Technische Universität Chemnitz
Fakultät für Elektrotechnik
und Informationstechnik
Professur Schaltkreis- und Systementwurf
Prof. Dr.-Ing. Heinkel

IMPLEMENTING A CRC CODE USING A SPECIFICATION TOOL

Manual

Practical course for the lecture “EDA Tools I”

Contents

1 Introduction

In this exercise, we will focus on modelling a specific CRC code using the formal specification and requirements management tool *SpecScribe*.

To solve the preparation tasks, basic knowledge about CRC calculation and its hardware implementation is provided in section 2. To gain knowledge about SpecScribe, the tool will be introduced in general in section 3.

Your task in the practice session is split into two parts: First create a formal CRC-4 specification in SpecScribe and export it (refer to section 4), then test your model using ModelSim (section 5).

Important: Answer the preparation questions on page 13 **before** the practice and bring the written solutions with you. **If you fail to show the answers at the beginning of the practice, your attendance will be refused!**

2 Basics of the CRC calculation

The Cyclic Redundancy Check (CRC) is one of the most frequently used methods for creating checksums in order to ensure a correct transmission of digital data. A typical transmission flow is done in the following steps: encoding, modulation, channel transmission, demodulation and decoding. In this section, only the encoding process is described.

The CRC is a special class of block codes. A block code y with the length n consists of k data bits u and $(n-k)$ redundancy bits r. A commonly used notation for the shape of a block code is (n, k). The redundancy bits are a representation of the data block and used by the receiver for checking the integrity of the received data. Within the block codes various algorithms for calculating such redundancies are known, e. g. a simple parity bit calculation of an amount of data bits.

The computation of the CRC redundancy bits is based on a polynomial division. For the following CRC operations, it is very helpful to transform a sequence of data bits u with the length k in a polynomial form: $u(x) = u_{k-1} \cdot x^{k-1} + u_{k-2} \cdot x^{k-2} + ... + u_0 \cdot x^0$. Similar to that the redundancy bits are represented by: $r(x) = r_{n-k-1} \cdot x^{n-k-1} + r_{n-k-2} \cdot x^{n-k-2} + ... + r_0 \cdot x^0$. Notice that the variable x defines the order of each bit in the block code and u_i and r_i are 0 or $+1$. The required redundancy bits are defined as the remainder of the polynomial division:

$$\mathbf{r(x) = u(x) \cdot x^{n-k} : g(x)} \tag{1}$$

The divisor $g(x)$ is known as the generator polynomial and specified by various technical standards. One of the most important example is the Ethernet standard, which uses a CRC-32. For this the generator polynomial $g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ is used.

To understand the calculation of the CRC, please look at the following (simpler) example.

Example Here, the calculation of the CRC-4 block of a $(9, 5)$ block code is presented. The data block is given by $u(x) = x^3 + x$ (binary shape: $u = 01010$). Multiplied this with $x^{n-k} = x^4$

and divided by the standard generator polynomial $g(x) = x^4 + x + 1$ the polynomial division is executed as follows:

$$
\begin{array}{rl}
 & (x^7 + x^5) : (x^4 + x + 1) = x^3 + x + 1 \\
-(x^7 & \quad + x^4 + x^3) \\
\hline
 & \quad (x^5 + x^4 + x^3) \\
- & \quad (x^5 \quad\quad + x^2 + x) \\
\hline
 & \quad\quad (x^4 + x^3 + x^2 + x) \\
- & \quad\quad (x^4 \quad\quad + x + 1) \\
\hline
 & \quad\quad\quad (x^3 + x^2 \quad + 1)
\end{array}
$$

The remainder $r(x) = x^3 + x^2 + 1$ describes the redundancy block with $r = 1101$, which is appended to the data block u to get the whole encoded block code $y = 010101101$.

2.1 Hardware realization

After understanding the mathematical calculation, the question has to be answered how to transfer this formal description into a digital circuit. For that, look at the shape of the generator polynomial $g(x) = x^4 + x + 1$. The exponent of the term with the highest order is used to specify the amount of required D flip-flops. In our example four D flip-flops are required (because the highest power in the polynomial is 4).

The feedback is realized by XOR gates and is defined by the remaining term of the generator polynomial. In our example, the remaining term of the generator polynomial is $x + 1$. We can see that the zeroth and 1st power is defined there, but the 2nd and 3rd power is undefined. Therefore, we have to introduce a feedback using XOR gates for the D inputs of flip flop “0” and flip flop “1”. However, the inputs of flip flop “2” and “3” are not connected to an XOR gate. The result is displayed in figure 1.

While the incoming data is a continuous serial stream, the output should be 4 bit in parallel. In order to not lose the track of the whole system, the clock, reset and output signals are not represented in this figure.

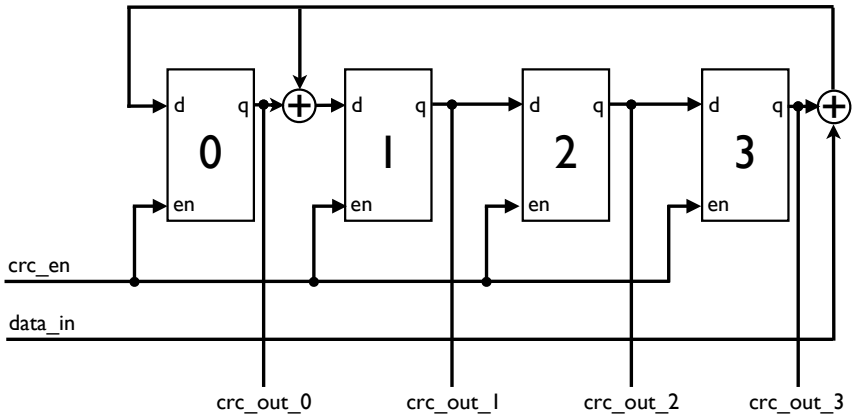


Figure 1: CRC-4 register using generator polynomial $g(x) = x^4 + x + 1$. The highest order term (x^4) defines the number of flip flops. The powers of the remaining term ($x + 1$) defines the XOR feedbacks (here: inputs of flip flop 0 and 1 are attached to an XOR feedback).

Further information about the basics of the CRC-4 can be found at [1, p. 421 ff.]

3 Formal system specification using *SpecScribe*

Various methods are used for the specification of embedded systems. Commonly, a specification is a large document written in natural language by individuals. Specification documents often grow during the specification lifecycle which leads to inconsistencies and the lack of traceability. The natural language causes ambiguities that allow misinterpretation of the specification. An ambiguous specification may lead to interoperability defects at the component and system interfaces in turn.

Tracing and tracking are the key aspects in requirements engineering (RE) since they allow for qualitative and quantitative measures about the project progress. However, the main problem with commonly-used requirement management (RM) tools is that they do not use formal methods for the description of requirements and for tracing. There, requirements are textual descriptions with meta data such as creation date and creator. Trace data is added manually to the requirements.

SpecScribe extends the concepts of RE/RM tools by formal methods. Specification data is captured by pre-defined, but extendable forms and stored in a database. This data can be checked for consistence and completeness and is reused in later design steps, implementation, verification and test.

3.1 Hierarchical system specification

The first step in formalisation is to structure the requirements in a hierarchical order to identify dependencies. In a next step, the textual/graphical requirements need to be formalised which requires differentiation regarding the time continuity domain and analog/digital behaviour.

In typical RM tools, requirements are organised hierarchically, similar to paragraphs in a textual document. In SpecScribe, requirements are assigned to the top level, components or component instances.

As in VHDL or Verilog, we distinguish between components and their instances. A component is a logical entity in a component database, which refers to a subsystem or subfunctionality. If a component shall be used in a target system, it has to be instantiated. These instances are built from components and represent their usage. Furthermore, a requirement is assigned to a component or to an instance.

Components can be implemented in software or hardware using a structured or a behavioural description, depending on which is available. Behavioural components like finite state machines (FSMs) or logic descriptions can be arranged hierarchically in structured components. These subcomponents communicate via signals and allow for system partitioning.

The division of entities into three classes (requirements, components and component instances) lead to a powerful reuse concept since parts of the system can be reused in other specifications. The relations between these classes provide the ability of extended derivation analyses and give the reason why a specific component exists.

3.2 Components

Each component has at least the following attributes:

- name and
- port definitions,

where each port is part of the component's interface. A port can be used to communicate with other components or with the test environment.

3.2.1 Structured components

One possible component type in SpecScribe is the structured component, which is basically a grey-box view on the component. A typical structured component is comprised of several subcomponent instances, whose ports can be connected to ports of the main component or other subcomponents.

3.2.2 Behavioural components

At a certain point in the design, it is not possible to specify the system by instantiating subcomponents. Behavioural models fill this gap. In the digital design, these are usually represented by FSMs, where well-defined component behaviour will occur, depending on the current system state.

In SpecScribe, the behaviour of synchronous and asynchronous state machines and their connections can be described using an extended ADeVA notation [2]. Functional properties of an FSM are described in LTL.

3.3 User interface

The user interface of SpecScribe is divided into two main parts:

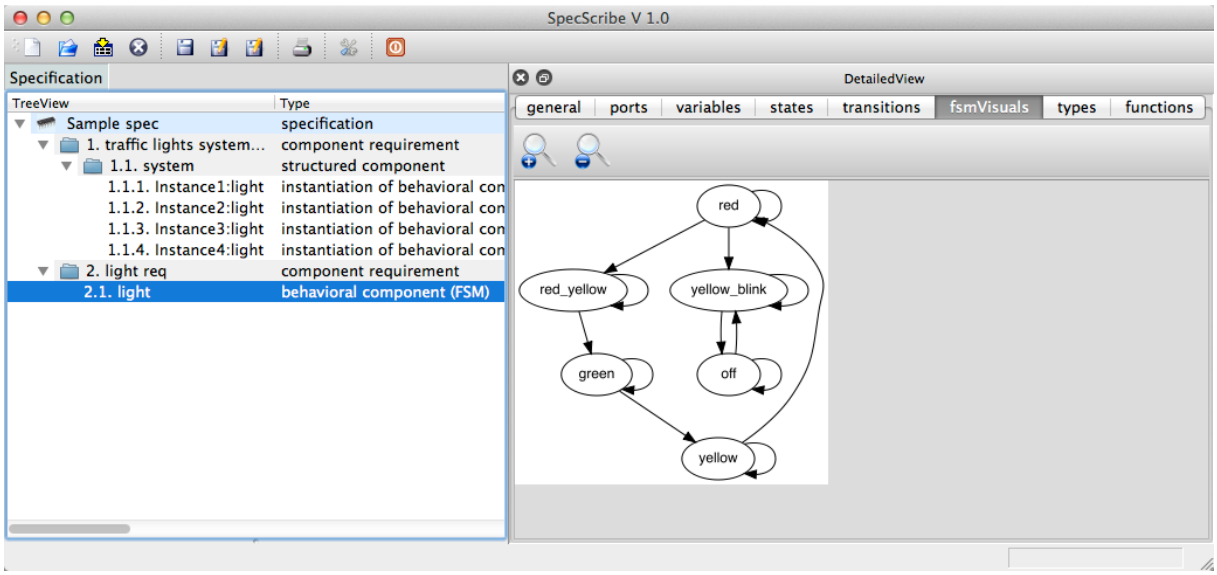


Figure 2: SpecScribe user interface.

- Tree view and
- Detailed view.

The hierarchical system specification is presented by the tree view, while the detailed view contains information about the currently selected tree element.

Figure 2 shows the tool *SpecScribe* after a sample specification has been loaded. There, a simple structured component is defined which contains several instantiations of behavioural models. One behavioural model is currently selected and the FSM graph is shown in the detailed view.

4 Modelling the CRC-4 in SpecScribe

4.1 Prepare your environment

After logging in, start a terminal. Create a working directory with the following commands:

```
cd edat
mkdir spec
```

Copy the predefined files (an XOR VHDL model and a CRC-4 testbench) to the working directory:

```
cp /home/4all/tmp/edat_pract/spec/* spec
```

Now, start SpecScribe:

```
module add specscribe/1.0
specscribe
```

The main SpecScribe window will appear.

Now, create a new specification with an arbitrary name. **Be sure to save your specification from time to time!**

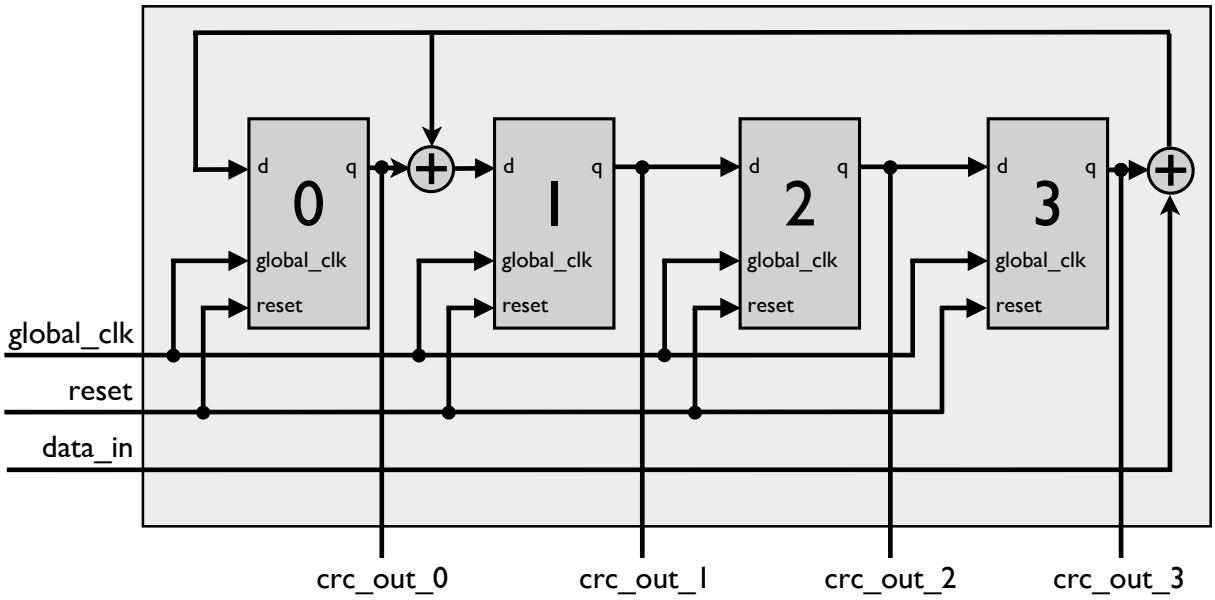


Figure 3: CRC-4 hardware implementation. Input ports are `global_clk`, `reset`, and `data_in`, while `crc_out_x` are output ports of the design.

4.2 The CRC-4 model

The hardware implementation structure of the CRC-4 can be found in figure 3. As can be seen there, two subcomponents are necessary: a flip flop and an xor component. Both have to be modelled first.

4.2.1 Flip flop model

Due to the relationship between requirements and components in SpecScribe, we create a flip flop requirement on the top level. We do so by selecting the specification item in the tree view, right-click it and select "append new requirement". Call it "**flip flop requirement**" and set its type to "component requirement".

A flip flop can be represented by a simple FSM with two states, as shown in fig. 4. Therefore, let's create a behavioural FSM component. To do that, first select the previously created "flip flop requirement", right-click and select "append new component". Name it "**ff_comp**" and set its type to "behavioural component (FSM)". Select the "Sequential" timing behaviour.

Interface. Next, define the flip flop interface. Select the new component and switch to the *ports* tab. Create the following ports (if not already defined):

- `global_clk` (input, clocktype)
- `reset` (input, resettype)
- `d` (input, boolean)
- `q` (output, boolean)

Output buffer. After that, define an output buffer variable "`q_var`' 'on the "variables" tab. Set its type to "boolean" and the initial value to "false".

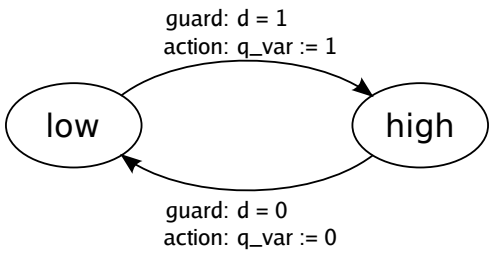


Figure 4: State machine of a D flip flop (input: *d*, output: *q_var*), including transition guards and actions.

Assign the output port (`q`) to the variable (column "PortList").

States. Then, model the two flip flop states as shown in fig. 4. State "low" should be marked as *initial state*.

Transitions. Next go to the "transitions" tab. Create both necessary transitions according to fig. 4. Leave priority -1.

Conditions. We have to model the guard conditions. Add the following condition:

```
d = '1'
```

This will create a new column in the transitions view. The condition will be true if (and only if) the flip flop input *d* is high.

For the transition from *low* to *high*, select "T" (*true*). For the transition from *high* to *low*, select "F" (*false*).

Actions. Finally, we have to update the output buffer variable according to the current state on each rising clock edge. Add the following actions:

- On the *low* to *high* transition: `q_var = '1'`
- On the *high* to *low* transition: `q_var = '0'`

4.2.2 XOR model

The model for the XOR component is predefined in an external VHDL file.

First, create a new top level component requirement called "**xor requirement**". Append a new "behavioural component (external)" and select the "Combinatorial" timing behaviour since the XOR logic does not provide sequential functionality. Call it "**xor_comp**" and choose the file `xor_comp.vhd` from your working directory (~/.SLinux6/edat/spec).

The ports for this component are extracted automatically from the VHDL file. Check that the following ports have been created:

- `a` (input, boolean)
- `b` (input, boolean)
- `c` (output, boolean)

All subcomponents which are necessary for the top level CRC-4 component have been defined now.

4.2.3 Top level CRC-4 model

The top level CRC-4 model is the main component. Create a top level component requirement called "**crc requirement**".

Then, create a child component called "**crc_comp**". It should be defined as a structured component. Select the "SynchronousReactive" model of computation.

Component instances. After selecting the component in the tree view, right-click the item. Then, component instances of the previously defined components (flip flop, xor) can be added to the structural CRC-4 model "crc_comp". Create the appropriate component instances according to fig. 3.

Ports. Then – if not already defined – create the following ports for component "crc_comp" (according to fig. 3):

- `global_clk` (input, clocktype)
- `reset` (input, resettype)
- `data_in` (input, boolean)
- `crc_out_0` (output, boolean)
- `crc_out_1` (output, boolean)
- `crc_out_2` (output, boolean)
- `crc_out_3` (output, boolean)

Connections. Next, go to the "connections" tab of "crc_comp". Here you define the wiring between the top level ports and subcomponents. Each connection has a name and can connect multiple ports. Create all connections that are necessary to implement the circuit shown in fig. 3.

Hint: Seven connections have to be created (excluding the predefined connections for `global_clk` and `reset`). Each connection can connect multiple signals/ports.

4.3 Export to VHDL

Important: Before exporting the specification to VHDL, double-check that all components are defined correctly according to the instructions (ports, variables, states, connections).

Select the "crc_comp" component, right-click it and select "codeExporter" → "VHDL". Select your working directory as export destination (~/.SLinux6/edat/spec). If the specification could be exported successfully, the following files should have been created in the working directory (next to the two predefined files which have been copied to this directory earlier):

- `crc_comp.vhd` (structural CRC-4 model)
- `ff_comp.vhd` (behavioural flip flop model)

5 Testing in ModelSim

Open up a new terminal window and type the following commands. These will change the current directory to the working directory and load ModelSim:

```
cd edat/spec
module add mentor/modelsim/6.6d-64
vsim
```

You should already be familiar with the ModelSim environment. Therefore, its usage will not be explained in detail.

Create a new project and import the necessary files from the working directory:

- `crc_comp.vhd` (structural CRC-4 model)
- `ff_comp.vhd` (behavioural flip flop model)
- `xor_comp.vhd` (predefined behavioural XOR model)
- `crc_comp_tb.vhd` (predefined CRC-4 testbench)

Compile all files. There must be no errors during compilation.

5.1 Simulation

After successful compilation, select "Simulate" → "Start simulation...". **Be sure that "Enable optimization" is not checked!** Select work → crc_comp_tb_tb_crc_comp_cfg and click on "OK". The ModelSim simulation perspective will open.

If not already open, select the "Wave" tab (or open it by "View" → "Wave"), and add all signals from the "Objects" window to the "Wave" view. Simulate 300 ns. The result should look like shown in fig. 5.

Note that the testbench will generate the signals `global_clk`, `reset`, `data_in`, and `correct_crc`. The signal `correct_crc` is generated for verification purposes only and is not used by the design. The signals `correct_crc` and `crc_out` will match for the first 300 ns of the simulation if your CRC-4 implementation is correct. The testbench also checks whether these signals are equal. If this is not the case, red indicators on the timeline will appear, as shown in fig. 6.

References

[1] S. Benedetto, E. Biglieri, and V. Castellani: "Digital transmission theory," Prentice-Hall, Inc, 1987.

[2] S. Gossens, W. Haas, and U. Heinkel: "Semantics of a Formal Specification Language for Advanced Design and Verification of ASICs (ADeVA)," in *11. E.I.S.-Workshop,*, April 2003.

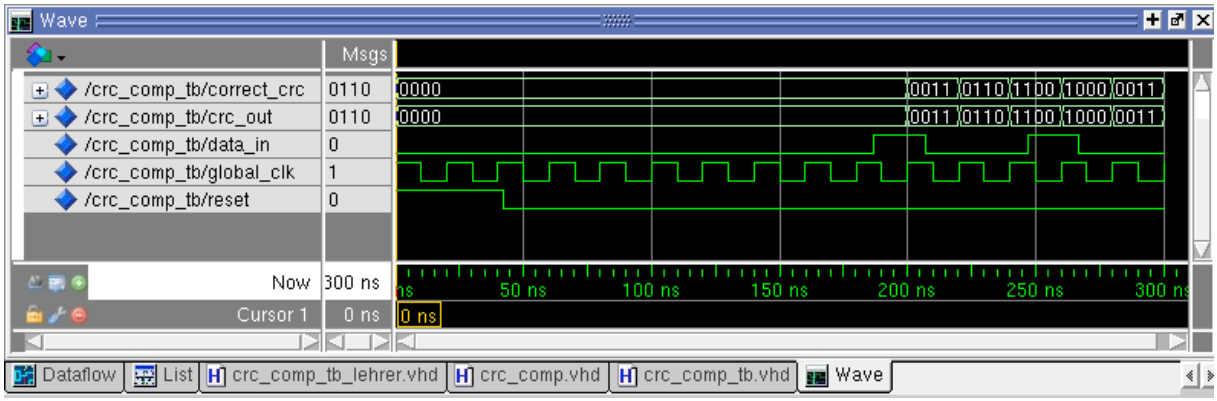


Figure 5: ModelSim Wave view after simulating the CRC-4 testbench for 300 ns.

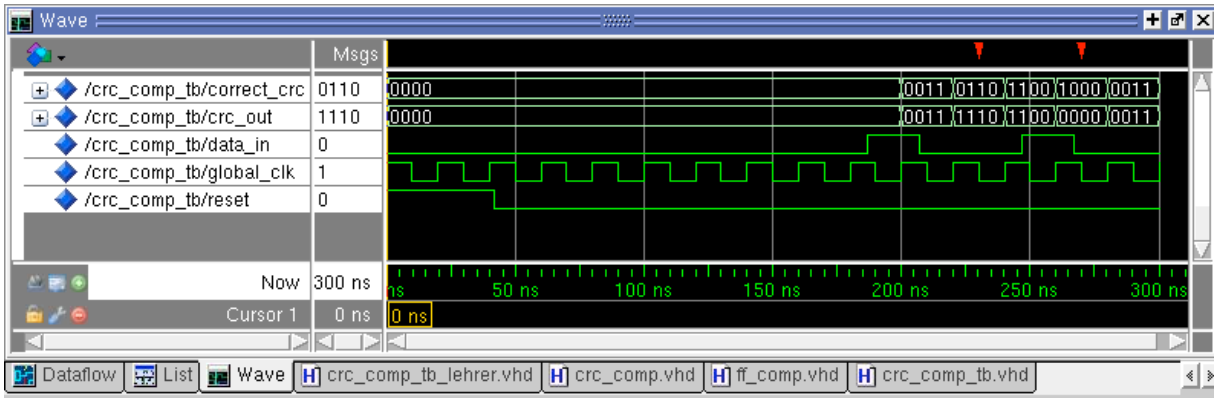


Figure 6: ModelSim simulation indicating an implementation error.

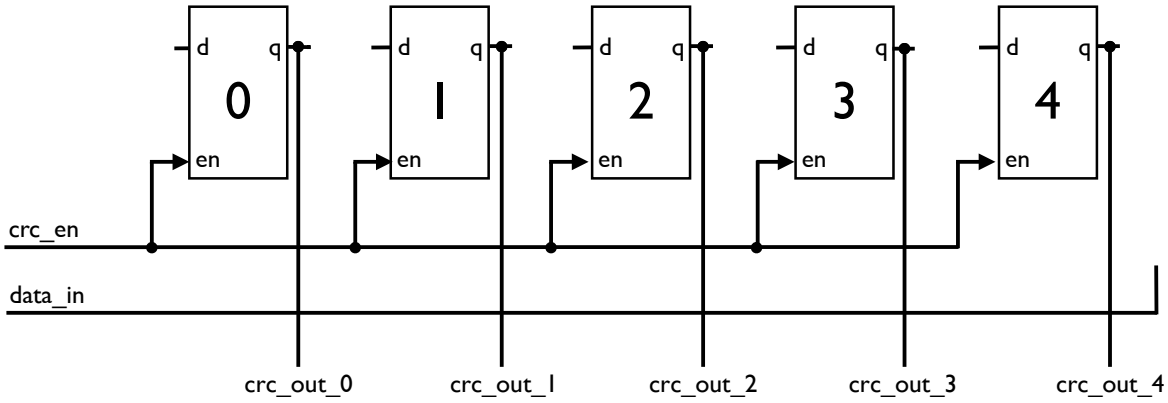
Name: Juncheng Hu
Matriculation number: _____

6 Task sheet

Please solve the following tasks and bring this paper to your practical course.
Not solving these tasks causes your exclusion of this course!

Task 1
Complete the hardware representation of the following CRC generator polynomial:

$$g(x) = x^5 + x^4 + x^3 + x + 1$$



Task 2
Explain the difference between a component and a component instance briefly.