Serial Receiver

Labor Digital Design

Contents

1 Goals 1
2 Serial Receiver 2
  2.1 Serial Transmission 2
  2.2 Circuit 2
3 Finite State Machine (FSM) 3
  3.1 FSM Elements 3
  3.2 States and Transitions 4
  3.3 Action code, Transition statements and Expression builder 4
4 Implementation 6
  4.1 Analyse 6
  4.2 Development 6
  4.3 Simulation 7
5 Checkout 8
Glossary 9

1 Goals

This lab aims to practice the design of synchronous digital circuits using a FSM. It focuses on the implementation of a serial RS232-type receiver, including:

- Understanding the principles of asynchronous serial communication.
- Designing and implementing a shift register and a clock divider counter.
- Creating an FSM in HDL Designer to control the reception process.

Through this lab, students will gain hands-on experience in combining sequential logic (FSM, counters) with data handling (shift registers) to realize a complete communication module.

The functionality of the receiver will be verified by decoding a message sent by the Xilinx PicoBlaze μProcessor developed in previous labs.

2 Serial Receiver

2.1 Serial Transmission

A serial receiver is a digital circuit that converts an incoming asynchronous serial data stream into parallel data suitable for further processing. The Figure 1 shows the timing of the serial transmission of a data word, where the data is transmitted one bit at a time, starting with a start bit, followed by data bits (Least Significant Bit (LSB) first), and ending with one stop bit.

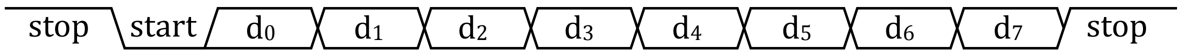


Figure 1 - Serial transmission

2.2 Circuit

The core of the design in Figure 2 consists of:

- A shift register to collect the incoming bits and convert them to parallel format.
- A counter to divide the system clock and synchronize with the baud rate of the incoming data.
- A finite state machine (FSM) to control the reception process, detect the start bit, sample each data bit at the correct time, and validate the stop bit.

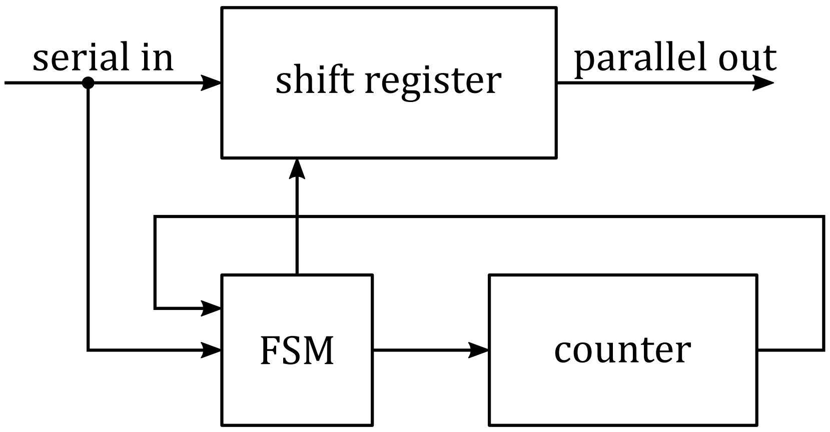


Figure 2 - Block diagram of the serial receiver

The FSM monitors the serial input line, detects the transition indicating the start of a frame (start bit), and triggers regular sampling pulses via the counter. Each sampled bit is shifted into the register until the complete byte is received. The output is then made available in parallel form for subsequent stages of the system.

Investigate the existing elements and signals in the bloc **`COM/serialPortReceiver`**.

3 FSM

In the bloc **`COM/receiverController`**, a FSM will be implemented to control the reception of serial data. The FSM will manage the timing of the sampling of incoming bits, ensuring that each bit is read at the correct time relative to the baud rate of the incoming data stream.

3.1 FSM Elements

Multiple elements are aready available they are color coded in Figure 3.

1. Definition of the clock signal and its triggering condition. In this case, the clock event occurs on the rising edge of the **`clock`** signal.
2. Definition of the reset signal, reset condition, and the state after reset. In this case, the reset signal is named **`reset`**; when it is high, the system resets and transitions to the **`startup`** state.
3. The recovery state used when there is no other valid state assignment.
4. States of the FSM. Each state requires a unique name. The code on the right is executed while in the residing in that state (*code is optional*). In this case:

```
restartCounter <= '1';
--    ^          ^   ^ ^
--    |          |   | +-- Required end symbol of a statement
--    |          |   +---- Value to assign to the signal
--    |          +-------- Assignation symbol (be careful <= not =)
--    +------------------- Signal to assign a value to
```

Listing 1 - Code executed in the state **`startup`**

5. Transition condition to link states together (*conditions are optional*). In this case:

```
serialIn = '1'
--   ^    ^  ^  ^
--   |    |  |  +-- No end symbol required since this is a condition
--   |    |  +----- Condition to take the transition
--   |    +-------- Comparison operator equals (be careful = not ==)
--   +------------- Left side of the condition is in this case the signal `serialIn`
```

Listing 2 - Condition to take the transition between the states **`startup`** and **`idle`**

It can be translated to "if the signal **`serialIn`** is equal to **`'1'`**, then the state will change".

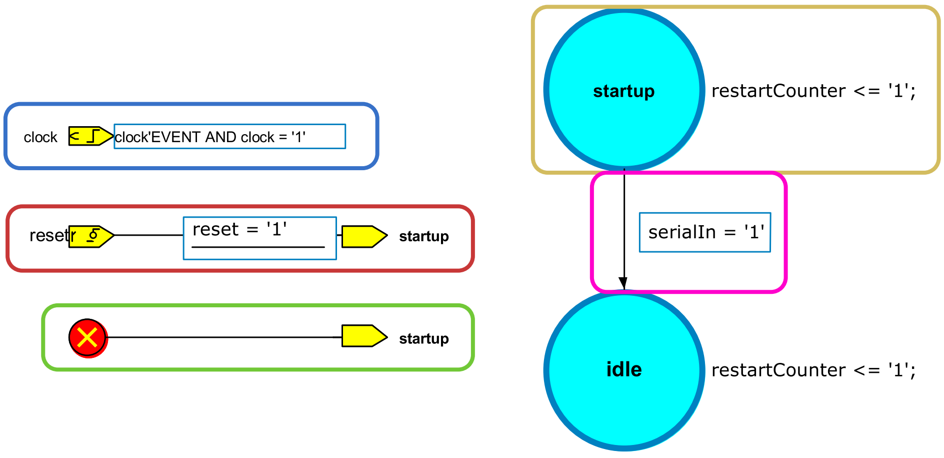


Figure 3 - Elements of the given FSM

3.2 States and Transitions

The FSM will have multiple states, each representing a specific phase of the reception process. The transitions between these states will be triggered by events defined in the arrows. If the arrow has no trigger it means that the transition is always active and at the next clock cycle jump in any case to the next state.

On top of the editor are serveral toolbars with all required functions to edit the FSM. The first toolbar Figure 4, which allows you to create new states (blue circle) and transitions (black arrow).

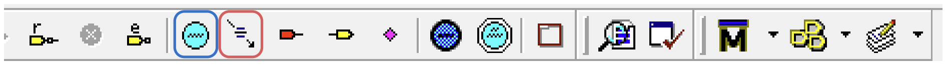


Figure 4 - Toolbar to create new states or transitions in a FSM

3.3 Action code, Transition statements and Expression builder

For all actions or transition conditions you have to write Very Highspeed Integrated Circuit Hardware Description Language (VHDL) code. For this labo we only need to write simple if conditions for the transitions, see Listing 2 or simple signal assignments for the states, see Listing 1.

The second toolbar contains the button to open the expression builder Figure 5, it allows you to create new expressions for the conditions of the transitions and code within the states.

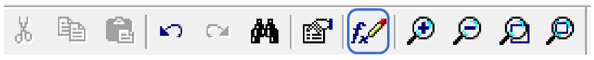
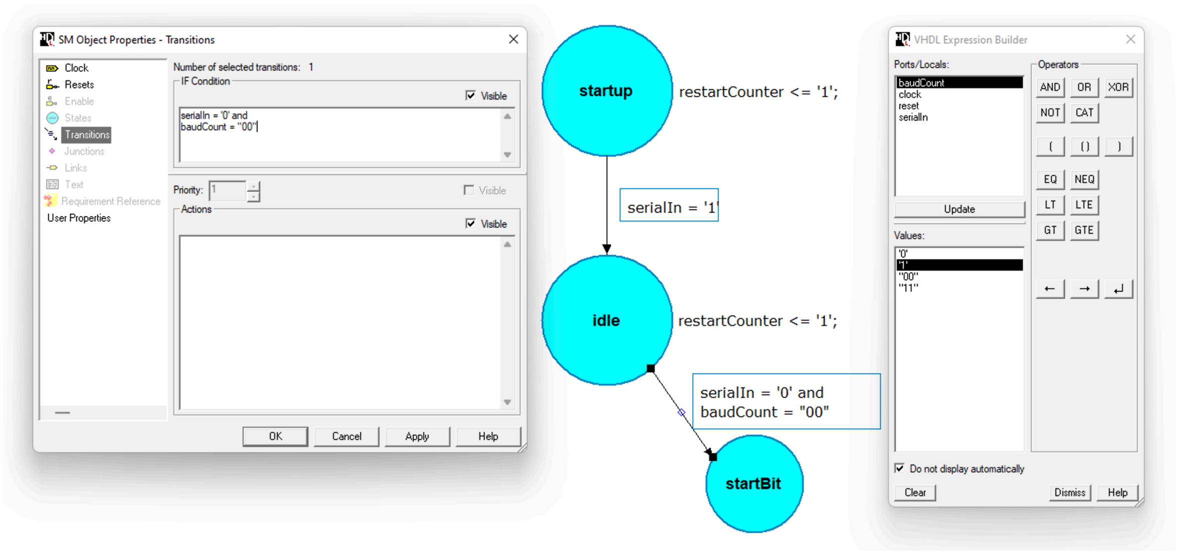


Figure 5 - Toolbar open up the expression builder

To edit a state action or a transition condition, you can double-click on the state or transition in the editor. This will open a code editor where you can write/ edit the VHDL code for the action or condition.



```
serialIn = '0' and
baudCount = "00"
```

Figure 6 - Code editor for a state action or transition condition

A comparison with a single bit signal such as **`serialIn`** or a multiple bit signal such as **`baudCount`** is slightly different.

1. First of all any comparison is won't reuqired two equal signs **`==`** as in other programming languages, but only one **`=`**.
2. The second difference is that the comparison value for single bit signals is enclosed in single quotes **`'`** and for multiple bits in double quotes **`"`**.
3. With boolean operators such as **`and`**, **`or`** and **`not`** you can combine multiple conditions together.

For example, to check if the serial input is low and the baud count is zero, you can write:

```
  serialIn = '0' and baudCount = "00"
--\____________/  ^  \_______________/
--        |       |            +-- Comparison of a multibit signal
--        |       +------------- boolean operator
--        +---------------------- Comparison of a single bit signal
```

4 Implementation

The three main components of the serial port receiver to be implemented are:
- **`COM/shiftRegister`** for collecting the incoming bits and converting them to parallel format.
- **`COM/baudrateCounter`** for dividing the system clock and synchronizing with the baud rate of the incoming data.
- **`COM/receiverController`** for controlling the reception process, detecting the start bit, sampling each data bit at the correct time, and validating the stop bit.

4.1 Analyse

In order to precicely control the reception of serial data, the timing of the received bits are crucial.

Launch a simulation of the **`COM_test/serialPortReceiver_tb`** with the file **`$SIMULATION_DIR/COM.do`**
- Determine the number of clock periods required by the testbench to send a single bit by analysing the signal **`serialOut`**.
- Determine the number of bits required for the counter **`COM/bauderateConter`**.

4.2 Development

With all the informations gathered in the previous section, you can now implement the three main components of the serial port receiver.

Implement the shift register **`COM/shiftRegister`**. The 8 bits of serial data per frame are output in parallel format. During each impulse of the **`shiftEn`** signal, the next bit is shifted into the shift register.

Implement the counter **`COM/baudrateCounter`**.
- Do not forget to validate/change the number of bits of the signal **`baudCount`** on the **`COM/serialPortReceiver`**.
- The counter need to support a synchronous reset with the **`restartCounter`** signal.
- The data has to be sampled as close to the middle of the bit period as possible.

Implement the finite state machine **`COM/receiverController`**. It needs to detect the beginning and ending of a packet and generates the signals **`restartCounter`** as well as **`shiftEn`** accordingly.

4.3 Simulation

Once implemented, simulate the three components with the testbench **`COM_test/serialPortReceiver_tb`**. The μProcessor Xilinx Picoblaze developed in the previous labors will be used to send the serial data to the **`COM/serialPortReceiver`**.

Simulate the testbench **`COM_test/serialPortReceiver_tb`** with the simulation file **`$SIMULATION_DIR/COM.do`**.
- Validate the send and the decoded data.
- Calculate the Baud rate of the serial transmission.

5 Checkout

This is the end of the labo, you have successfully built a system with several different blocks. Before leaving the laboratory, ensure you have completed the following tasks:

- [ ] Theory
  - [ ] You understand how to create FSM in HDL Designer.
- [ ] Circuit Design
  - [ ] The three blocks **`COM/shiftRegister`**, **`COM/baudrateCounter`**, and **`COM/receiverController`** have been created and tested.
- [ ] Simulations
  - [ ] The data sent by the **`COM/nanoprocessor`** is correctly read and parallelized by the **`COM/serialPortReceiver`**.
  - [ ] The individual bits are read as close to the middle as possible.
- [ ] Documentation and Project Files
  - [ ] Ensure all steps (design, simulations, comparison) are well documented in your lab report.
  - [ ] Save the project on a USB stick or in the shared network folder (**`\\filer01.hevs.ch`**).
  - [ ] Share the files with your lab partner to ensure continuity of work.

Glossary

***FSM* – Finite State Machine** 1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 8

***LSB* – Least Significant Bit** 2

PicoBlaze: PicoBlaze is a small, 8-bit microcontroller designed by Xilinx for use in FPGAs. It is often used in educational settings to teach basic microcontroller concepts. 1

***VHDL* – Very Highspeed Integrated Circuit Hardware Description Language** 4, 4