ADS I Class Project **WS 2025/26**
Hardware Design with Chisel
Prof. Dr.-Ing. Wolfgang Kunz RPTU Kaiserslautern-Landau
M.Sc. Tobias Jauch Fachbereich Elektrotechnik und Informationstechnik
Andro Mazmishvili Entwurf informationstechnischer Systeme

Assignment 1 Warm-Up

This class project's main focus is to give you an insight into digital design and modern hardware description languages. As ADS I is part of the EIT undergrad course as well as the ESY and EMECS master programm, your prerequisites may differ significantly based on which program you are in. Therefore, in addition to hardware design, you might also get in touch with object-oriented programming, version control with Git and some other things for the first time. If all of this doesn't ring a bell, don't worry. The class project material also provides some bootstrapping for everything you need. Nonetheless, it might be helpful to invest some time to learn about object-oriented programming and how to use Git and github, as they are standard concepts and tools in today's research as well as in industry and will also help you with other courses and projects during your studies.

Task 1.1 Prerequisits

Read the *Chisel Introduction* document and work through the necessary steps in order to have a working Chisel environment on your PC.

Task 1.2 Half Adder

Take a look at the half adder presented in the lecture. It is used to add two 1-bit numbers (a, b) and calculate the sum (s). In case the result does not fit into a single bit result, the carry-out (c_o) output signalizes an overflow. The truth table is shown in table 1.

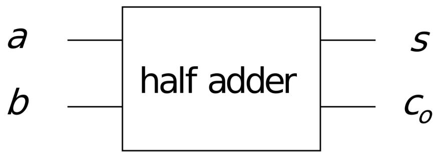

Figure 1: Half adder

a	b	s	c_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1: Half adder

In the repository that we used for the introduction, you can find a file named *Adder.scala* in the main source folder. This file already gives you the basic structure for the components that you are implementing during this warm-up. Your first task is to implement a basic half adder as presented in the lecture. Each signal should only be one bit wide (inputs and outputs). There should be no delay between input and output signals, we want to have a combinational behaviour of the component.
The test folder contains the skeleton of a *Half Adder Tester*. Use the truth table (table 1) to test all possible input combinations and the corresponding results exhaustively.

Task 1.3 Full Adder

We now move on to the full adder that has also been discussed in the lecture. It is able to add two 1-bit numbers (a, b). In addition it also uses the carry bit of a previous calculation as a third input (c_i) to calculate the sum (s). In addition to the sum, a full adder also has a carry output (c_o). The truth table is shown in table 2.

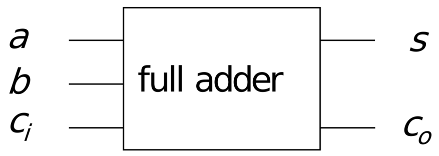

Figure 2: Full adder

a	b	c_i	s	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 2: Full adder

Your task is to implement a basic full adder. The component's behaviour should match the characteristics presented in the lecture. In addition, you are only allowed to use **two half adders** (use the class that you already implemented) and **basic logic operators** (AND, OR, ...). Each signal should only be one bit wide (inputs and outputs). There should be no delay between input and output signals, we want to have a combinational behaviour of the component.
The test folder also contains the skeleton of a *Full Adder Tester*. Use the truth table (table 2) to test all possible input combinations and the corresponding results exhaustively.

Task 1.4 4-bit Adder

In order to process more complex (unsigned) numbers with multiple bits, a chain of full adders with one half adder in the beginning can be used. Each full adder uses one bit of each number as inputs (a_i, b_i) and also takes the carry-out of the previous stage into account ($c_{in,i}$) to calculate the respective bit of the result (s_i). As the LSB do not get a carry from a previous stage, a half adder is sufficient for the first addition. The entire circuit's outputs are the individual bits for the sum (s_{n-1} through s_0) and a global carry bit ($c_{o,i}$), denoting whether an overflow happened.

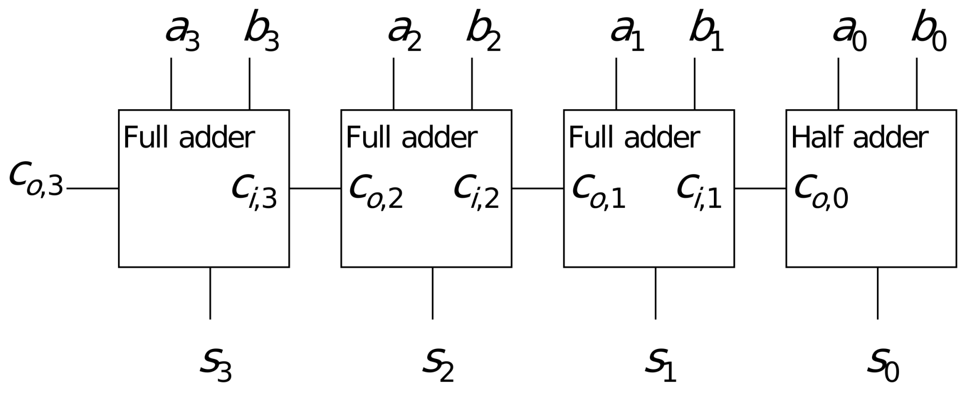

Figure 3: 4-bit adder

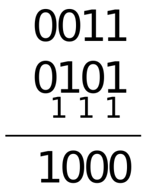

Figure 4: Example for a "rippling" carry bit

Your task is to implement a 4-bit ripple-carry-adder. The component's behaviour should match the characteristics presented in the lecture. Remember: An n - bit adder can be built **using one half adder and n - 1 full adders**.The inputs and the output (result) should all be 4-bit wide, the carry-out only needs one bit. There should be no delay between input and output signals, we want to have a combinational behaviour of the component.

The test folder also contains the skeleton of a *4-bit Adder Tester*. Truth tables are not very efficient for testing more complex components, as they grow exponentially with the number of input bits. Therefore, we have to think of a more clever way to test the 4-bit adder. To test the *Basic Adder* design in our *Chisel Introduction*, we used loops to generate a sequence of increasing input values testing the design. To generate test cases for the 4-bit adder, you should also start by using two nested loops. To determine the borders of the loop counter, think about the lowest and the highest unsigned integer that you can represent with four bit. To test the result produced by your design, think about what happens to the result in case of an overflow and at which point this can happen. Hint: It might be helpful to check the expected output behaviour for two different scenarios with the help of a condition.

Task 1.5 Serial Receiver

The last part deals with a more complex component. Your task is to design a serial receiver that you might already know from VDS Lab. It scans an input line ("serial bus") named rxd for serial transmissions of data bytes. A transmission begins with a start bit '0' followed by 8 data bits. The most significant bit (MSB) is transmitted first. There is no parity bit and no stop bit. After receiving the 8th data bit, a new transmission may immediately begin with a start bit ('0'), or the line may return to idle ('1'). If there is no new transmission the bus line goes high ('1', this is considered the "idle" bus signal). In this case the receiver waits until the next transmission begins.
If the reset signal goes high, an ongoing transmission is aborted by resetting the counter and setting the valid output to '0'. After a reset, a new transmission begins after the start bit '0' on the rxd-line is received again.

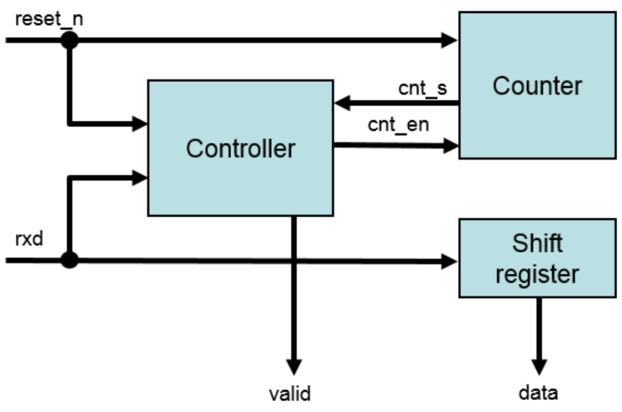

Figure 5: Serial Receiver Design

The outputs of the design are an 8-bit parallel data signal and a valid signal. The valid signal goes high ('1') for one clock cycle after the last serial bit has been transmitted, indicating that a new data byte is ready. Fig. 5 shows the structure of the serial receiver. It includes a bit counter, a shift register and a main controller. Whenever the controller detects a start bit it enables the counter. After 8 data bits have been received the data output is valid.

Writing test cases for this design already becomes quite complex, as infinite sequences of input bits are possible. The test folder contains the basic structure of a *Read Serial Tester*. Design test cases that validate the basic functionality and the (potentially) most error-prone parts of the transmission. If you already attended the VDS Lab, you are also welcome to (additionally) formally verify the generated System Verilog Design of your serial receiver.