Einführung in die FPGA-Programmierung mit VHDL

Alexander Köster

7. Oktober 2018

Dies ist eine Einführung in die Programmierung eines FPGA mit der Hardwarebeschreibungssprache VHDL. Es handelt sich um eine Zusammenfassung wichtiger Begriffe, Eigenschaften und Beispiele für VHDL aus dem Hardwarepraktikum im Sommer 2018 an der Universität Siegen in Form einer „FAQ“[1]. Zum Test und zur Demonstration wurde ein Digilent Spartan-3 Starter Kit Board (Xilinx-FPGA) verwendet, und dementsprechend die „Xilinx ISE 11“-Programmierumgebung zum Schreiben der FPGA-Programme.

1.) Welche Software und Hardware wird i.d.R. benötigt, um einen FPGA zu programmieren?

Die Programmierung eines FPGA besteht aus mehreren Schritten:

1. **Design** – das Erstellen einer Idee, wie man das gegebene Problem umsetzen kann, ist zunächst unabhängig von spezieller Software oder Hardware.
2. **Editing** – zum Erstellen des Codes in einer Hardwarebeschreibungssprache (*hardware description language*, oder HDL) wird ein Texteditor und/oder eine IDE benötigt, wie z.B. *Xilinx ISE*.
3. **Simulation** – zum Testen des Codes wird eine sogenannte *test bench* benötigt, die ebenfalls programmiert wird und z.B. die Eingangssignale und Ausgabe-Testlogik enthält. Dazu wird ein entsprechendes Programm benötigt, um diese ausführen zu können, oft enthalten in den IDEs.
4. **Synthese** – hier wird aus dem HDL-Code ein Hardwaremapping erstellt. Dabei ist u.a. das Erstellen einer *user constraints file* (UCF) erforderlich, in der z.B. genau angegeben wird, welche Eingangs- und Ausgangssignale auf dem spezifisch genutzten FPGA-Board für den geschriebenen Code verwendet werden sollen.
5. **Post-Synthesis Simulation** – nach der Synthese kann man die synthetisierte Hardwarebeschreibung noch einmal als Simulation testen, um besonders z.B. Timing-bezogene Fehler bei der jeweiligen Hardware zu erkennen, bevor man direkt auf das FPGA geht.
6. **Implementierung** – zum Schluss kann die Hardwarebeschreibung in eine tatsächliche Datei zur Programmierung eines FPGA umgewandelt werden. Man benötigt dann nur noch ein tatsächliches FPGA (z.B. ein Development Board), und kann mit der dem Board zugehörigen Verbindungshardware zum Computer das geschriebene Programm auf das FPGA laden und in einer echten Schaltung testen.

Quellen:

- `https://numato.com/kb/learning-fpga-verilog-beginners-guide-part-1-introduction/`
- `https://www.embedded.com/design/prototyping-and-development/4006429/FPGA-programming-step-by-step`
- `https://en.wikipedia.org/wiki/Field-programmable_gate_array#Design_and_programming`

[1] *frequently asked questions*, häufig gestellte Fragen – *hier:* Frage-Antwort-Zusammenfassung

© study.woalk.de

2.) Fragen zu VHDL

2.a) Was ist der Unteschied zwischen Verhaltens-, Struktur- und Datenflussbeschreibung?

Dies sind die verschiedenen „*code styles*“, in denen man VHDL-Code (genauer, den Code der VHDL *architecture*, also der tatsächlichen Logik des FPGAs) schreiben kann.

- **Verhaltensbeschreibung** oder ***behavioral description model***:
  Als abstraktestes Modell für VHDL-Programmierung wird hier die Logik tatsächlich „programmiert“, entweder durch sequentielle oder parallele Algorithmen in `process`es.

  Dabei werden meist *statements* wie `if`, `case` und `loop` verwendet.

  Diese Art der Beschreibung ist nicht immer auf jede Hardware synthetisierbar, da sie sich am wenigsten um die Hardwareebene kümmert und kaum auf solche optimiert werden kann, ähnlich wie Mikroprozessorcode (z.B. *C*).

  Bei einfachen Designs entspricht ein *behavioral model* auch oft dem Abarbeiten der Wahrheitstabelle dieses Bauteils durch `if` *statements*.

- **Datenflussbeschreibung** oder ***data flow description model***:
  Hier werden Boolsche Funktionen für die parallele Berechnung der Ausgangssignale verwendet (`and`, `or`, `not`). Dies unterscheidet sich erst bei komplexeren Programmen vom *behavioral modeling*.

  Beim *data flow modeling* wird damit beschrieben, wie die Daten sich zwischen den Ein- und Ausgängen bewegen – wie die verschiedenen Signale verbunden sind. Das macht auch deutlich, warum für den Signalzuweisungsoperator `>=` statt `=` oder `:=` verwendet wird, da hier keine Datenzuweisung, sondern eine Beziehungsbeschreibung stattfindet.

- **Strukturbeschreibung** oder ***structural description model***:
  Dies ist die eindeutigste und hardwarenaheste Beschreibung der jeweiligen Logik. Hier wird die Logik aufgeteilt in `component`s, die z.B. die elementaren Gatter darstellen. Damit kann deutlich mehr Einfluss darüber genommen werden, wie die einzelnen Elemente verbunden werden. Damit kann man den Code oft stärker optimieren (weniger Zwischensignale, weniger Verbindungen, aber gleiche Funktionalität wie z.B. *data flow implementations* derselben Logik). Dies vergrößert aber auch den Code und macht ihn weniger flüssig lesbar.

Quellen:

- https://steemit.com/vhdl/@drifter1/logic-design-vhdl-behavioral-dataflow-and-structural-models
- http://www.gmvhdl.com/firstex.htm
- http://www.csun.edu/edaasic/roosta/lecture%205.pdf
- http://surf-vhdl.com/vhdl-syntax-web-course-surf-vhdl/vhdl-behavioral-modeling-style/

2.b) Aus welchen Modulen besteht die Beschreibung eines VHDL-Modells?

1. ***Package*** (optional, *keyword*: `package`) – hier können allgemein genutzte Definitionen gemacht werden, z.B. das Deklarieren eigener Typen mit dem `type` *statement*. Das *package* kann dann mit dem `use` statement in anderen Modulen genutzt werden.

2. ***Entity*** (*keyword*: `entity`) – hier werden üblicherweise die Ein- und Ausgangssignale (`port`s) des Modells definiert. Man kann sich das *entity*-Modul also vorstellen wie ein Blockschaltbild.

3. ***Architecture*** (*keyword*: `architecture`) – die Architektur des Modells ist die eigentliche Beschreibung des Designs, also die Logik. Hier werden die in 8.a) beschriebenen *description models* zum Ausdrücken der Funktion verwendet.

4. ***Configuration*** (optional, *keyword*: `configuration`) – bei komplexeren Designs kann hier das *mapping* zwischen *entities* und *architectures* oder (z.B. im Falle des *structural description models*) zwischen Schaltnetz und *low-level* `component`s definiert werden. Dies ist oft vor allem bei Simulation und *testing* komplexerer Modelle hilfreich.

Quellen:

- `https://www.ics.uci.edu/~jmoorkan/vhdlref/Synario%20VHDL%20Manual.pdf`
- `http://www.srmuniv.ac.in/ramapuram/sites/ramapuram/files/EC308.pdf`
- `https://www.nandland.com/vhdl/examples/example-configuration-vhdl.html`

2.c) Welche beiden Beschreibungsmittel werden bei der Verhaltensbeschreibung unterschieden?

Man unterscheidet zwischen **sequentiellen** (*sequential*) und **parallelen** (*concurrent*) Algorithmen.
Bei *sequential designs* werden, ähnlich wie in reinem C oder Assembler, alle *statements* nacheinander ausgeführt. Bei *concurrent designs* werden mehrere *statements* gleichzeitig ausgewertet, was Ausführungszeit spart.

Quellen:

- `http://www.csun.edu/edaasic/roosta/lecture%205.pdf`

2.d) Welche Datentypen gibt es in VHDL?

VHDL ist *strongly typed*, achtet also zwingend auf den Typ von Signalen und Variablen.

- Skalare Datentypen
  - Numerische Datentypen
    * `integer` – ein üblicherweise 32-Bit Ganzzahlwert (`-2147483648` bis `2147483648`) als Zweierkomplement, je nach Software und Hardware.
    * `real` – ein rationaler Zahlenwert, üblicherweise als *double precision floating point* nach IEEE 754, je nach Software und Hardware. (nicht synthetisierbar)
  - Aufzählungstypen (*enumerated*)
    * `bit` – ein einzelner Binärwert, `0` oder `1`.
    * `boolean` – ein Wahrheitswert, `true` oder `false`.
    * `character` – ein ASCII-Zeichen für Strings. (nicht synthetisierbar)
    * `std_ulogic` bzw. `std_logic` – der 9-wertige Logiktyp nach *IEEE standard 1164*, siehe 8.e).
- Physikalische Typen
    * **Zeit** – Zeitwerte im Bereich Femtosekunden (fs) bis Stunden.

  (nicht synthetisierbar)
- Vektorielle Typen (`std_logic_vector`, Strings, Arrays, ...)

Quellen:

- `https://www.ics.uci.edu/~jmoorkan/vhdlref/Synario%20VHDL%20Manual.pdf`, S.19-24
- `http://www.srmuniv.ac.in/ramapuram/sites/ramapuram/files/EC308.pdf`, S.16ff.
- `https://en.wikibooks.org/wiki/Programmable_Logic/VHDL_Data_Types`
- `https://www.csee.umbc.edu/portal/help/VHDL/types.html`
- `https://web.engr.oregonstate.edu/~sllu/vhdl/lec2e.html`

2.e) Welche Zustände werden bei 9-wertiger Logik unterschieden?

```
type std_ulogic is (
    'U', -- nicht initialisiert
    'X', -- klares "unbekannt"
    '0', -- klare 0
    '1', -- klare 1
    'Z', -- hohe Impedanz/hochohmig
    'W', -- schwaches "unbekannt"
    'L', -- schwache 0
    'H', -- schwache 1
    '-'  -- Don't care
);
```

Bei der Synthese sind nur `'0'`, `'1'`, `'Z'` und `'-'` sinnvoll unterstützt.

Quellen:

- `https://www.ics.uci.edu/~jmoorkan/vhdlref/Synario%20VHDL%20Manual.pdf`, S.22

2.f) Beschreiben Sie die drei Objektklassen in VHDL.

- **Konstante** – kann praktisch überall deklariert werden und speichert einen konstanten Wert, um den Code besser lesbar und leichter aktualisierbar zu machen (analog der Konstanten oder Compilermakros in anderen typischen Programmiersprachen).

  Syntax:

  ```
  constant constant_name : constant_type := value;
  ```

  z.B.:

  ```
  constant storage_size : integer := 1048576;
  ```

- **Variable** – kann in einem sequentiellen Block benutzt werden, um temporär einen Wert zu speichern, der nur in dem jeweiligen Block verfügbar ist (analog einer lokalen Variable in anderen typischen Programmiersprachen).

  Syntax:

  ```
  variable variable_name : variable_type;
  variable_name := value;
  ```

- **Signal** – ähnlich zur Variable wird ein Signal zum Halten eines Wertes verwendet, stellt hier jedoch üblicherweise eine Verbindung zwischen Eingangs-, Ausgangs- oder interner Gatterports dar.

  Im Gegensatz zu Variablen kann man das Zuweisen eines Signals zeitlich verschieben.

  Syntax:

  ```
  signal signal_name : signal_type;
  signal_name <= value after 20 ns;
  -- oder
  signal_name <= value;
  ```

  Selbst wenn man kein solches `after` angibt, wird ein Signal nicht sofort zugewiesen, sondern minimal verzögert. Man nennt diese Verzögerung *delta delay* und nimmt dies als infinitesimale Zeiteinheit größer als 0 an.

  Dies ist wichtig für Szenarien wie das folgende Beispiel, in dem gebündelte Statements jeweils gleichzeitig ausgeführt werden:

```
variable x : integer;
signal sig : integer;
sig <= 20;
-- ... später:
sig <= 42;
x := sig;
-- x hat aber jetzt den Wert 20
```

Im Gegensatz zu Variablen kann mit Signalen auch nicht arithmetisch operiert werden (Inkrement, Dekrement, ...).

Quellen:

- https://www.ics.uci.edu/~jmoorkan/vhdlref/Synario%20VHDL%20Manual.pdf, S.16-18
- http://surf-vhdl.com/vhdl-syntax-web-course-surf-vhdl/vhdl-types-of-data-object/
- http://www.srmuniv.ac.in/ramapuram/sites/ramapuram/files/EC308.pdf, S.20

2.g) Wie erfolgt eine Wertzuweisung bei einem Signal, wie bei einer Variablen?

s.o. 8.f) (:= bei Variablen und <= bei einem Signal)

2.h) Was ist bei VHDL ein `port` bzw. ein `generic`?

Die Deklarationen `port` und `generic` werden im `entity`-Block verwendet, um die Schnittstellen zur Außenwelt zu definieren.
Ein `port` ist eine dynamische Schnittstelle zur Außenwelt und damit äquivalent zu einem Pin auf dem Blockschaltbild. Ein Port entspricht in VHDL einem Signal, wird also wie ein solches gelesen und mit <= zugewiesen. Man deklariert den Port mit einem der folgenden Modi:

- `in` – Eingang, nur Lesezugriff.
- `out` – Ausgang, nur Schreibzugriff.
- `inout` – Ein- und Ausgang, hat Lese- und Schreibzugriff.
- `buffer` – wie `out`, aber mit Lesezugriff innerhalb der `entity` (gepufferte Variable).

Ein `generic` ist ein statischer Wert ähnlich einer Konstanten, und hat auch ähnliche Anwendungszwecke (Vereinfachung, Lesbarkeit).

Beispiel:

```
entity my_processor is
    generic (bus_width : integer := 7);
    port (data_bus : in std_logic_vector (bus_width downto 0);
          q_out : out std_logic_vector (bus_width downto 0));
end my_processor;
```

Hier wird die Datenbreite der Eingänge und Ausgänge durch ein `generic` definiert. Möchte man die Busbreite später ändern, kann man einfach `bus_width` anders definieren. (Die Datenbreite wird hier von einem Vektor definiert, `x downto 0` bezeichnet eine *range* von $x > 0$ bis 0.)

Quellen:

- http://www.srmuniv.ac.in/ramapuram/sites/ramapuram/files/EC308.pdf, S.5-7
- https://www.ics.uci.edu/~jmoorkan/vhdlref/Synario%20VHDL%20Manual.pdf, S.12
- http://insights.sigasi.com/tech/to-downto-ranges-vhdl.html

2.i) Was ist bei VHDl eine Assertion und wann erfolgt eine Ausgabe?

Eine Assertion („Versicherung") ist, wie in anderen Sprachen (z.B. Java) auch, ein Konsistenzcheck, der versichert, dass eine Bedingung zutrifft und ansonsten eine Fehlermeldung wirft.
Syntax:

```
assert boolean_condition report "error message here" severity ERROR;
```

Dabei sind sowohl `report` als auch `severity` optional. `report` gibt den angegebenen String aus, wenn `boolean_condition` nicht zutrifft. `severity` gibt die Fehlerstufe an, mit der die Ausgabe geschieht, mögliche Werte sind `NOTE`, `WARNING`, `ERROR` und `FAILURE`. Ein Level von `FAILURE` bricht üblicherweise eine Simulation sofort ab.
Assertionen bieten sich an für das Testen von Schaltungen im Simulator als *test bench*-Code, so wie auch im Anhang des Skripts vorgeschlagen. Sie werden nicht synthetisiert.

Beispiele:

```
assert a=(b or c);
assert j<i report "wrong order";
assert clk='1' report "clock not up" severity WARNING;
```

Quellen:

- `http://www.srmuniv.ac.in/ramapuram/sites/ramapuram/files/EC308.pdf`, S.27
- `https://www.ics.uci.edu/~jmoorkan/vhdlref/assert.html`

2.j) Was ist bei VHDL ein Prozess?

Ein `process` ist ein Block, der das sequentielle Ausführen von *statements* erlaubt, die ohne einen solchen Block immer parallel ausgeführt werden.
`process`es werden im `architecture`-Modul deklariert. Jeder `process` wird zunächst zu Beginn einer Simulation jeweils parallel ausgeführt.
Ein `process` kann aber auch verwendet werden, um auf Änderungen (*events*) von Signalen zu hören. Dazu kann man eine sogenannte *sensitivity list* bestehend aus bereits deklarierten Signalen zu jedem `process` deklarieren. Ändert sich ein Signal in der *sensitivity list*, so wird der jeweilige `process` (erneut) ausgeführt. Im `process` deklarierte Variablen werden dabei nur beim ersten Aufruf deklariert, nicht jedes Mal erneut.
Hier ein Beispiel:

```
signal x : some_signal_type;
-- ...
count: process(x)
    variable c : integer := -1;
    begin
        c := c + 1;
end process;
```

`x` ist hier in der *sensitivity list*, dieser `process` zählt also in der Variable `c` die Anzahl der *events* von `x`.

Prozesse können auch aufeinander oder auf andere Bedinungen inmitten ihrer Ausführung warten, indem man das `wait`-*statement* verwendet.

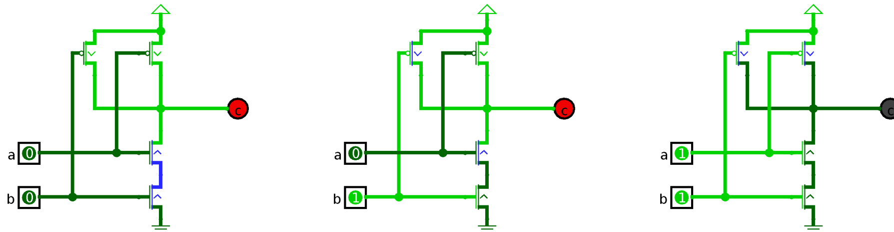
```
wait until clk='1'; -- wartet auf die gegebene Bool'sche Bedingung
wait on sig1, sig2; -- wartet auf ein Event auf einem der beiden Signale
```

Quellen:

- `http://www.srmuniv.ac.in/ramapuram/sites/ramapuram/files/EC308.pdf`, S.19-27
- `https://www.ics.uci.edu/~jmoorkan/vhdlref/Synario%20VHDL%20Manual.pdf`, S.41-44

3.) Wie ist ein NAND2-Gatter in CMOS-Technologie aufgebaut?

Ein NAND2-Gatter („nicht-und“ mit 2 Eingängen) gibt genau dann 1 aus, wenn mindestens einer der beiden Eingänge 0 ist, sonst gibt es 0 aus.
Dies wird durch Parallelschaltung von selbstleitenden p-Kanal-MOSFETs auf der U_{DD}-Seite und Reihenschaltung von selbstsperrenden n-Kanal-MOSFETs auf der U_{SS}-Seite erreicht. Sind beide Eingänge 1, sperren die parallelen MOSFETs beide und die gereihten MOSFETs leiten beide, somit erreicht U_{SS} den Ausgang. Ist eine der Leitungen 0, sperrt einer der gereihten MOSFETs, wodurch dieses Signal nicht mehr den Ausgang erreicht, und einer der parallelen MOSFETs leitet, wodurch U_{DD} den Ausgang erreicht.



a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

$c := a \text{ NAND } b$

Quellen:

- `https://tams.informatik.uni-hamburg.de/lehre/2003ss/vorlesung/T2/t2-gatter.pdf`, S.31
- Schaltbilder aus eigenem NAND-Gatter in Logisim

4.) Erstellen Sie mit *Xilinx ISE* ein NAND2-Gatter.

Datei: `nand2.vhd`

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity nand2 is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           x : out  STD_LOGIC);
end nand2;

architecture Dataflow of nand2 is
begin
    x <= a NAND b after 1 ns;
end Dataflow;
```

5.) Fragen zur Aufgabe 4:

5.a) Was ist der Unterschied zwischen Verhaltenssimulation und *post-timing simulation*?

Die Verhaltenssimulation führt nur den VHDL-Code aus, man kann damit prüfen, ob es generelle Probleme mit der Funktionalität des Codes gibt. Die *timing simulation* (auch *post-synthesis simulation*) wird nach dem Umwandeln in FPGA-Code auf dem tatsächlichen Bitcode ausgeführt. Hier wird versucht, so nah wie möglich an die tatsächlichen Hardwarebeschränkungen beim Simulieren heranzukommen, um Fehler durch die Synthese in FPGA-Baustein-Code, Fehler in den *UCF files* und Timing-Probleme durch zeitabhängige Befehle und asynchrone Pfade zu finden.

Quellen:

- `https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_simulation_timing.htm`

5.b) Womit sind die Pins des *UCF files* verbunden?

Die Eingänge `a` und `b` sind verbunden mit den beiden *slide switches* *SW0* (F12) und *SW1* (G12).
Der Ausgang `x` ist verbunden mit der LED *LD0* (K12).

Quellen:

- *Spartan-3 Starter Kit Board User Guide*, S.19-20

6.) Erstellen Sie mit *Xilinx ISE* einen 1-Bit-Volladdierer als *behavioral* inklusive *UCF file*.

Datei: `myAdd.vhd`

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity myAdd is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           c_in : in  STD_LOGIC;
           s : out  STD_LOGIC;
           c_out : out  STD_LOGIC);
end myAdd;

architecture Behavioral of myAdd is
begin
    process(a,b,c_in)
    begin
        if (a='1' and b='1') then
            if (c_in='1') then
                s <= '1';
            else
                s <= '0';
            end if;
            c_out <= '1';
        elsif (a='1' or b='1') then
            if (c_in='1') then
                s <= '0';
                c_out <= '1';
            else
                s <= '1';
                c_out <= '0';
            end if;
        else
            if (c_in='1') then
                s <= '1';
            else
                s <= '0';
            end if;
            c_out <= '0';
        end if;
    end process;
end Behavioral;
```

Auszug aus `architecture` von Datei: `testbench.vhd`

```vhdl
-- ...
stim_proc: process
begin
    a <= '0'; b <= '0'; c_in <= '0';
    wait for 100 ns;
    a <= '0'; b <= '0'; c_in <= '1';
    wait for 100 ns;
    a <= '0'; b <= '1'; c_in <= '0';
    wait for 100 ns;
    a <= '0'; b <= '1'; c_in <= '1';
    wait for 100 ns;
    a <= '1'; b <= '0'; c_in <= '0';
    wait for 100 ns;
    a <= '1'; b <= '0'; c_in <= '1';
    wait for 100 ns;
    a <= '1'; b <= '1'; c_in <= '0';
    wait for 100 ns;
    a <= '1'; b <= '1'; c_in <= '1';
    wait for 100 ns;
end process;

response_control: process
begin
    report "simulation:" severity note;
    wait for 5 ns;
    assert s='0' and c_out='0' report "wrong answer for 0 + 0 c 0" severity error;
    assert s='0' nand c_out='0' report "right answer: 0 + 0 c 0 = 0 c 0" severity note;
    wait for 100 ns;
    assert s='1' and c_out='0' report "wrong answer for 0 + 0 c 1" severity error;
    assert s='1' nand c_out='0' report "right answer: 0 + 0 c 1 = 1 c 0" severity note;
    wait for 100 ns;
    assert s='1' and c_out='0' report "wrong answer for 0 + 1 c 0" severity error;
    assert s='1' nand c_out='0' report "right answer: 0 + 1 c 0 = 1 c 0" severity note;
    wait for 100 ns;
    assert s='0' and c_out='1' report "wrong answer for 0 + 1 c 1" severity error;
    assert s='0' nand c_out='1' report "right answer: 0 + 1 c 1 = 0 c 1" severity note;
    wait for 100 ns;
    assert s='1' and c_out='0' report "wrong answer for 1 + 0 c 0" severity error;
    assert s='1' nand c_out='0' report "right answer: 1 + 0 c 0 = 1 c 0" severity note;
    wait for 100 ns;
    assert s='0' and c_out='1' report "wrong answer for 1 + 0 c 1" severity error;
    assert s='0' nand c_out='1' report "right answer: 1 + 0 c 1 = 0 c 1" severity note;
    wait for 100 ns;
    assert s='0' and c_out='1' report "wrong answer for 1 + 1 c 0" severity error;
    assert s='0' nand c_out='1' report "right answer: 1 + 1 c 0 = 0 c 1" severity note;
    wait for 100 ns;
    assert s='1' and c_out='1' report "wrong answer for 1 + 1 c 1" severity error;
    assert s='1' nand c_out='1' report "right answer: 1 + 1 c 1 = 1 c 1" severity note;
    wait;
end process;
-- ...
```

Datei: `myAdd.ucf`

```
NET 'a' LOC = F12;      # SW0
NET 'b' LOC = G12;      # SW1
NET 'c_in' LOC = H14;   # SW2
NET 's' LOC = K12;      # LD0
NET 'c_out' LOC = P14;  # LD1
```

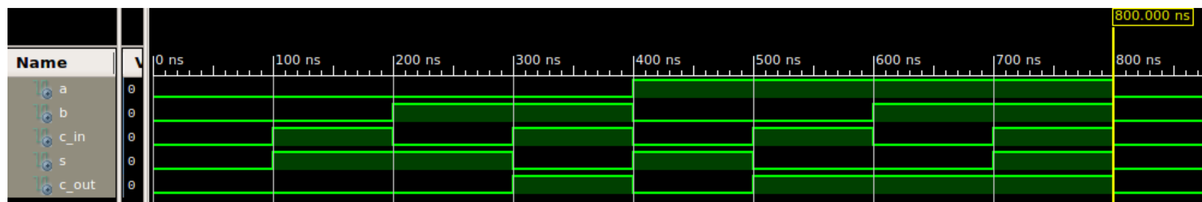


Abbildung 2: Werteverlauf bei *post-route simulation* des Volladdierers.

7.) Fragen zu einer LED-Blinksteuerung

7.a) Mit welchem Takt arbeitet der Oszillator des Testboards?

Der *Epson SG-8002JF series clock oscillator* auf dem *Spartan-3 Starter Kit board* hat eine Taktfrequenz von **50 MHz**.
Außerdem gibt es einen 8-Pin *DIP socket* für den Anschluss eines externen Oszillators.

Der *on-board*-Oszillator ist damit deutlich zu schnell, um eine Blinksteuerung zu realisieren – die LED würde maximal etwas dunkler erscheinen, aber für das menschliche Auge trotzdem dauerhaft leuchten.

Quellen:

- *Spartan-3 Starter Kit Board User Guide*, S.35

7.b) Entwerfen Sie einen Baustein mit VHDL, der als Eingang einen Takt von 50 MHz erhält und als Ausgang einen Takt mit 1 Hz ausgibt.

Man könnte einen Prozess erstellen, der die Anzahl der Taktsignaländerungen zählt, und nach jeweils 50 Millionen Änderungen (bei 50 MHz entspricht das einer halben Sekunde) das Ausgangssignal ändert. Dies teilt die Eingangsfrequenz durch 50 Millionen, macht also aus dem 50 MHz-Signal ein 1 Hz-Signal.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity my1Hz is
    Port ( clk : in  STD_LOGIC;
           clk_out : out  STD_LOGIC);
end my1Hz;

architecture Behavioral of my1Hz is
    signal count : integer := 0;
begin
    process(clk)
    begin
```

```
14          if (rising_edge(clk)) then
15              if (count<25000000) then
16                  -- after 50M events, we changed 50M times from 0 to 1 and back,
17                  -- which at a frequency of 50 MHz equals passed time of half a second
18                  clk_out <= '0';
19                  count <= count + 1;
20              elsif (count<50000000) then
21                  -- after 100M events, we changed 100M times from 0 to 1 and back,
22                  -- now having completed 1 second,
23                  -- so we go back to '0', completing the 1 Hz signal.
24              clk_out <= '1';
25                  -- reset the counter
26                  count <= count + 1;
27              else
28                  count <= 0;
29                  clk_out <= '0';
30              end if;
31          end if;
32      end process;
33  end Behavioral;
```

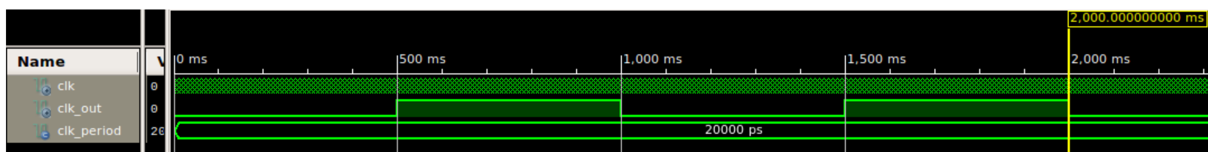


Abbildung 3: Werteverlauf des angegebenen Taktwandlers bei Eingangstakt von 50 MHz.

7.c) Wie können Sie das Signal mit dem Namen `clk` mit dem Oszillator verbinden? (*UCF file*)

```
NET "clk" LOC = T9; # 50 MHz Oszillator liegt per User Guide auf FPGA-Pin T9
```

8.) LED-Blinklichtsteuerung

Um zwischen den verschiedenen gewünschten „Blinkmodi“ der LED wechseln zu können, verwenden wir einen 4x1-MUX. Zusätzlich verwenden wir den 1 Hz-Taktwandler aus Aufgabe 13.b).

Datei: `mux_4to1.vhd`

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity mux_4to1 is
5       Port (a,b,c,d : in  STD_LOGIC;
6             s1,s2 : in  STD_LOGIC;
7             o : out  STD_LOGIC);
8   end mux_4to1;
9
10  architecture Dataflow of mux_4to1 is
11  begin
12      o <= a when (s1='0' and s2='0') else
```

```
            b when (s1='0' and s2='1') else
            c when (s1='1' and s2='0') else
            d; --  (s1='1' and s2='1')
end Dataflow;
```

Datei: `myBlinker.vhd`

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity myBlinker is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           d : out  STD_LOGIC);
end myBlinker;

architecture Structural of myBlinker is
    signal clk1Hz : STD_LOGIC;

    component my1Hz port(clk : in STD_LOGIC;
                         clk_out : out STD_LOGIC);
    end component;

    component mux_4to1 port (a,b,c,d : in  STD_LOGIC;
                          s1,s2 : in  STD_LOGIC;
                          o : out  STD_LOGIC);
    end component;
begin
    U1: my1Hz port map(clk => clk, clk_out => clk1Hz);
    U2: mux_4to1 port map(a => '0', b => clk, c => clk1Hz, d => '1',
                       s1 => b, s2 => a,  o => d);
end Structural;
```

Datei: `myBlinker.ucf`

```
NET 'a' LOC = F12; # SW0
NET 'b' LOC = G12; # SW1
NET 'clk' LOC = T9; # 50MHz oscillator
NET 'd' LOC = K12; # LD0
```

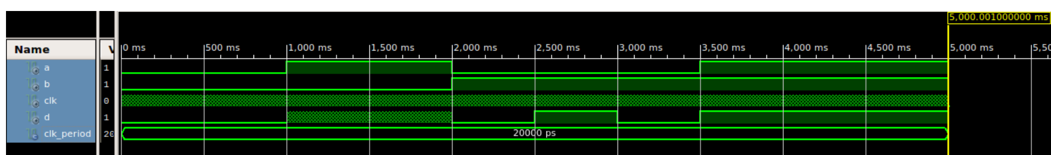


Abbildung 4: Werteverlauf des Blinklicht-Moduls.

Datei: `myBlinker.vhd`

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY testbenchBlinker IS
END testbenchBlinker;

ARCHITECTURE behavior OF testbenchBlinker IS
    COMPONENT myBlinker PORT(a : IN  std_logic;
                             b : IN  std_logic;
                             clk : IN  std_logic;
                             d : OUT  std_logic);
    END COMPONENT;

    --Inputs
    signal a : std_logic := '0';
    signal b : std_logic := '0';
    signal clk : std_logic := '0';
    --Outputs
    signal d : std_logic;

    constant clk_period : time := 20 ns;
BEGIN
    uut: myBlinker PORT MAP (a => a, b => b, clk => clk, d => d);

    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    stim_proc: process
    begin
        wait for 1000 ms; a <= '1'; b <= '0';
        wait for 1000 ms; a <= '0'; b <= '1';
        wait for 1500 ms; a <= '1'; b <= '1';
        wait; -- 1500 ms
    end process;
END;
```

9.) Wie sieht also der generelle Ablauf einer „Programm-Erstellung" für den FPGA aus?

- Man durchläuft die Stufen beschrieben in Aufgabe 1.
- Ein Modul besteht aus den Teilmodulen in Aufgabe 8.b). Ein gesamtes Modell kann aus mehreren Modulen bestehen, die man (z.B. in strukturellen Modellen) als `component`s verwendet.
- Der 1 Hz-Takt wurde erzeugt mit dem Code aus 13.b) und im Modell als `component` verwendet.

10.) Fragen zu *UCF files*:

10.a) Was bedeutet `IOSTANDARD = LVTTL`?

Eine Anweisung der Form

```
NET "HouseOnOff" IOSTANDARD = LVTTL;
```

setzt den gewünschten Standard für Ein-/Ausgänge (*I/O*) auf `LVTTL`.
Der I/O-Standard beschreibt u.a. die am Pin übliche Spannung der logischen 1, gegeben durch die Versorgungsspannung (meist genannt U_{DD} oder V_{DD}).
Der hier verwendete Standard `LVTTL` entspricht $V_{\mathrm{DD}} = 3.3\ \mathrm{V}$, definiert durch den *JEDEC*-Standard *JESD8C.01*.

Quellen:

- https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/cgd.pdf, S.132-134
- https://www.jedec.org/system/files/docs/JESD8C-01.pdf, S.8, Punkt 2.3.3 (Anmeldung erforderlich)
- https://www.xilinx.com/support/documentation/user_guides/ug331.pdf, S.334-345, insbes. S.336, 339

Man muss beachten, dass nur durch das Setzen des Standards nicht die tatsächliche Spannung verändert wird – diese muss in der Schaltung dann entsprechend korrekt an V_{CCAUX} oder V_{CCO} der jeweiligen Pin-Bank angelegt werden.

Quellen:

- https://www.xilinx.com/support/documentation/user_guides/ug331.pdf, S.353ff.
- https://forums.xilinx.com/t5/Welcome-Join/in-ucf-file-Pull-up-IOSTANDARD-Pads-and-wildcard/m-p/277940/highlight/true#M2496

10.b) Was ist die I/O-Standardeinstellung?

Ohne Angabe von `IOSTANDARD` wird dieser als `LVCMOS25` angenommen.

Quellen:

- https://www.xilinx.com/support/documentation/user_guides/ug331.pdf, S.343, vorletzter Absatz.

10.c) Wo liegt der Unterschied zwischen `LVTTL` und `LVCMOS25`?

`LVCMOS25` entspricht $V_{\mathrm{DD}} = 2.5\ \mathrm{V}$, und ist konform nach dem Standard *JESD8C.01* mit dieser V_{DD}.

Darüber hinaus sind auch andere Faktoren durch die `IOSTANDARD`-Einstellung beeinflusst, wie die maximale Stromstärke im Pin. Siehe auch 5.a) und deren Quellen.

Quellen:

- https://www.xilinx.com/support/documentation/user_guides/ug331.pdf, S.335-345, insbes. S.339
- https://www.jedec.org/system/files/docs/JESD8C-01.pdf, S.9, Punkt 2.3.4 (Anmeldung erforderlich)

10.d) Was bedeuten die folgenden beiden Zeilen des *UCF file*?

```
NET "ClockIn" TNM_NET = ClockIn;
TIMESPEC TS_ClockIn = PERIOD "ClockIn" 20 ns HIGH 50%;
```

Mit `TNM_NET` wird eine *timing net group* für `"ClockIn"` erzeugt, die verwendet werden kann, um Zeitspezifische Einstellungen im *UCF file* für die gruppierten Elemente vorzunehmen.
Mit dem `TIMESPEC`-/`PERIOD`-*constraint* wird eine Voraussetzung eines Taktsignals in den Signalen der *timing net group* gegeben, in diesem Fall ein Signal mit Periode von 20 Nanosekunden (= Frequenz von 50 MHz) und 50% *duty cycle* (gleichmäßiger Takt).

Das Angeben dieses *constraints* sorgt dafür, taktgesteuerte Elemente in dem Ast der *timing net group* zu synchronisieren und Verschiebungen (*delays* und *clock skew*) zu analysieren.

Quellen:

- `https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/cgd.pdf`, S.281-284, S.192-193
- `https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug612.pdf`, S.36, S.51-55

11.) Definieren Sie die Zeichen 2-F zur Anzeige auf der Siebesegmentanzeige.

Die Siebensegmentanzeige lässt ein Segment aufleuchten, wenn das Signal an dem entsprechenden Pin 0 ist, und erlischen, wenn es 1 ist. Um die vier einzelnen Anzeigen zu steuern, wird das `AN{0-3}`-Signal verwendet. Ist das `AN`-Signal für eine Ziffer 0, so reagiert diese auf die gegebenen Segmentbelegungen, sonst nicht.

Zeichen	a	b	c	d	e	f	g
0	0	0	0	0	0	0	1
1	1	0	0	1	1	1	1
2	0	0	1	0	0	1	0
3	0	0	0	0	1	1	0
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	0
6	0	1	0	0	0	0	0
7	0	0	0	1	1	1/0	1
8	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0
A	0	0	0	1	0	0	0
b	1	1	0	0	0	0	0
c	0	0	0	1	1	0	1
C	0	1	1	0	0	0	1
d	1	0	0	0	0	1	0
E	0	1	1	0	0	0	0
F	0	1	1	1	0	0	0

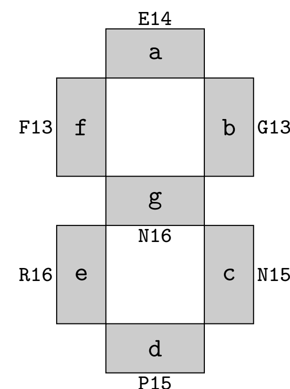


Abbildung 5: Belegung der Siebensegmentanzeige

Quellen:

- *Xilinx Spartan-3 Starter Kit User Guide*, S.15-17
- `https://www.cs.sfu.ca/~ggbaker/reference/std_logic/1164/std_logic_vector.html`

12.) Fragen zur Implementierung einfacher Strukturen:

12.a) Was ist eine Zustandsmaschine in Zwei-Prozess-Darstellung?

Es gibt verschiedene Varianten, mit denen man in VHDL endliche Moore- und Mealy-Automaten (*finite state machines* (*FSM*), oder auch „Zustandsmaschinen“) implementieren kann, sodass sie vom Synthesizer auch als solche erkannt werden.
Die verschiedenen Varianten sind danach benannt, wie viele `process`*es* für die Implementierung verwendet werden. Im **Zwei-Prozess-Modell** eines Moore-/Mealy-Automaten wird ein taktsynchroner Prozess verwendet, um abhängig von den Eingaben den nächsten Zustand zuzuweisen, und ein rein kombinatorischer (nicht taktsynchroner) Prozess, um die Ausgabe abhängig vom Zustand zuzuweisen (bei Mealy-Automaten zusätzlich auch abhängig von der Eingabe).
Für das Zustandssignal verwendet man einen eigenen *enumerated type*.

Quellen:

- `http://www.tkt.cs.tut.fi/kurssit/50200/S16/Kalvot/Lecture%207%20-%20FSM.pdf`, insbes. S.8
- `https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/VHDL/docs-pdf/lab10.pdf`

12.b) Welches Problem kann bei einem Button-Drücken als Eingangssignal der FSM auftreten?

Wenn man nur stur auf den Eingang $= 1$ oder $= 0$ hört, können sehr leicht ein Button-Druck als mehrere aufgefasst werden, da der Button für mehrere Taktdurchläufe gedrückt ist, und so mehrere Zustandsübergänge ausgelöst werden.

12.c) Was kann stattdessen gemacht werden?

Man verwendet einen „Zwischenzustand“, in den übergegangen wird, wenn der Button gedrückt ist, in dem dann verblieben wird, bis der Button das erste Mal losgelassen wird. Dann erst wird in den „eigentlichen“ Zielzustand übergegangen.

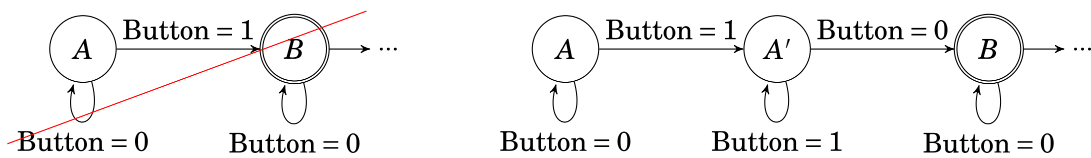


Abbildung 6: FSM mit Zwischenzustand für Verhinderung von mehreren Übergängen

13.) Was definiert das Paket `IEEE.NUMERIC_STD.ALL`?

Durch das Paket, definiert nach *IEEE standard 1076.3*, werden zwei neue Typen für VHDL definiert:

- `signed`, ein vorzeichenbehafteter Ganzzahlwert (*integer*)
- `unsigned`, ein vorzeichenloser Ganzzahlwert (*integer*), bzw. natürliche Zahlen und Null.

Beide Typen sind intern definiert als `array`s von `std_logic`-Signalen.
Wie andere `array`s und `std_logic_vector` wird daher bei der Deklaration eine *range* mit angegeben, die die Bitbreite beschreibt. Eine Bitbreite von 8 Bit wie im folgenden Beispiel kann $2^8 = 256$ verschiedene Werte darstellen, also im Fall von `signed` ein Wert von -127 bis $+127$, und im Fall von `unsigned` ein Wert von 0 bis 255.

```
signal z : signed(7 downto 0); -- 8 mögliche Indizes in dieser Range (0-7)
signal n : unsigned(7 downto 0);
```

Die Besonderheit der Typen ist im Gegensatz zu `std_logic_vector`, dass man viele bereits definierte Operationen hat, wodurch man mit diesen numerischen Werten rechnen kann.
Folgende Operationen sind in `IEEE.NUMERIC_STD` definiert:

- Arithmetische Operationen:
  - → `abs(signed) : signed`
    Absolutwert (Betrag)
  - → `-(signed) : signed`
    Vorzeichenwechsel (unäres Minus)
  - → `+(signed, signed) : signed` bzw. „`signed + signed`"
    `+(unsigned, unsigned) : unsigned` bzw. „`unsigned + unsigned`"
    auch „`signed + unsigned`", „`signed + integer`", „`unsigned + natural`" und umgekehrt.
    Addition (Werte müssen nicht die gleiche Länge haben, Ergebnis hat Länge des Größeren.)
  - → `-(signed, signed) : signed` bzw. „`signed + signed`"
    Subtraktion (andere Kombinationen analog Addition)
    (Werte müssen nicht die gleiche Länge haben, Ergebnis hat Länge des Größeren.)
  - → `*(signed, signed) : signed` bzw. „`signed * signed`"
    Multiplikation (andere Kombinationen analog Addition & Subtraktion)
    (Werte müssen nicht die gleiche Länge haben, Ergebnis hat Länge von beiden zusammen.)
  - → `/(signed, signed) : signed` bzw. „`signed / signed`"
    Division (andere Kombinationen analog Multiplikation, rechte Seite $\neq 0$, sonst `ERROR`)
    (Werte müssen nicht die gleiche Länge haben, Ergebnis hat Länge der linken Seite.)
  - → `rem(signed, signed) : signed` bzw. „`signed rem signed`"
    Divisionsrest mit Vorzeichen von linker Seite (andere Kombis & r.S. $\neq 0$ analog Division)
    (Werte müssen nicht die gleiche Länge haben, Ergebnis hat Länge der linken Seite.)
  - → `mod(signed, signed) : signed` bzw. „`signed mod signed`"
    Modulo (Divisionsrest mit Vorzeichen von rechter Seite, sonst analog Divisionsrest)
    (Werte müssen nicht die gleiche Länge haben, Ergebnis hat Länge der linken Seite.)
- Vergleichsoperatoren:
  - → `>(signed, signed) : boolean` bzw. „`signed > signed`"
    „Größer als" (andere Kombinationen analog arithmetischer Operationen möglich, Werte müssen nicht die gleiche Länge haben)
  - → Analog: „`signed < signed`", „`signed >= signed`" „`signed <= signed`", „`signed = signed`"
  - → `/=(signed, signed) : boolean` bzw. „`signed /= signed`"
    „Ungleichheit" (analog andere Vergleichsoperationen)

- Verschiebefunktionen (*shift and rotate*):
  - → `shift_left(unsigned, natural) : unsigned`, `shift_left(signed, natural) : signed`
    Linksverschiebung um `natural` Stellen (Länge bleibt erhalten, links überlaufende Elemente gehen verloren, rechts wird mit Nullen aufgefüllt)
  - → `shift_right(unsigned, natural) : unsigned`, `shift_right(signed, natural) : signed`
    Rechtsverschiebung um `natural` Stellen (Länge bleibt erhalten, rechts überlaufende Elemente gehen verloren, links wird mit Nullen aufgefüllt)
  - → `rotate_left`, `rotate_right` sind **Rotationen** analog Verschiebungen, aber überlaufende Bits werden auf der anderen Seite wieder eingefügt.

  Info: in neueren VHDL-Versionen können auch die Kurzschreibweisen (in wie obiger Reihenfolge) „`signed sll signed`“, „`signed srl signed`“, „`signed rol signed`“ und „`signed ror signed`“ verwendet werden.

- `resize(unsigned, natural) : unsigned`, `resize(signed, natural) : signed`
  Längenänderung, neue Bits werden bei Vergrößerung links ggf. mit dem Vorzeichenbit gefüllt, bei Verkleinerung wird der linke Teil abgeschnitten und Vorzeichenbit wird ggf. beibehalten

- Umwandlungsoperationen:
  - → `to_integer(unsigned) : natural`, `to_integer(signed) : integer`
  - → `to_unsigned(natural, natural) : unsigned`
    `to_signed(integer, natural) : signed`
    Rechte Seite gibt Größe des Ergebnisses vor.

- Logische Operationen:
  - → `not(signed) : signed`, `not(unsigned) : unsigned`
    Komplement oder **Inversion** (Ergebnis hat gleiche Länge wie Eingabe)
  - → `and(signed, signed) : signed` bzw. „`signed and signed`“
    `and(unsigned, unsigned) : unsigned` bzw. „`unsigned and unsigned`“
    Vektor-UND (Eingaben müssen gleiche Länge haben und geben die gleiche Länge zurück)
  - → „`or`“, „`nand`“, „`nor`“, „`xor`“, („`xnor`“ *in neueren Versionen*) analog Vektor-UND.

- `std_match(std_ulogic, std_ulogic) : boolean`
  `std_match(unsigned, unsigned) : boolean`
  `std_match(signed, signed) : boolean`
  `std_match(std_logic_vector, std_logic_vector) : boolean`
  ***Matching*-Operationen** prüfen Gleichheit, aber jedes „*don't care*“-Bit (`'-'`) gilt als gleich mit einem beliebigen anderen Wert an seiner Stelle.

- `to_01(unsigned, std_logic) : unsigned`, `to_01(signed, std_logic) : signed`
  Translationsoperation, ersetzt die schwachen Signale `'H'` und `'L'` durch die jeweils starken Gegenstücke `'1'` und `'0'`. Bits, die keinem dieser vier entsprechen, werden auf das übergebene `std_logic`-Argument gesetzt (per Standard `'0'`).

Da es sich um `std_logic`-`array`s handelt, können auch einzelne Bits einer solchen Zahl durch Array-Operatoren als/mit `std_logic`-Signalen verwendet werden, z.B.:

```
signal x : std_logic;   signal y : unsigned(3 downto 0) := "0111";
x <= y(2);   y(1) <= '0';
-- Ergebnis: x = '1';   y = "0101";
```

Da dadurch diese Typen so nah miteinander verwandt sind, kann zwischen ihnen ohne eine Umwandlungsoperation ge*cast*et werden:

```
signal x : unsigned(3 downto 0) := "0111";
signal y : std_logic_vector(3 downto 0) := "1001";

x <= unsigned(y);
y <= std_logic_vector(x);
-- Ergebnis: x = "1001";   y = "0111";
```

Um einen Wert des Typs `std_logic_vector` mit einem `signed` oder `unsigned` zu vergleichen, muss man diesen also zuerst mit den Umwandlungsoperationen in einen numerischen Wert umwandeln, oder sich einzelne Bits herausgreifen, je nachdem, welche Art des Vergleichs gewünscht ist.

Quellen:

- `https://www.csee.umbc.edu/portal/help/VHDL/numeric_std.vhdl` (*header file* der `NUMERIC_STD`)
- `https://www.csee.umbc.edu/portal/help/VHDL/packages/numeric_std.vhd` (kompletter Quellcode)
- `https://www.csee.umbc.edu/portal/help/VHDL/operator.html`
- `http://www.synthworks.com/papers/vhdl_math_tricks_mapld_2003.pdf`, insbes. S.3-11

14.) Fragen zu Störungen und anderen häufigen Problemen:

14.a) Wozu dient eine Entprellung für der Verwendung von Buttons ohne Störung?

Wie im Aufgabentext bereits gesagt wurde, dient die Entprellung (*debouncing*) dazu, die Buttons ohne bzw. mit weniger Störung verwenden zu können. Da Buttons analoge Signale sind, können sie *glitches* enthalten, wo das Signal kurzzeitig den Wert wechselt (meist im Mikro- oder Nanosekundenbereich), obwohl der Button noch gedrückt ist.
Für ein Software-*debouncing* filtert man das Signal des Buttons, indem man ein Modul dazwischen schaltet, das das Eingangssignal nur in größeren Abständen abtastet und den letzten abgetasteten Zustand als Ausgangssignal für die Nutzung als Button-Signal in der tatsächlichen Programmlogik ausgibt. Dies sorgt damit für ein stabiles Signal, das außerhalb der Abtastintervallgrenzen auch „*glitch*en“ darf.
Wird z.B. als Abtastintervall 1 Million Taktflanken verwendet, entspricht das bei dem 50-MHz-Takt des Boards einer Abtastrate von 25 Hz.

Quellen:

- `http://www.labbookpages.co.uk/electronics/debounce.html`

14.b) Wie kann verhindert werden, dass *latches* entstehen und Signale verloren gehen?

Bei der Synthese von VHDL für einen FPGA entstehen *latches* u.a., wenn ein Signal nicht in jedem möglichen Zweig eines kombinatorischen Prozesses zugewiesen wird – der *compiler* denkt dann, dass dieses Signal *stateful* ist, d.h. es soll in diesem Fall aus der vorherigen Ausführung beibehalten werden anstatt willkürlich ersetzt. In diesem Fall entsteht ein *latch* oder *flip-flop*, ein Speicherbaustein für ein Signal.

Der beste Weg, ein *latch* zu vermeiden, ist also, bei kombinatorischen Prozessen darauf zu achten, dass man bei jeder möglichen Kombination (jedem möglichen „Zweig“) immer alle Ausgänge zugewiesen hat, bzw. in `if`-Konstrukten immer ein voll-ausgefülltes `else` und in `case`-Konstrukten immer ein voll-ausgefülltes `when others` zu haben, wenn nicht jede Eingangskombination genutzt wird.

Signalverluste können entstehen, weil durch *latches* oft Zeitverzögerungen und „Umwege“ im Schaltkreis entstehen. Viele (Xilinx-)FPGAs haben kein eigenes Bauelement für *latches* und setzen eines im Fall, dass es vom Code „benötigt“ wird, aus anderen Logikelementen zusammen, was ineffizient ist.

Quellen:

- `https://www.xilinx.com/support/answers/3583.html`
- `https://www.nandland.com/articles/what-is-a-latch-fpga.html`
- `https://www.doulos.com/knowhow/fpga/latches/`

14.c) Entwerfen Sie einen Zustandsautomaten für ein elektronisches Kombinationsschloss.

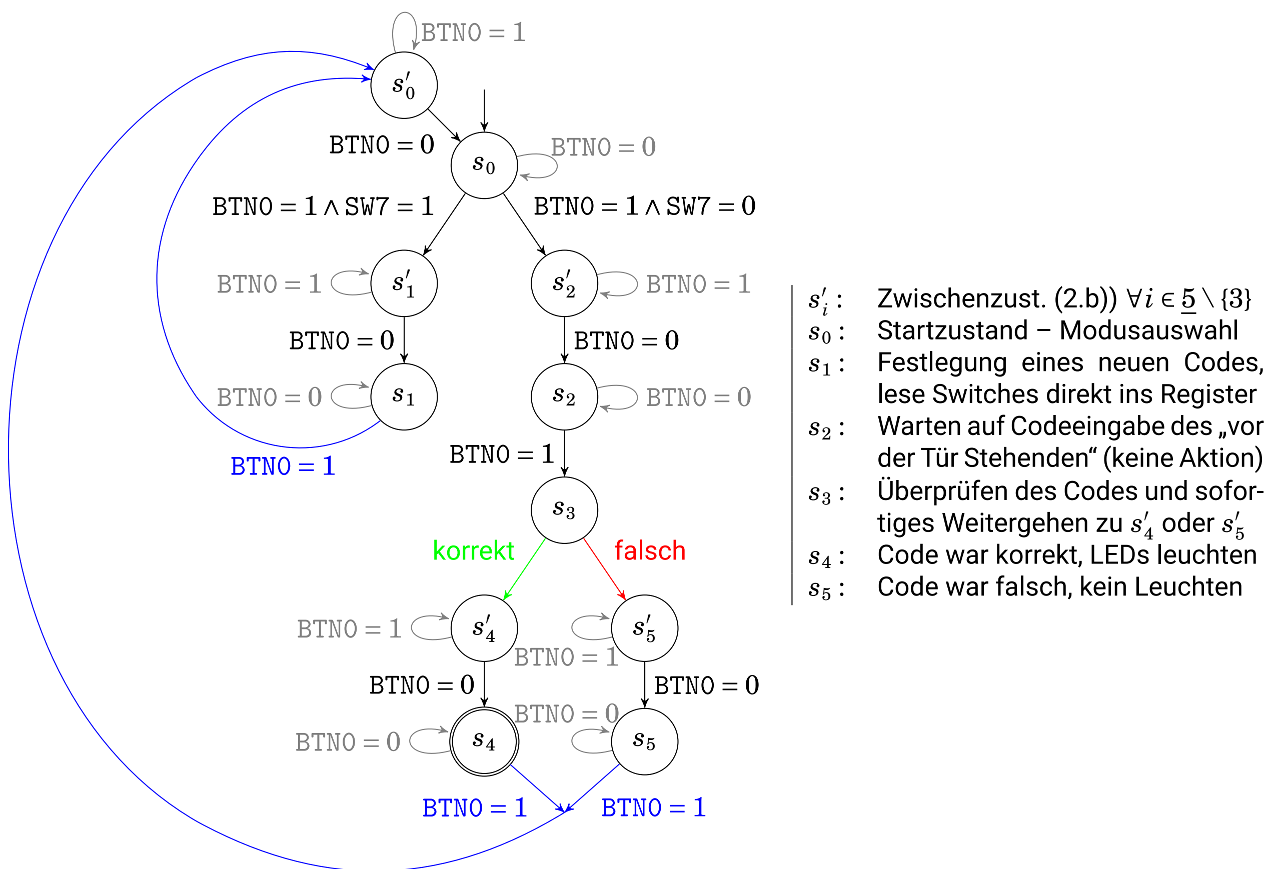


Abbildung 7: FSM-Entwurf des programmierbaren Kombinationsschlosses

Das Signal für `BTN0` muss dabei entprellt werden.
Unter „korrekt“ und „falsch“ wird eine Abfrage verbunden, die prüft, ob der aktuell an den Switches anliegende Code dem gespeicherten Code im Register entspricht.
Der 6-Bit-Code wird also absichtlich *stateful* durch dadurch entstehende Speicherbausteine gespeichert. Würde man dies nicht tun, würde dem Automaten an mehreren Stellen $2^6 = 64$ Zustände hinzugefügt, was nicht mehr übersichtlich wäre.