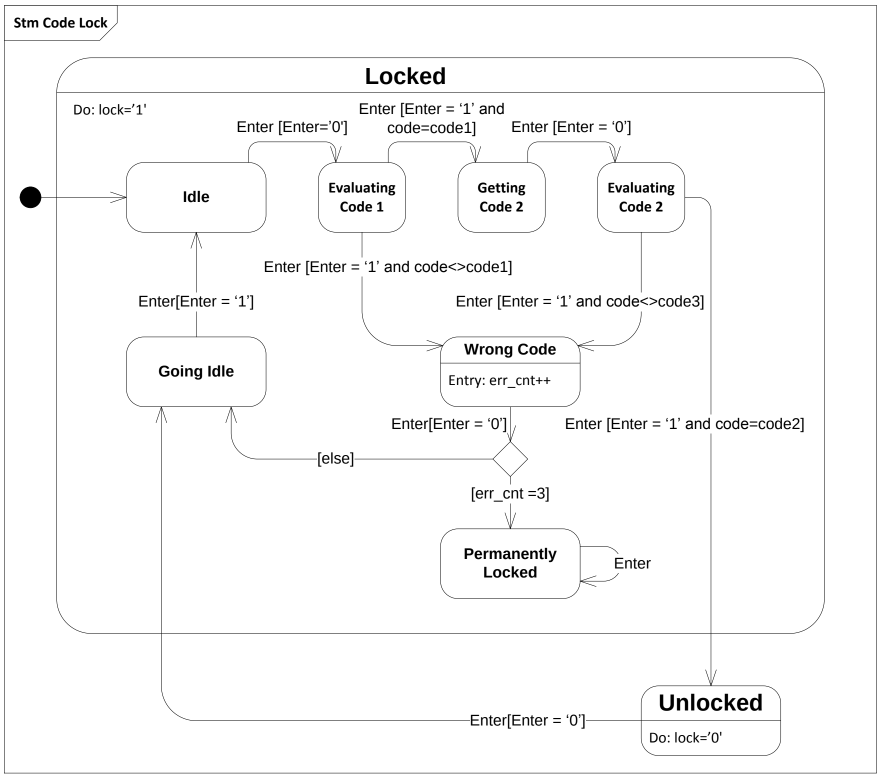
DSD Exercise



Finite State Machines in VHDL

(The Code Lock (or UART) Exercise)

8-12-2015/PHM

GOALS

State machines are essential in most digital systems. They lurk many places: In protocols, in control systems, as the main building block of operating systems and so on. In this exercise you will work with two basic state machine designs: Mealy and Moore. For those of you who have only seen the state machine diagrams, it's finally time to implement!

You must complete exercise 1, plus exercise 2- or 3 to pass this exercise.

The goals for this exercise are:

- To master the 3-Process State Machine Template
- To enhance your VHDL skills
- To decipher state machine diagrams and implement them

PREREQUISITES

- Have Quartus II up and running
- That you have read THE DSD EXERCISE GUIDELINES!!!

THE EXERCISE ITSELF

You will start out with a basic Mealy / Moore finite state machine and then progress into more complicated versions that implement a code lock or a UART interface.

1) MEE-MOO – A COMBINED MEALY / MOORE STATE MACHINE

We'll start out with a basic state machine that implements both a Mealy and a Moore state machine. Bear in mind that it is only the output process that differ between the two. The "next state" process handling the state transitions is the same!

1) Implement the state-diagram for the MeeMoo state machine illustrated in Figure 1 using the three process template in the book. The diagram has both Moore and Mealy outputs with conditional and unconditional transitions. In other words: if you can solve this exercise you will actually be able to implement VHDL code for both state machine types! Use the interface described in Figure 2. Compile your design in Quartus II and view the output in the Netlist Viewers->State Machine Viewer. *How does it compare with your design?*

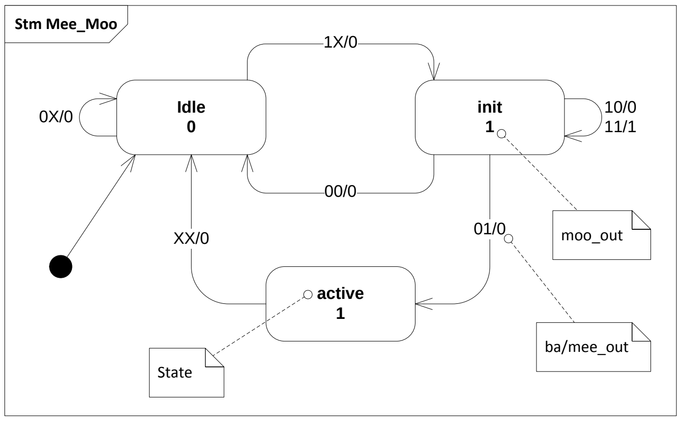


FIGURE 1 MEE MOO STATE MACHINE DIAGRAM

Hint! Start with just looking at the Moore output (Moo_out) and implement the state machine to do this. When it works, create a Mealy output process to generate the Mealy output (Mea_out). In VHDL, create a new "state" type to reflect the three states: "idle", "init" and "active (See section 10.4 in the book). This will make your code more readable and state names will be visible in the simulation waveform tool.

2) Create a functional simulation of your state machine. Add the pins, but also add the next_state signal in the waveform editor: Insert signals->node finder->Filter:Design Entry (all names)->List ->(pick the one with the type StateMachine) and add it to the waveform. This latter one will show you which state you are in. *How does it compare with the state machine view? When do the mealy and moore outputs change in relation to the clock?*

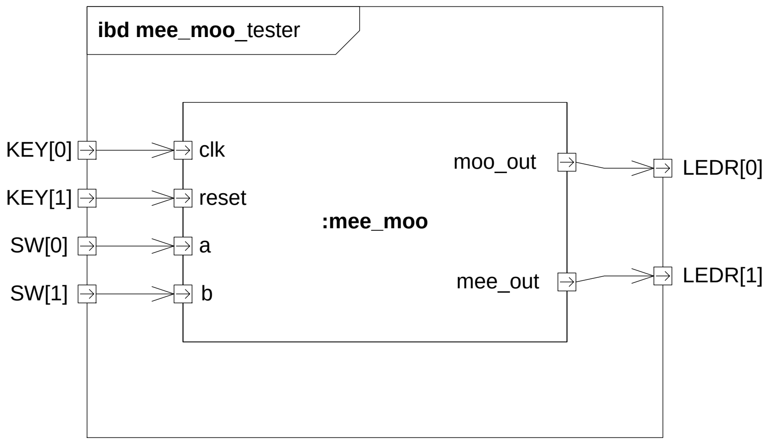


FIGURE 2 MEE MOO TESTER IBD

3) Build the tester and test your design on the DE2-board.

2) Code Lock (Skip this if you'd rather build the UART)

The purpose of this exercise is to implement a code lock as a finite state machine. The code-lock functionality can be described as:

The code lock accepts a sequence of two correct codes, each followed by an "enter" press, to let the "lock" output change from locked ('1') to unlocked ('0'). The code lock remains unlocked until "enter" is pressed again. Three wrongly entered codes will lock the code lock permanently until reset is asserted.

The two secret codes are hard-coded into the VHDL program.

In solving the exercise you must consider and explain, in text, how the listed SysML terms correspond to Moore/Mealy terms and explain how they can be implemented in VHDL using snippets from your code:

- SysML triggers, guards and effects (no effects in exercise)
- SysML do actions
- SysML entry actions (err_cnt++ in "Wrong Code")
- SysML exit action (None in the exercise)

Note that entry- and exit actions are properties of a state, but can also be interpreted as common effects on the in- and outgoing transitions.

1) Start out by implementing the simple version of code lock described in Figure 4 using the interface described in Figure 3. Use the three-process template to implement the state register, next state- and output process for the code lock.

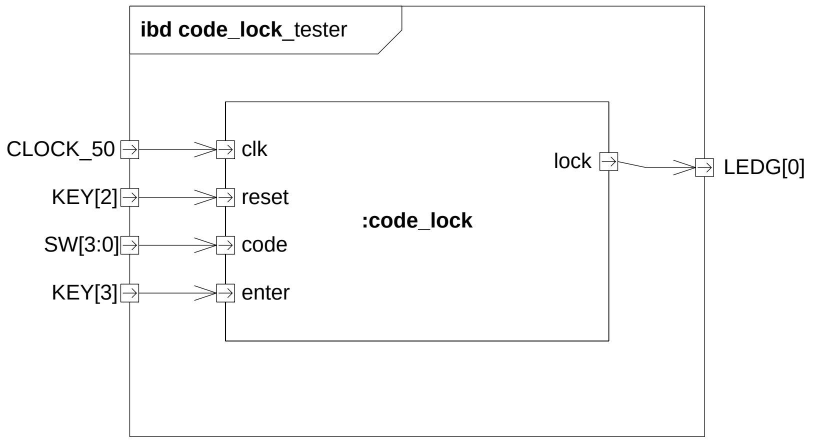


FIGURE 3 CODE LOCK TESTER IBD

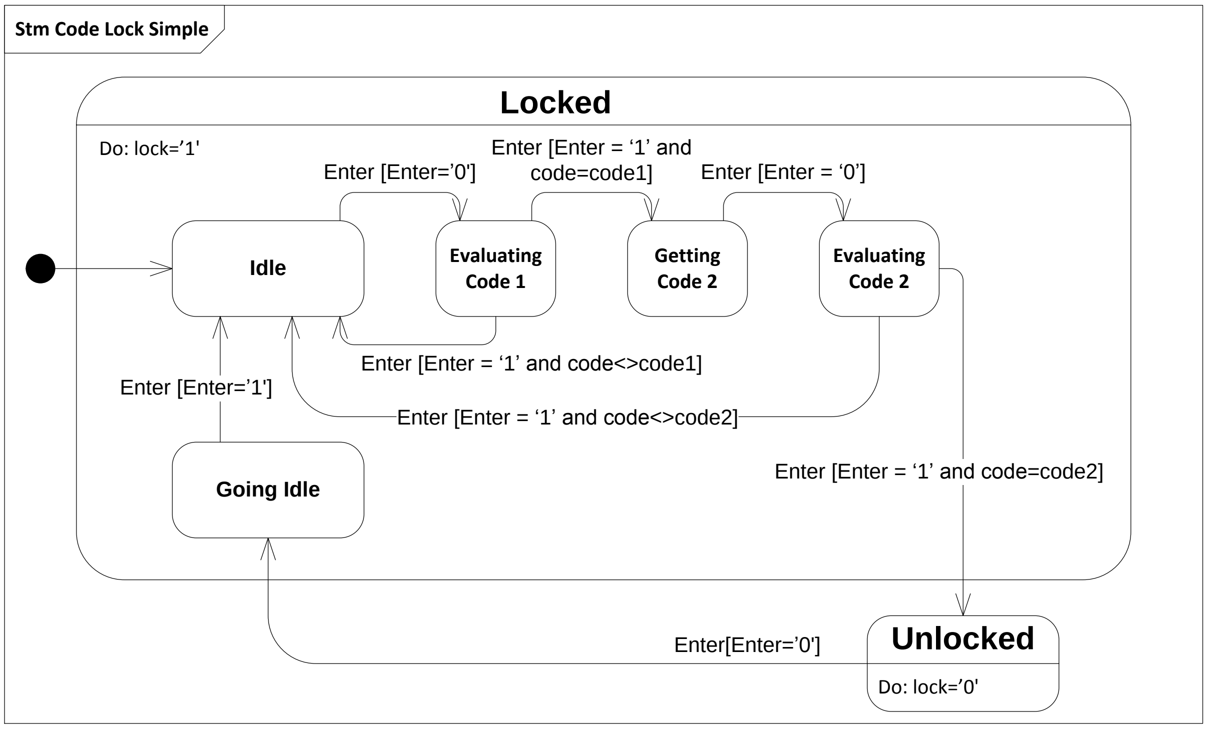


FIGURE 4 CODE LOCK SIMPLE STM

2) Create a functional simulation of your design, just as you did with the Mee-Moo state machine. *Does the state change happen as expected? Add simulation waveform to your journal*

3) Create a tester and test your design on the DE2-board.

We will now add functionality to limit the number of wrong entries to three and put the code lock in a permanently locked condition if this is exceeded. To do this, we will have to add two new states in the code lock as shown in Figure 5.

4) Add the two new states: "wrong code" and "permanently locked" to the code lock state machine. Let the state machine pass directly from "Wrong code" to "permanently locked" for now. Simulate to verify, that you end in "Permanently Locked" I you enter a wrong code.

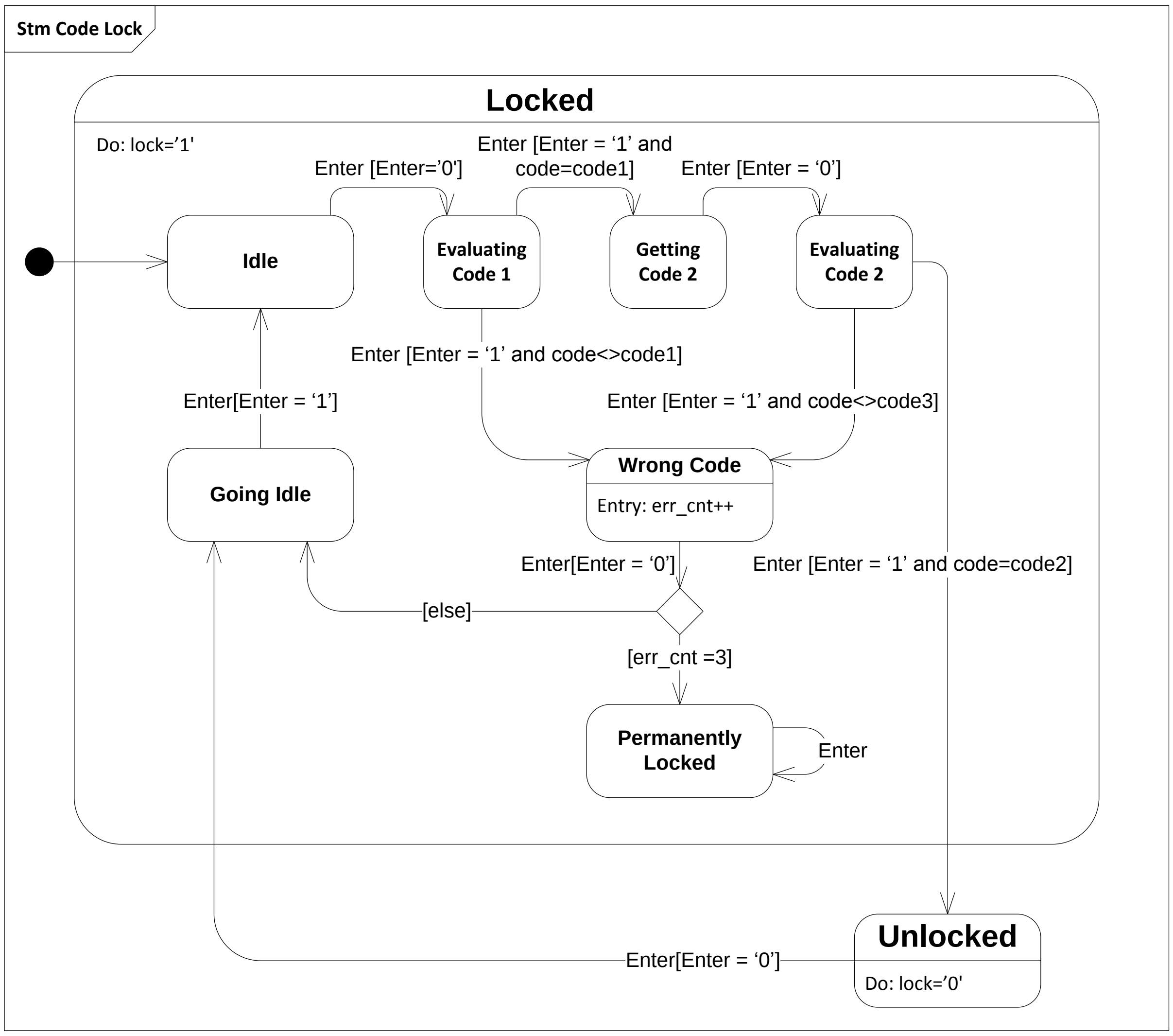
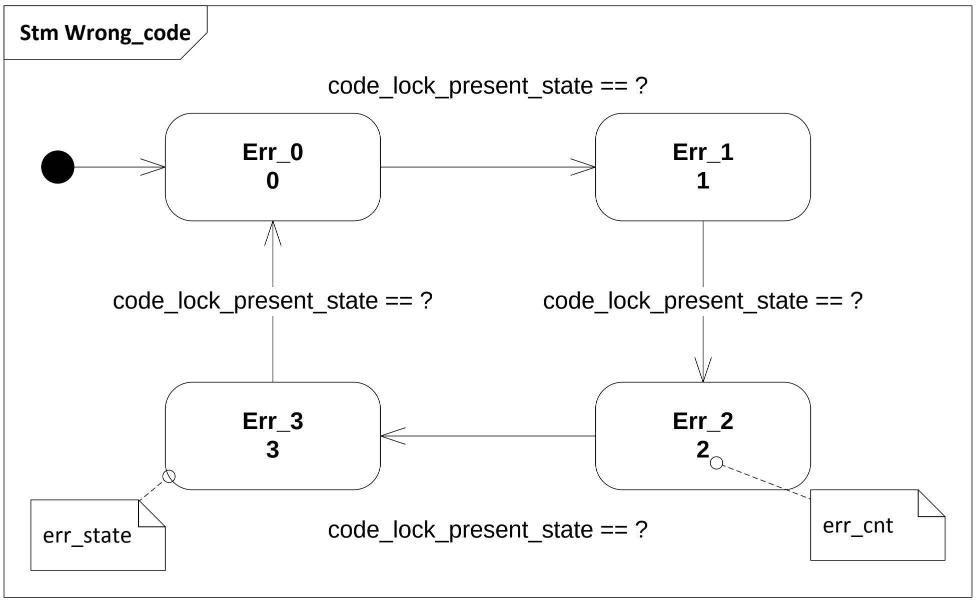


FIGURE 5 CODE LOCK STM

A counter is actually a very simple Moore state machine (See VHDL for engineers section 10.9). To count a value in ”Wrong State” we are actually introducing a state machine in a state machine aka a sub-state machine!

5) Create a new 3-process state machine in your code_lock design file. This state machine must implement the Moore machine described in Figure 6. This state machine must be triggered as we enter a certain state in the code_lock state machine, *which one?*
6) Modify your “wrong code” state to check if the err_cnt value is above a certain threshold to set the next state to “permanently locked” otherwise set it to “idle” as shown in Figure 5



Stm Wrong_code
code_lock_present_state == ?
Err_0
0
Err_1
1
code_lock_present_state == ?
code_lock_present_state == ?
Err_3
3
Err_2
2
err_state
code_lock_present_state == ?
err_cnt

FIGURE 6 WRONG CODE STM

7) Simulate your state machine to verify is functionality. *Put waveforms with state transitions in your journal.*
8) Build your design with Quartus II and test on the DE2-Board. *Did you have any latches? If so, why?*

3) UART (Optional If You have Completed 1+2)

The purpose of this exercise is to construct a UART controller that allows us to transmit data between the DE2 board and a secondary device with UART interface, such as a PC (with UART adapter), the STK500 board or similar. The technical goal of this exercise is:

To receive one byte of data and show its content on LEDR[7:0]or two 7-segment displays and to transmit one byte of data defined by the switches, SW[7:0], when the push-button KEY[3] is asserted.

Hardware Assembly

As before mentioned, we'll create a UART interface in the FPGA and it to a secondary UART device using wires connected to the DE2 board's GPIO ports. We will also add connections that will allow us to debug the interface with a serial protocol analyzer, such as the Analog Discovery kit. The overall hardware is shown in Figure 7.

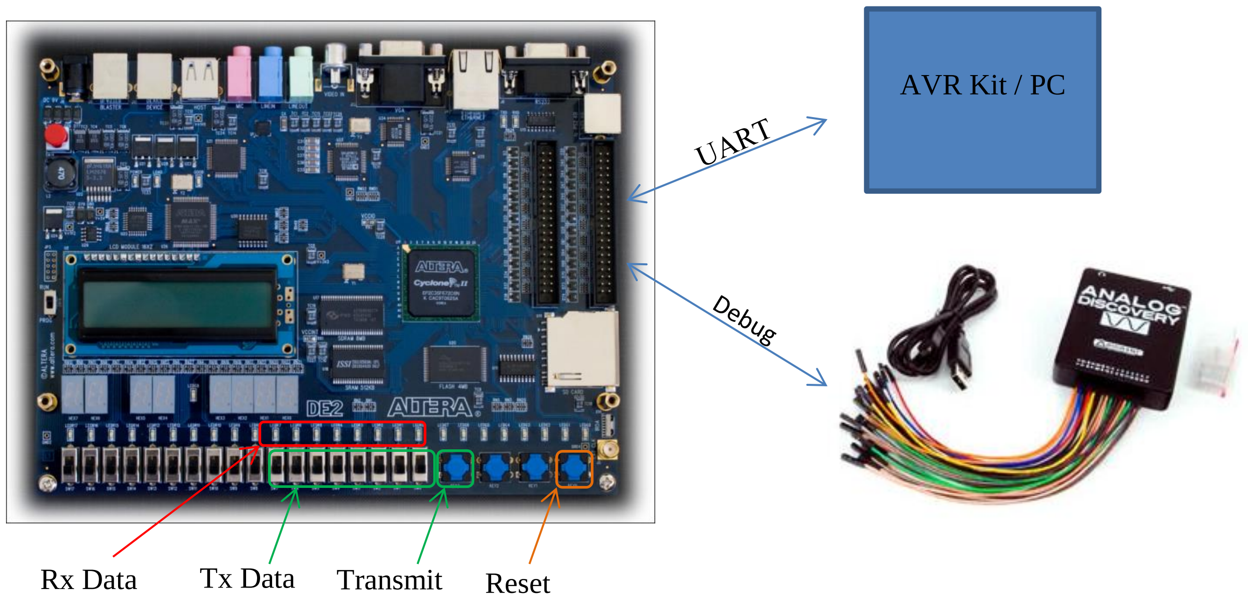


FIGURE 7 UART EXERCISE HARDWARE AND INTERFACES

Design

The design must consist of two design entities:

- uart_tester – That instantiates the UART component and interfaces it to switches and displays
- uart – The UART itself

The UART

Universal Asynchronous Receiver Tramsmitter is a basic serial communication scheme commonly used for embedded devices. It is asynchronous in the sense that no clock is transmitted with the data, the transmitter and receiver must use a similar time reference ex. 115200 Hz, known as the baud rate. The receiver waits for a start bit (line changes from logical '1' to logical '0'), and from here it samples the input at the baud rate agreed upon. The example given in Figure 8 illustrates the communication of two data bytes.

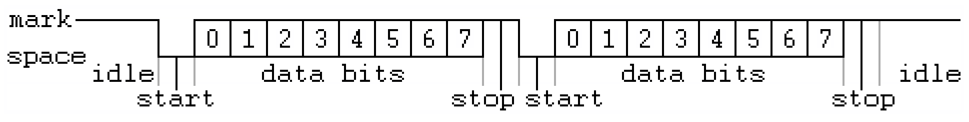


Figure 8 UART Communication of two data bytes (wikimedia.org)

The UART controller must consist of three blocks as shown in Figure 9. The blocks does not have to be design entities, but may just as well be fit into a single entity, the uart.

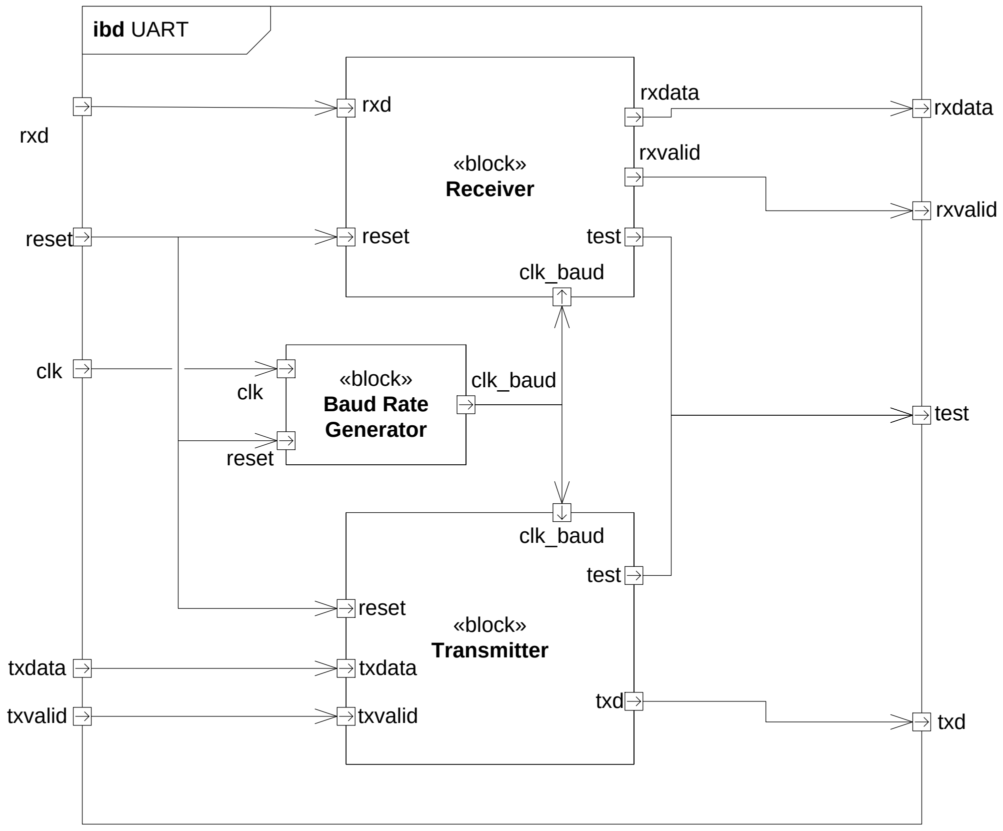


FIGURE 9 UART IBD

The UART is very basic and storage of received data must be handled externally. The external interface of the UART is described in table 1.

Name	Direction	Description
clk	in	50 MHz Clock input
reset	in	Asynchronous reset
rxd	in	UART data input
txdata[7:0]	in	Data byte to be send via UART Txd
txvalid	in	txdata is to be send when txvalid = ’1’
txd	out	UART data output
rxdata[7:0]	out	Data byte received via UARD Rxd
rxvalid	out	Is ’1’ when rxdata is valid
Test (optional)	out	Connections to be used for debugging on the DE2 board. Could be connected to the Analog Discovery kit to monitor UART transmission

Table 1 UART Interfaces

Baud Rate Generator

This block must generate a baud rate “clock” signal, derived by using a counter to divide the 50 MHz clock down to a suitable baud rate, ex. 115200 Hz.

UART Receiver

Investigating the UART interface, we can derive the state diagram in Figure 10

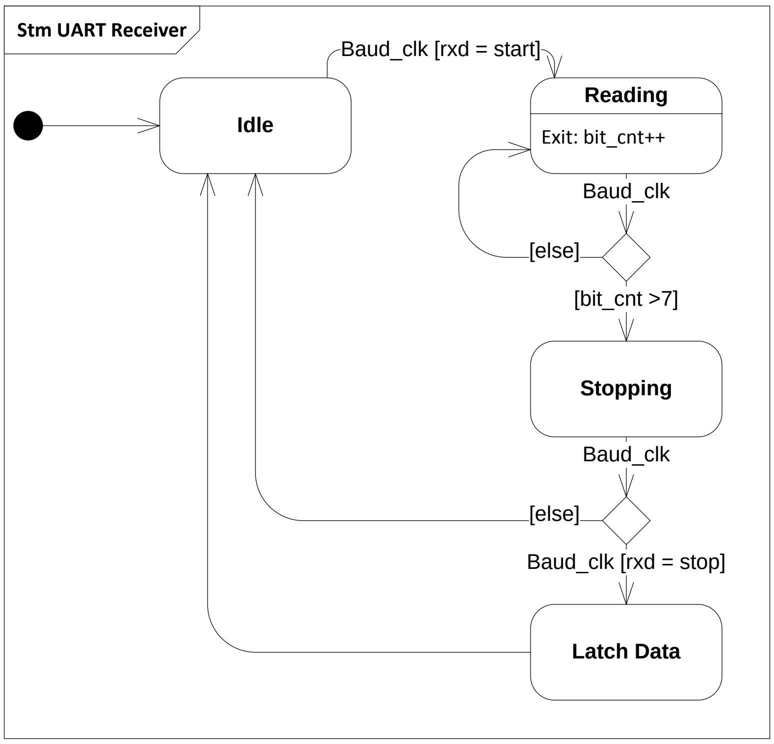


FIGURE 10 UART RECEIVER STATE DIAGRAM

We’ll remain in *Idle* until a start bit I received, then we’ll continue to *reading*, where rx data is shifted in. When the bit counter exceeds 7, we proceed to *Stopping* where a check is made, that a stop bit is received, before latching rx data and returning to *idle*. This sequence is also illustrated in the timing diagram of Figure 11. Note that the clock used in the simulation is only high one 1/50M period, where yours may have a different duty cycle..

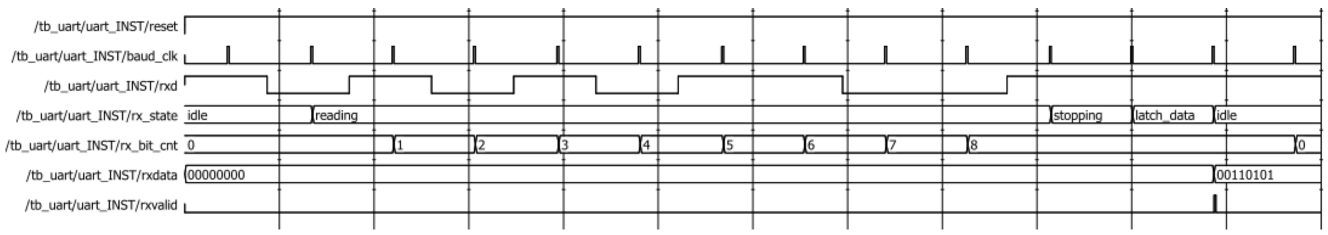


FIGURE 11 UART RECEIVER TIMING DIAGRAM

UART Transmitter

Is similar to the receiver, it must however, shift data out rather than in… Number of states is not nescessarily the same…

The UART Tester

To test the UART on the DE2 board we'll create a UART tester, which will become our new top-level design entity. This component will map the UART to the appropriate ports and it must contain a register to hold the received data byte. The tester IBD is illustrated in Figure 12.

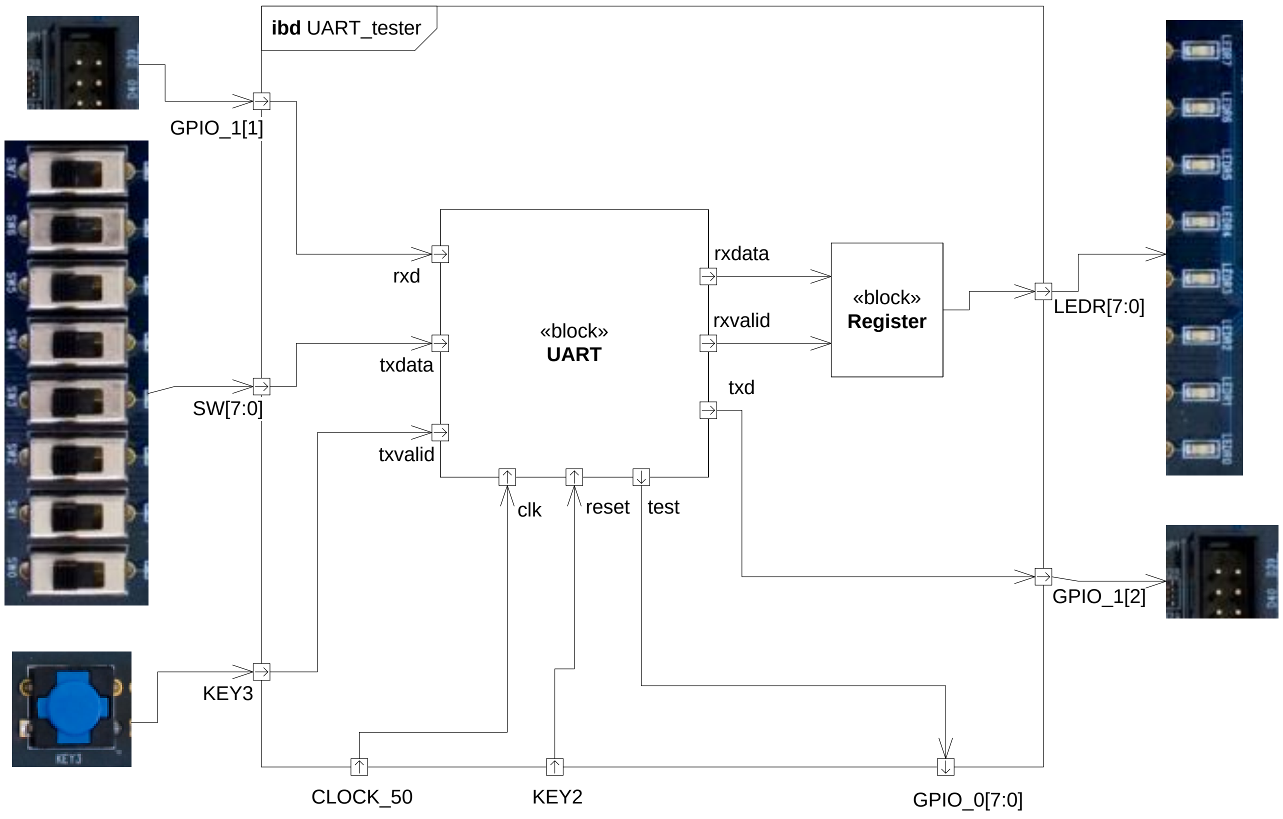


FIGURE 12 UART TESTER IBD

Implementation

1) Start out with the baud rate generator. This should be similar to work you did when creating the watch…
2) When it is working, look at the receiver. Start out by looking at the state- and timing diagrams in the description.
   When using the bit counter note that it is actually a sub-state of the reading state. This is described in the code lock exercise step (5). You may use a counter directly, just remember that you will need to store the present_bit_cnt using the state_register, as the output- and next state processes do not have a clock at therefore will infer a latch to store a value (ex: bit_cnt <= bit_cnt + 1 must be bit_cnt_next <= bit_cnt_present + 1 in output process)
3) Test by connecting to the AVR kit by using a USB/UART adapter (provided by exercise tutor). Use you Analog Discovery Kit to monitor and debug UART data as well as state transitions. Remember to put waveform dumps into your journal.
4) When the receiver is working it is time to design the transmitter. Look carefully at the UART timing diagram and consider which states are required. When the state machine has been designed, its quite straight forward to implement it. Remember to put your design and considerations in the journal!