Computer Engineering – Submission due 03.11.2020

Jan Vogt, Yannik Köllmann - in gleichen Teilen

30. Oktober 2025

1 Gate-level Modeling

1.1 My First Module

Tasks

1. Implement Eq. (3.1.1) in a module named `my_module` using gate-level primitives. Save the code as `my_module.sv`.

   `(siehe ./src/task3.1/my_module.sv)`

2. Test your implementation using the testbench `my_module_tb`. Save the code as `my_module_tb.sv`. Verify the outputs for all input combinations (see Table 3.1.1).

   `(siehe ./src/task3.1/my_module_tb.sv)`

3. Visualize the resulting waveforms.

   `(siehe ./src/task3.1/my_module_tb_waveform.png)`

1.2 Full Adder

Tasks

1. Implement the module `full_adder`. Exclusively use gate-level primitives in your design.

   `(siehe ./src/task3.2/full_adder.sv)`

2. Test your implementation in the testbench `full_adder_tb`. Your testbench should check the outputs for all possible inputs (see Table 3.2.1) through `assert()` statements.

   `(siehe ./src/task3.2/full_adder_tb.sv)`

3. Visualize the waveform generated by your module. Only show `i_a`, `i_b`, `i_carry_in`, `o_s` and `o_carry_out` of the `full_adder` module in that order.

   `(siehe ./src/task3.2/full_adder_tb_waveform.png)`

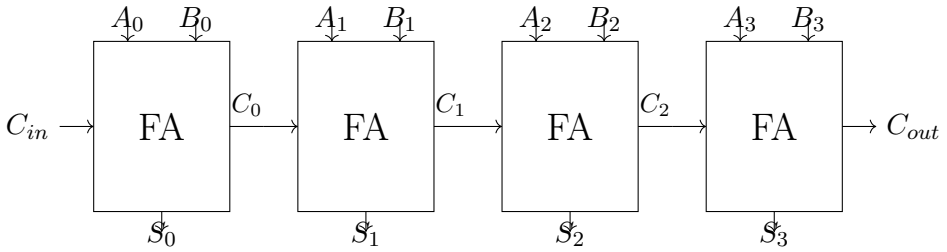
1.3 Four-bit Ripple-Carry Adder

Tasks

1. Draw the schematic of the four-bit ripple-carry adder as a sequence of full adders.

   Schaltbild des 4-Bit Ripple-Carry Adders:

   Der 4-Bit Ripple-Carry Adder besteht aus vier kaskadierten Volladdierern (FA). Jeder Volladdierer addiert zwei Eingangsbits (A_i und B_i) sowie den Carry-Eingang (C_i) und erzeugt ein Summenbit (S_i) sowie einen Carry-Ausgang, der zum nächsten Volladdierer weitergeleitet wird.



2. Derive the total number of possible inputs for the four-bit ripple-carry adder! How would this change if we were to implement a 32-bit ripple-carry adder?

   Anzahl möglicher Eingangskombinationen:

   Der 4-Bit Ripple-Carry Adder hat folgende Eingänge:

   - $A_{[3:0]}$: 4 Bits
   - $B_{[3:0]}$: 4 Bits
   - C_{in}: 1 Bit

   Insgesamt: $4 + 4 + 1 = 9$ Eingangsbits.

   Gesamtzahl möglicher Eingangskombinationen:

   $$N_4 = 2^9 = 512$$

   Für einen 32-Bit Ripple-Carry Adder:

   Eingänge: $32 + 32 + 1 = 65$ Bits.

   Gesamtzahl möglicher Eingangskombinationen:

   $$N_{32} = 2^{65} \approx 3{,}69 \times 10^{19}$$

3. Complete Table 3.3.1!

   Vervollständigte Wahrheitstabelle:

$A_{[3:0]}$	$B_{[3:0]}$	C_{in}	$S_{[3:0]}$	C_{out}
0000	0000	0	0000	0
1010	0001	0	1011	0
1100	0000	1	1101	0
0101	1010	1	0000	1
0111	1100	0	0011	1
1111	1111	1	1111	1

4. Implement the module `ripple_carry_adder_4` by completing the module in Listing 3.3.1.

   `(siehe ./src/task3.3/ripple_carry_adder_4.sv)`

5. Test your implementation in the testbench `ripple_carry_adder_4_tb` by completing the module in Listing 3.3.2. Your testbench should check the outputs for all inputs in Table 3.3.1 using `assert()` statements.

   (siehe ./src/task3.3/ripple_carry_adder_4_tb.sv)

6. Visualize the waveform generated by your module for Table 3.3.1's inputs. Only show the four-bit inputs `i_a`, `i_b`, the carry in `i_carry_in`, the four-bit output `o_s` and the carry out `o_carry_out` of the `ripple_carry_adder_4` module in that order.

   (siehe ./src/task3.3/ripple_carry_adder_4_tb_waveform.png)

1.4 Carry-Lookahead Adder

Tasks

1. Implement `cla_pre_4` (Fig. 3.4.1) and test with `cla_pre_4_tb`.

   (siehe ./src/task3.4/cla_pre_4.sv)

   (siehe ./src/task3.4/cla_pre_4_tb.sv)

2. Implement `cla_logic_4` (Fig. 3.4.2) and test with `cla_logic_4_tb`.

   (siehe ./src/task3.4/cla_logic_4.sv)

   (siehe ./src/task3.4/cla_logic_4_tb.sv)

3. Implement the 4-bit carry-lookahead block `cla_block_4` (Fig. 3.4.3). Use the two modules `cla_pre_4` and `cla_logic_4` in your design. Use the testbench `cla_block_4_tb` for your tests.

   (siehe ./src/task3.4/cla_block_4.sv)

   (siehe ./src/task3.4/cla_block_4_tb.sv)

4. Implement a 64-bit carry-lookahead adder by chaining sixteen 4-bit carry-lookahead blocks (Fig. 3.4.4). Use the testbench `cla_adder_64_tb`.

   (siehe ./src/task3.4/cla_adder_64.sv)

   (siehe ./src/task3.4/cla_adder_64_tb.sv)

1.5 Delays

Tasks

1. Implement the circuit shown in Fig. 3.5.1 in SystemVerilog using gate-level primitives. Save as `module_with_dela`
   Add delays to your circuit.

   (siehe ./src/task3.5/module_with_delays.sv)

2. Demonstrate the behavior with a testbench `module_with_delays_tb.sv`. Show that a glitch appears when changing the inputs from $(A, B, C) = (1, 1, 1)$ to $(A, B, C) = (1, 1, 0)$. Visualize the obtained waveforms.

   (siehe ./src/task3.5/module_with_delays_tb.sv)

   (siehe ./src/task3.5/module_with_delays_tb_waveform.png)