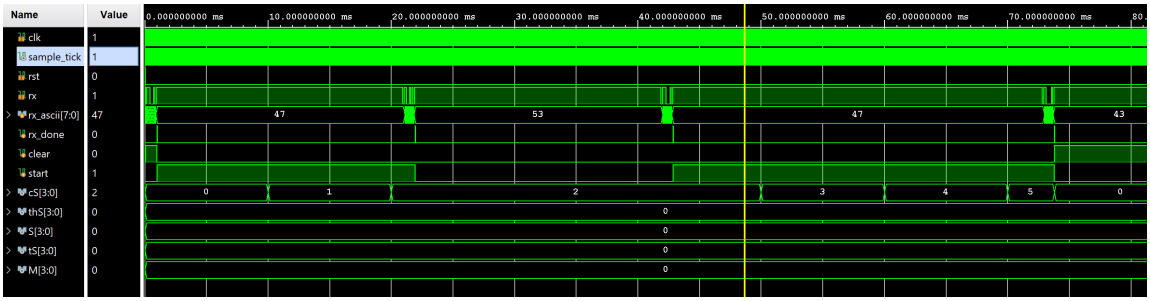


Figure 10: *UartRx* module

Since everything seems to work properly, the *uartRx* module can be exploited to check the *uartTx* behaviour. Therefore the ASCII character 52, which enables the timestamping, is provided to the *uartRx* input. In this case, two different parts are tested at the same time: the registers for the time saving and the real UART transmitter.

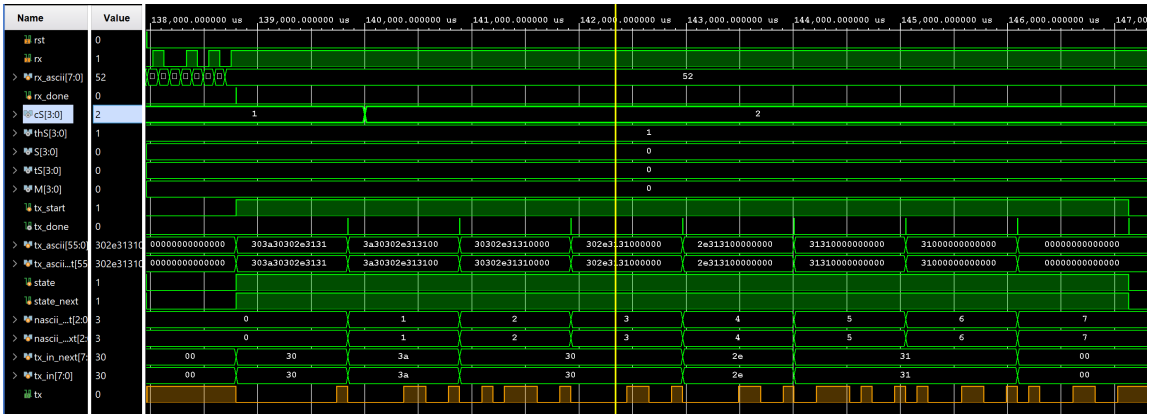


Figure 11: *UartTx* module

4 Implementation

Since the RTL simulation behaves as expected, it is possible to move on to the next steps in the design flow: synthesis and implementation.
At the end of the placement and routing procedure, the Vivado tool provides some important information: the resource utilization and the results of the static timing analysis. Initially, the following timing constraints are set:

- clock of period 10 ns
- input delay 4 ns
- output delay 4 ns

and, concerning the outputs, setup timing violations arise. In order to avoid them, a bench of flipflops is placed between the combinatorial logic outputs and the output pins. However, this solution is not enough and setup violations persist. The only way to get rid of them is to use a slower clock and this is exactly the reason why the frequency of the PLL output clock is set to 10 MHz. This choice does not affect the behaviour of the circuit since the counter is much slower, but rather the power consumption is reduced. Therefore, by assuming the same delays as before, the timing report is now:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,102 ns	Worst Hold Slack (WHS):	0,033 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	409	Total Number of Endpoints:	409	Total Number of Endpoints:	183

All user specified timing constraints are met.

Figure 12: Timing analysis report

At this point, the attention can be focused on the resource utilization. The following figures report the percentage of the assigned FPGA primitives and show their placement.

Resource	Utilization	Available	Utilization %
LUT	133	20800	0.64
FF	177	41600	0.43
IO	25	210	11.90
BUFG	2	32	6.25
PLL	1	5	20.00

Figure 13: Resource utilization

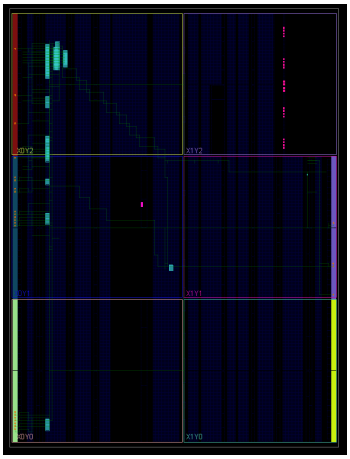
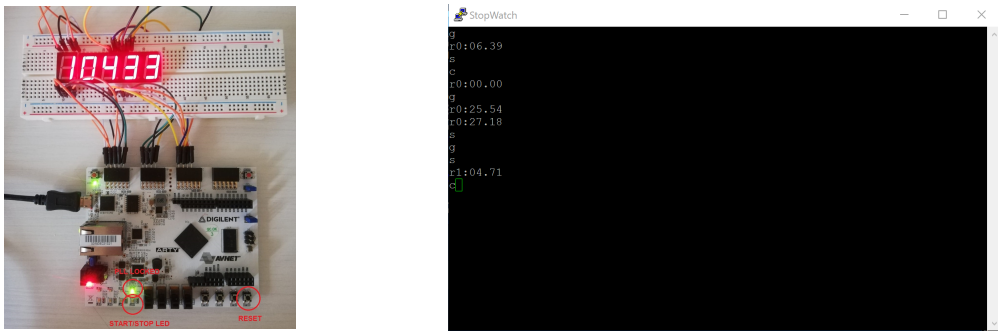


Figure 14: Resource placement

5 User interface and real hardware

In the end, a possible use of the stopwatch previously described is reported in the figures below. The PuTTY console allows to open a serial PC-FPGA comunication once the right COM port is set. Therefore, the user can directly control the stopwatch through the keyboard. In order to display both inputs and outputs, the character echo should be enabled. Moreover, according to the transmitted signal, specific leds on FPGA turn on or off so that the user can always check if the device works properly or not. For instance, when the PLL is locked and so the device is ready to be used, a red led becomes high.



(a) FPGA stopwatch

(b) PuTTY interface

Figure 15: Example of a possibile use

6 Conclusions

The implementation of the stopwatch is successful. The final design is timing clean, the displays show the time running and the communication to and from the computer works as intended, as expected from the simulations. The end product is a functional tool.

A Schematics

In order to make the paper more readable the key schematics, which can be exported form Vivado, are reported here.

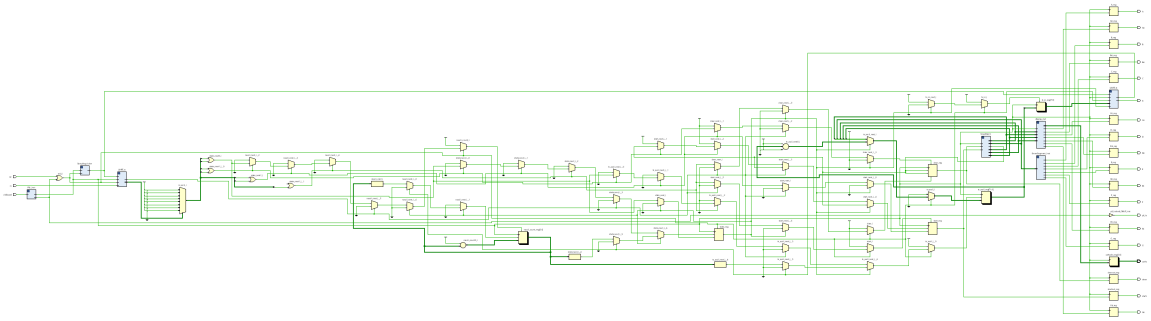


Figure 16: Post RTL analysis schematic

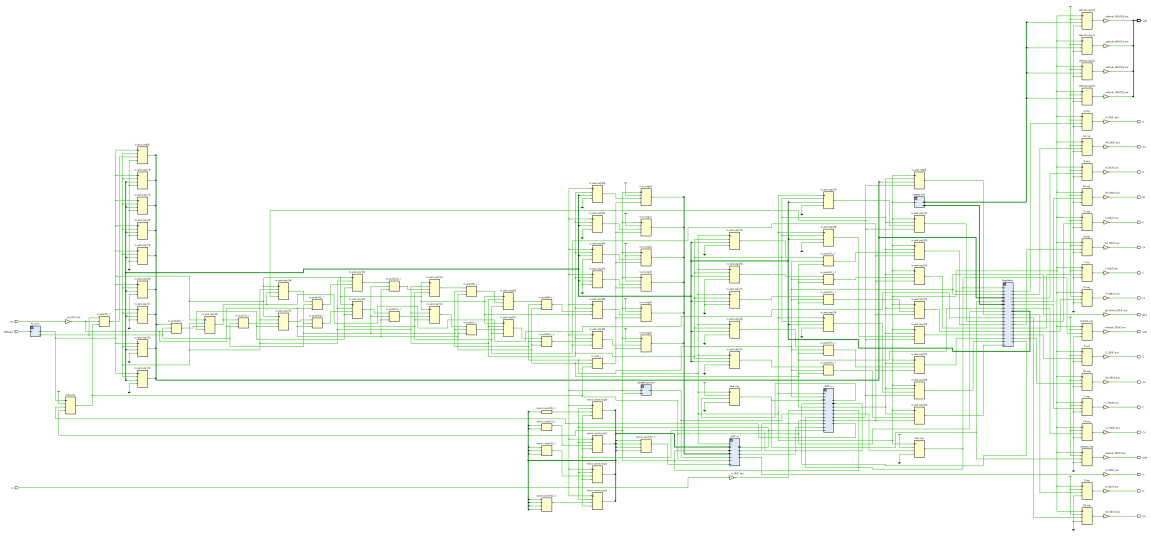


Figure 17: Post implementation schematic

References

[1] Pong P. Chu, *FPGA prototyping by Verilog examples*, *Wiley-Interscience, New York, NY, USA*, (2008) .

[2] https://github.com/lpacher/lae