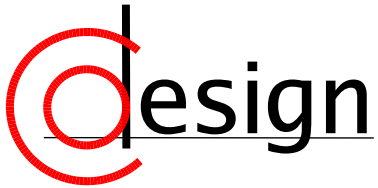
Übungen zur Grundlagen der Technischen Informatik

Übung 12 – VHDL und Komparatoren

Florian Frank

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2018/19



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Was machen wir heute?

Organisatorisches: Vorlesungsevaluation

Was machen wir heute?

Organisatorisches: Vorlesungsevaluation

Aufgabe 1 – VHDL: Funktionen

Was machen wir heute?

Organisatorisches: Vorlesungsevaluation

Aufgabe 1 – VHDL: Funktionen

Aufgabe 2 – ALU in VHDL

Was machen wir heute?

Organisatorisches: Vorlesungsevaluation

Aufgabe 1 – VHDL: Funktionen

Aufgabe 2 – ALU in VHDL

Aufgabe 3 – VHDL: Automaten

Was machen wir heute?

Organisatorisches: Vorlesungsevaluation

Aufgabe 1 – VHDL: Funktionen

Aufgabe 2 – ALU in VHDL

Aufgabe 3 – VHDL: Automaten

Aufgabe 4 – Komparator

Was machen wir heute?

Organisatorisches: Vorlesungsevaluation

Aufgabe 1 – VHDL: Funktionen

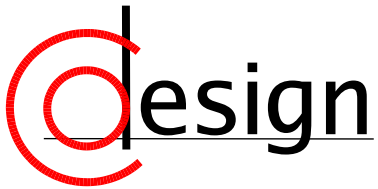
Aufgabe 2 – ALU in VHDL

Aufgabe 3 – VHDL: Automaten

Aufgabe 4 – Komparator

Korrektur und Besprechung der ersten Miniklausur

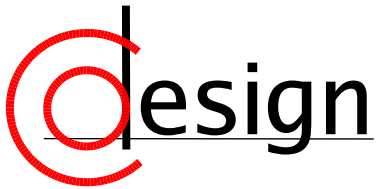
Organisatorisches: Vorlesungsevaluation



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Aufgabe 1 – VHDL: Funktionen



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Aufgabe 1 – VHDL: Funktionen

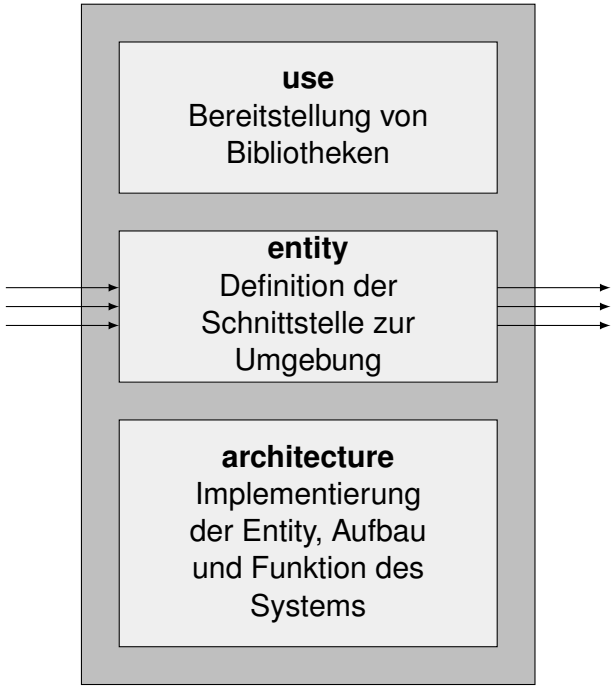
Im VHDL-2008-Standard wurde ein unärer `or`-Operator eingeführt, der die Elemente eines beliebig langen Vektors vom Typ `std_logic_vector` mittels sukzessiver Veroderung auf eine 1 Bit lange Ausgabe vom Typ `std_logic` reduziert.
Entwickeln Sie eine Funktion `or_reduce`, die dieselbe Semantik hat wie der beschriebene Operator, um auch Werkzeuge zu unterstützen, die den Standard noch nicht implementieren. Verwenden Sie dazu eine `for`-Schleife und erläutern Sie, wie sich deren Semantik von Schleifen in Software-Programmiersprachen unterscheidet.

VHDL: VHSIC Hardware Description Language (I)

Mit VHDL „programmieren“ wir unsere Hardware und das auf einem möglichst hohen Abstraktionsniveau.

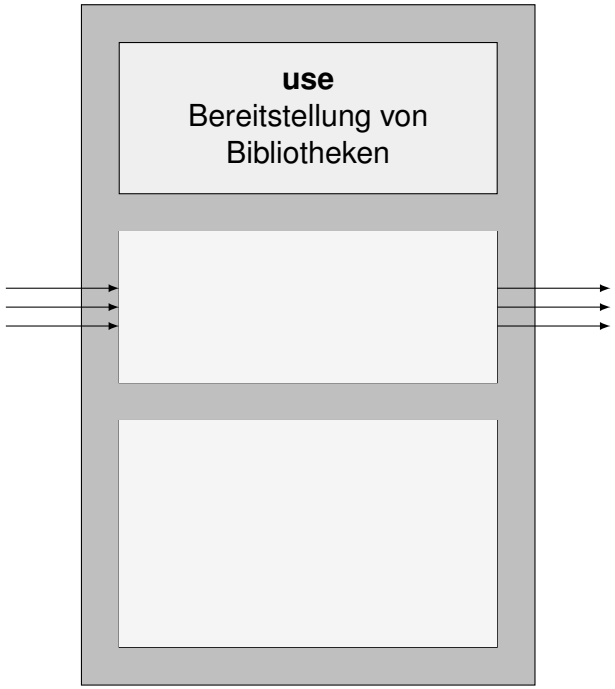
In der Darstellung rechts ist der allgemeine Aufbau einer VHDL-Beschreibung dargestellt.

```
1 use IEEE.std_logic_1164.all; -- Inkludiert das
    Paket std_logic_1164 aus der Bibliothek IEEE
2
```



VHDL: VHSIC Hardware Description Language (I)

Mit VHDL „programmieren“ wir unsere Hardware und das auf einem möglichst hohen Abstraktionsniveau.
In der Darstellung rechts ist der allgemeine Aufbau einer VHDL-Beschreibung dargestellt.



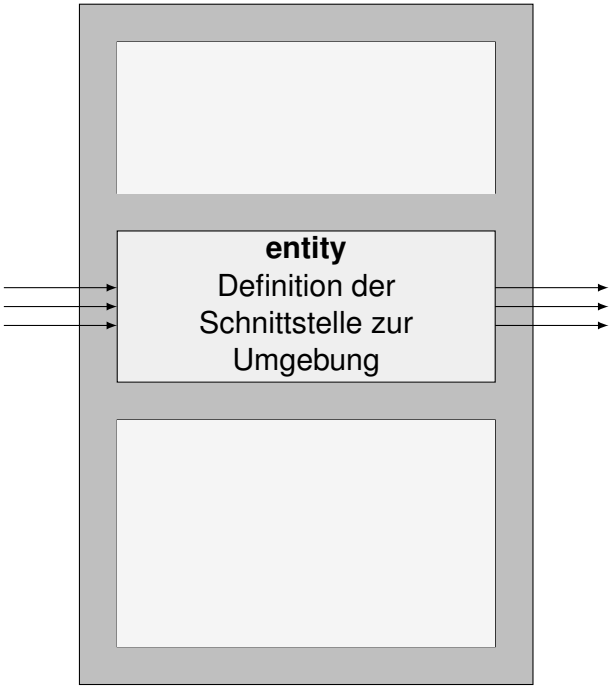
use – Bereitstellung von Bibliotheken

- Genauso wie in Java, gibt es auch in VHDL Pakete, in denen verschiedene Typen oder Funktionen definiert sind (zum Beispiel der Typ `std_logic` aus der `IEEE`-Bibliothek) .
- Diese werden dann über den `use`-Befehl „importiert“. Am Beispiel:

```
1 use IEEE.std_logic_1164.all; -- Inkludiert das
    Paket std_logic_1164 aus der Bibliothek IEEE
2
```

VHDL: VHSIC Hardware Description Language (I)

Mit VHDL „programmieren“ wir unsere Hardware und das auf einem möglichst hohen Abstraktionsniveau.
In der Darstellung rechts ist der allgemeine Aufbau einer VHDL-Beschreibung dargestellt.



entity – Definition einer Schnittstelle

Eine „Entität“ beschreibt die „*black-box*“ einer Komponente. Sie besteht nur aus den **Ein-** sowie *Ausgängen* derselbigen.

```
1 entity entity_name is
2   port (
3     port_list -- hier stehen Ein- und Ausgaenge
4   );
5 end [entity_name];
```

VHDL: VHSIC Hardware Description Language (I)

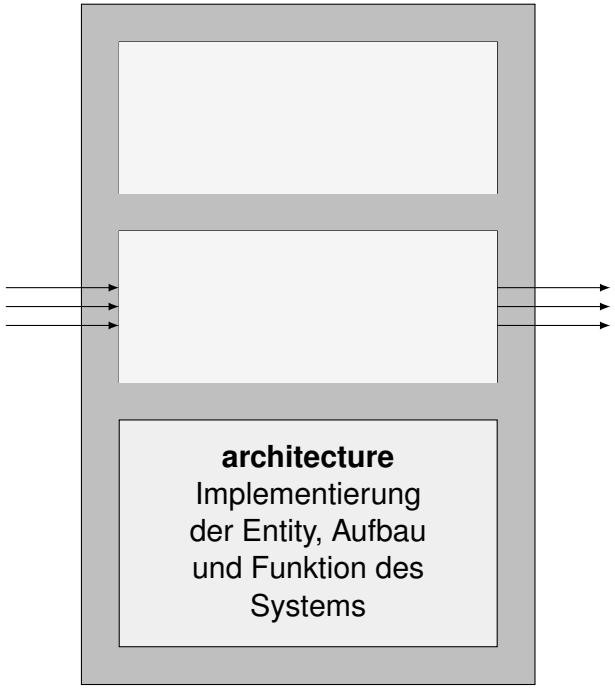
Mit VHDL „programmieren“ wir unsere Hardware und das auf einem möglichst hohen Abstraktionsniveau.

In der Darstellung rechts ist der allgemeine Aufbau einer VHDL-Beschreibung dargestellt.

architecture – Implementierung

Die „Architektur“ einer Entität beschreibt nun den inneren Aufbau, sowie die Funktionalität derselbigen.

```
1 architecture architecture_name of entity_name is
2     -- hier stehen nur intern genutzte Signale
3 begin
4     -- hier steht, was die Entitaet macht
5 end [architecture] architecture_name;
```



VHDL: VHSIC Hardware Description Language (II) – Grundlegendes

Kommentare

```
1 -- Ein Kommentar muss immer mit einem doppelten Minus (--) beginnen und geht bis
    zum Ende der Zeile
2 Hier ist kein Kommentar
3 -- Hier schon
```

Bezeichner

Bezeichner sind in VHDL *case-insensitive* (sprich: Groß- und Kleinschreibung werden nicht unterschieden).
Sie **müssen** mit einem Buchstaben anfangen, anschließend können Buchstaben, Zahlen oder Unterstriche folgen. Zwei Unterstriche dürfen sich dabei **nicht** unmittelbar folgen.

Reservierte Wörter

after	else	library	port	sll
alias	elsif	linkage	postponed	sra
all	end	literal	procedure	srl
and	entity	loop	process	subtype
architecture	exit	map	pure	then
array	file	mod	range	to
assert	for	nand	record	transport
attribute	function	new	register	type
begin	generate	next	reject	unaffected
block	generic	nor	rem	units
body	group	not	report	until
buffer	guarded	null	return	use
bus	if	of	rol	variable
case	impure	on	ror	wait
component	in	open	select	when
configuration	inertial	or	severity	while
constant	inout	others	signal	with
disconnect	is	out	shared	xnor
downto	label	package	sla	xor

VHDL: VHSIC Hardware Description Language (II) – Grundlegendes

Variablen

Genauso wie in Java oder C, enthält eine Variable in VHDL nur eine Information: den aktuellen Wert.

```
1 variable v1 :  std_logic;      -- Variablendeklaration
2 v1          := 1;              -- Wertzuweisung
```

Konstanten

Konstanten verhalten sich ähnlich zu Variablen mit dem Unterschied, dass sie nicht verändert werden können.

```
1 constant c1 : std_logic := X; -- Konstantendeklaration
```

Signale

Signale und Variablen sind ähnlich, es gibt aber einige Unterschiede bei der Verwendung der beiden.

```
1 signal s1  :  std_logic;       -- Signaldeklaration
2 s1 <= X;                       -- Wertzuweisung
```

VHDL: VHSIC Hardware Description Language (II) – Grundlegendes

- Variablen
  Genauso wie in Java oder C, enthält eine Variable in VHDL nur eine Information: den aktuellen Wert
  Variablen verhalten sich insbesondere gleich dem Programmiersprachenkonstrukt, als dass sie **sequentiell** genutzt werden und **überschreibbar** sind.
- Konstanten
  Konstanten verhalten sich ähnlich zu Variablen mit dem Unterschied, dass sie nicht verändert werden können.

```
1 constant c1 : std_logic := X; -- Konstantendeklaration
```

- Signale
  Signale und Variablen sind ähnlich, es gibt aber einige Unterschiede bei der Verwendung der beiden.
  Signale sind vorzustellen als verdrahtete Leitungen, was ihre **nebenläufige** Natur begründet. Ihnen kann in einem Abschnitt nicht mehrfach Werte zugewiesen werden, der zuletzt zugewiesene Wert gilt.

VHDL: VHSIC Hardware Description Language (II) – Beispiel

Welchen Wert enthalten s_2 und v_2 am Ende der Prozedur?

```
1  procedure p_wertezuweisung (
2      variable  v1  : integer;
3      variable  v2  : integer;
4      signal    s1  : out integer := 5;
5      signal    s2  : out integer
6    ) is
7    begin
8      for I in 1 to 10 loop
9          v1 <= I;
10         s1 <= I;
11     end loop;
12     v2 <= v1;
13     s2 <= s1;
14 end p_wertezuweisung;
```

VHDL: VHSIC Hardware Description Language (II) – Beispiel

Welchen Wert enthalten s_2 und v_2 am Ende der Prozedur?

```
1  procedure p_wertezuweisung (
2      variable  v1  : integer;
3      variable  v2  : integer;
4      signal    s1  : out integer := 5;
5      signal    s2  : out integer
6    ) is
7    begin
8      for I in 1 to 10 loop
9          v1 <= I;
10         s1 <= I;
11     end loop;
12     v2 <= v1;
13     s2 <= s1;
14 end p_wertezuweisung;
```

v_2 enthält den Wert 10, s_2 aber 5.

VHDL: VHSIC Hardware Description Language (III) – Funktionen und Prozeduren

Funktionen

Funktionen stellen ein *sequentielles* Unterprogramm mit Rückgabewert dar und können auch rekursiv aufgerufen werden. Sie dürfen ihre Parameter **nicht** verändern.

```
1 function identifier [ ( formal parameter list ) ] return a_type is
2     [ declarations, see allowed list below ]
3 begin
4     sequential statement(s)
5     return some_value; -- muss vom Typ a_type sein
6 end function identifier ;
```

Die Elemente der „*formal parameter list*“ werden durch ein Semikolon (;) von einander getrennt, dem letzten folgt aber **keines**. Ebenfalls darf kein Parameter vom Modus **inout** oder **out** sein.
Erlaubte Deklarationen enthalten unter anderem ...

- ... die Deklaration und der Körper eines Unterprogramms
- ... Konstanten
- ... Variablen
- ... Typen und Subtypen

... aber **nicht** die Deklaration von Signalen.

VHDL: VHSIC Hardware Description Language (III) – Funktionen und Prozeduren

Prozeduren

Prozeduren stellen ein *sequentielles* Unterprogramm ohne Rückgabewert dar. Sie geben Werte zurück, indem sie ihre Parameter oder globale Objekte verändern.

```
1 procedure identifier [ ( formal parameter list ) ] is
2     [ declarations, see allowed list below ]
3 begin
4     sequential statement(s)
5 end procedure identifier ;
```

Die Elemente der „*formal parameter list*“ werden durch ein Semikolon (;) von einander getrennt, dem letzten folgt aber **keines**.
Erlaubte Deklarationen enthalten unter anderem ...

- ... die Deklaration und der Körper eines Unterprogramms
- ... Konstanten
- ... Variablen
- ... Typen und Subtypen

... aber **nicht** die Deklaration von Signalen.

VHDL: VHSIC Hardware Description Language (III) – Funktionen und Prozeduren

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Purpose: This function performs a bitwise xor on the input vector
function f_BITWISE_XOR (
  r_SLV_IN    : in std_logic_vector
) return std_logic is
  variable v_XOR : std_logic := '0';
begin
  for i in 0 to r_SLV_IN'length-1 loop
    v_XOR := v_XOR xor r_SLV_IN(i);
  end loop;
  return v_XOR;
end function f_BITWISE_XOR;
```

VHDL: VHSIC Hardware Description Language (III) – Funktionen und Prozeduren

Prozeduren

Prozeduren stellen ein *sequentielles* Unterprogramm ohne Rückgabewert dar. Sie geben Werte zurück, indem sie ihre Parameter oder globale Objekte verändern.

```
1 procedure identifier [ ( formal parameter list ) ] is
2     [ declarations, see allowed list below ]
3 begin
4     sequential statement(s)
5 end procedure identifier ;
```

Die Elemente der „*formal parameter list*“ werden durch ein Semikolon (;) von einander getrennt, dem letzten folgt aber **keines**.
Erlaubte Deklarationen enthalten unter anderem ...

- ... die Deklaration und der Körper eines Unterprogramms
- ... Konstanten
- ... Variablen
- ... Typen und Subtypen

... aber **nicht** die Deklaration von Signalen.

VHDL: VHSIC Hardware Description Language (IV) – Sequentielle Anweisungen

- Prozedur-/Funktionaufruf

  Ruft eine Prozedur oder Funktion auf.

```
1 [ label: ] procedure-name [ ( actual parameters ) ] ;
```

- if-Abfragen

```
1  [ label: ] if  condition1  then
2       sequence-of-statements
3  elsif  condition2  then      \_ optional
4       sequence-of-statements  /
5  elsif  condition3  then      \_ optional
6       sequence-of-statements  /
7  ...
8  else                        \_ optional
9       sequence-of-statements  /
10 end if [ label ] ;
```

VHDL: VHSIC Hardware Description Language (IV) – Sequentielle Anweisungen

- switch-case

  Führt aufgrund einer gewissen Wahl einen spezifischen Fall aus. Wahlmöglichkeiten müssen Konstanten desselben diskreten Typen wie der Ausdruck sein.

```
1 [ label: ] case  expression  is
2     when choice1 =>
3         sequence-of-statements
4     when choice2 =>             \_ optional
5         sequence-of-statements  /
6     ...
7     when others =>              \_ optional if all choices covered
8         sequence-of-statements  /
9 end case [ label ] ;
```

VHDL: VHSIC Hardware Description Language (IV) – Sequentielle Anweisungen

- Schleifen

  Wiederholte Ausfürhung von Code, kommt in dreierlei verschiedenen Ausführungen.

```
1  [ label: ] loop
2       sequence-of-statements -- use exit statement to get out
3        end loop [ label ] ;
4
5  [ label: ] for variable in range loop
6       sequence-of-statements
7  end loop [ label ] ;
8
9  [ label: ] while  condition  loop
10      sequence-of-statements
11 end loop [ label ] ;
```

VHDL: VHSIC Hardware Description Language (IV) – Sequentielle Anweisungen

- next

  Das `continue` von VHDL, kann auch gleichzeitig noch Bedingung für Fortsetzung enthalten.

```
1 [ label: ] next [ label2 ] [ when condition ] ;
```

- exit

  Das `break` von VHDL, kann auch gleichzeitig noch Bedingung für „Ausbruch“ enthalten.

```
1 [ label: ] exit [ label2 ] [ when condition ] ;
```

- return

  Gibt einen Wert in Funktionen zurück.

```
1 [ label: ] return [ expression ] ;
```

- null

  Wird ein „Statement“ gebraucht, man will nichts tun, so hilft null im nun.

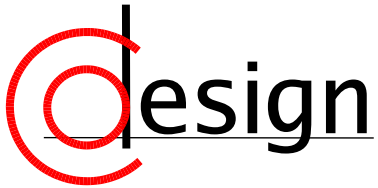
```
1 null ;
```

Aufgabe 1 – VHDL: Funktionen, Lösung der Aufgabe

```
1 function or_reduce(arg: std_logic_vector) return std_logic is
2   variable result : std_logic;
3 begin
4   result := '0';
5   for i in arg'range loop
6     result := result or arg(i);
7   end loop;
8   return result;
9 end or_reduce;
```

Die Ähnlichkeit zu Software-Programmiersprachen trügt:
Wird `or_reduce` in zu synthetisierendem Code (das heißt in tatsächlichen Hardware-Designs und nicht der Simulation) verwendet, wird die Schleife *nicht* sequentiell realisiert. Stattdessen werden für jede Iteration Hardware-Komponenten inferriert (im einfachsten, unoptimierten Fall hier zum Beispiel schlicht `arg'high` Oder-Gatter). Dies ist auch der Grund, wieso in synthetisierbarem VHDL-Code die Schleifengrenzen zur Übersetzungszeit bekannt sein müssen.

Aufgabe 2 – ALU in VHDL



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Aufgabe 2 – ALU in VHDL

Entwerfen Sie ein Rechenwerk `alu` (*Arithmetic Logic Unit*), das in Abhängigkeit eines Steuersignals *op* auf zwei 8 Bit lange Eingabevektoren *a* und *b* die folgenden Operationen durchführt und das Ergebnis auf dem ebenfalls 8 Bit langen Ausgabevektor *result* ausgibt:

- $result \leftarrow a + b$ falls $op = 00$
- $result \leftarrow a - b$ falls $op = 01$
- $result \leftarrow a \wedge b$ falls $op = 10$
- $result \leftarrow a \cdot 2$ (um 1 Linksschieben) falls $op = 11$

Verwenden Sie Signale des Typs `std_logic_vector` für die Schnittstelle und die in der IEEE-Bibliothek `numeric_std` definierten Operationen für die Berechnungen.

Aufgabe 2 – ALU in VHDL, Lösung

Zuerst müssen die benötigten Bibliotheken eingebunden werden:

```
1 library ieee;
2 use ieee.std_logic_1164.all; -- std_logic_vector
3 use ieee.numeric_std.all; -- fuer Arithmetik
```

Aufgabe 2 – ALU in VHDL, Lösung

Zuerst müssen die benötigten Bibliotheken eingebunden werden:

```
1 library ieee;
2 use ieee.std_logic_1164.all; -- std_logic_vector
3 use ieee.numeric_std.all; -- fuer Arithmetik
```

Anschließend wird die zu entwerfende `entity` beschrieben mit ihren Ein- und Ausgängen. Für die ALU benötigen wir die beiden Operanden *a* und *b* sowie die auszuführende Operation *op* als Eingänge und das Ergebnis *result* als Ausgang:

```
4 entity alu is
5     port(
6         a, b : in std_logic_vector(7 downto 0);
7         op : in std_logic_vector(1 downto 0);
8         result : out std_logic_vector(7 downto 0)
9     );
10 end entity alu;
```

Aufgabe 2 – ALU in VHDL, Lösung

Als nächstes folgt die eigentliche Implementierung der `entity`[1], innerhalb einer `architecture`. Zu Beginn der `architecture` müssen zunächst alle internen Signale deklariert werden.

```
11 architecture behavioral of alu is
12   signal signed_result : signed(8 downto 0);
13   signal integer_b : integer;
```

[1] Eine `entity` kann mehrere `architecture`s haben.

Aufgabe 2 – ALU in VHDL, Lösung

Als nächstes folgt die eigentliche Implementierung der `entity`[1], innerhalb einer `architecture`. Zu Beginn der `architecture` müssen zunächst alle internen Signale deklariert werden.

```
11 architecture behavioral of alu is
12   signal signed_result : signed(8 downto 0);
13   signal integer_b : integer;
```

Schließlich beschreiben wir das Verhalten der `architecture`. Der `sll`-Operator (Linksschieben) benötigt einen zweiten Operanden vom Typ `integer`, weshalb wir ein Hilfssignal `integer_b` verwenden. Dem Compiler muss bei der Umwandlung in einen `integer` mitgeteilt werden, ob der Bitvektor *b* vorzeichenbehaftet interpretiert werden soll oder nicht, weshalb zwei Typwandlungen notwendig sind:

```
14 begin
15   integer_b <= to_integer(signed(b));
```

[1] Eine `entity` kann mehrere `architecture`s haben.

Aufgabe 2 – ALU in VHDL, Lösung

Nun folgen die eigentliche durchzuführende Operation, welche wir kombinatorisch implementieren und somit keinen `process` benötigen.

Aufgabe 2 – ALU in VHDL, Lösung

Nun folgen die eigentliche durchzuführende Operation, welche wir kombinatorisch implementieren und somit keinen `process` benötigen. Die arithmetischen Bitoperationen aus `numeric_std` sind für vorzeichenbehaftete und nicht vorzeichenbehaftete Operanden unterschiedlich überladen, weshalb wir, falls nötig, die geforderte Typumwandlung vornehmen:

Aufgabe 2 – ALU in VHDL, Lösung

Nun folgen die eigentliche durchzuführende Operation, welche wir kombinatorisch implementieren und somit keinen `process` benötigen. Die arithmetischen Bitoperationen aus `numeric_std` sind für vorzeichenbehaftete und nicht vorzeichenbehaftete Operanden unterschiedlich überladen, weshalb wir, falls nötig, die geforderte Typumwandlung vornehmen:

```
16   with op select signed_result <=
17     (signed(a) + signed(b)) when "00",
18     (signed(a) - signed(b)) when "01",
19     signed(a and b) when "10",
20     signed(a sll 1) when "11",
21     "00000000" when others;
```

Aufgabe 2 – ALU in VHDL, Lösung

```
16  with op select signed_result <=
17    (signed(a) + signed(b)) when "00",
18    (signed(a) - signed(b)) when "01",
19    signed(a and b) when "10",
20    signed(a sll 1) when "11",
21    "00000000" when others;
```

Das `with` *signal* `select`-Konstrukt inferriert einen Multiplexer, der zwischen den vier Ergebnissen auswählt. Entsprechend werden alle vier möglichen Operationen *gleichzeitig* berechnet.

Aufgabe 2 – ALU in VHDL, Lösung

```
16  with op select signed_result <=
17    (signed(a) + signed(b)) when "00",
18    (signed(a) - signed(b)) when "01",
19    signed(a and b) when "10",
20    signed(a sll 1) when "11",
21    "00000000" when others;
```

Das `with` *signal* `select`-Konstrukt inferriert einen Multiplexer, der zwischen den vier Ergebnissen auswählt. Entsprechend werden alle vier möglichen Operationen *gleichzeitig* berechnet. Zuletzt muss das Ergebnis zurück in einen `std_logic_vector` umgewandelt und dem Ausgangs-`port` zugewiesen werden:

Aufgabe 2 – ALU in VHDL, Lösung

```
16   with op select signed_result <=
17     (signed(a) + signed(b)) when "00",
18     (signed(a) - signed(b)) when "01",
19     signed(a and b) when "10",
20     signed(a sll 1) when "11",
21     "00000000" when others;
```

Das `with` *signal* `select`-Konstrukt inferriert einen Multiplexer, der zwischen den vier Ergebnissen auswählt. Entsprechend werden alle vier möglichen Operationen *gleichzeitig* berechnet. Zuletzt muss das Ergebnis zurück in einen `std_logic_vector` umgewandelt und dem Ausgangs-`port` zugewiesen werden:

```
22   result <= std_logic_vector(signed_result)(7 downto 0);
23 end architecture;
```

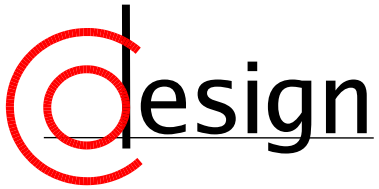
```vhdl
library ieee;
use ieee.std_logic_1164.all; -- std_logic_vector
use ieee.numeric_std.all; -- fuer Arithmetik

entity alu is
    port(
        a, b : in std_logic_vector(7 downto 0);
        op : in std_logic_vector(1 downto 0);
        result : out std_logic_vector(7 downto 0)
    );
end entity alu;

architecture behavioral of alu is
  signal signed_result : signed(8 downto 0);
  signal integer_b : integer;
begin
  integer_b <= to_integer(signed(b));

  with op select signed_result <=
    (signed(a) + signed(b)) when "00",
    (signed(a) - signed(b)) when "01",
    signed(a and b) when "10",
    signed(a sll 1) when "11",
    "00000000" when others;
  result <= std_logic_vector(signed_result)(7 downto 0);
end architecture;
```

Aufgabe 3 – VHDL: Automaten

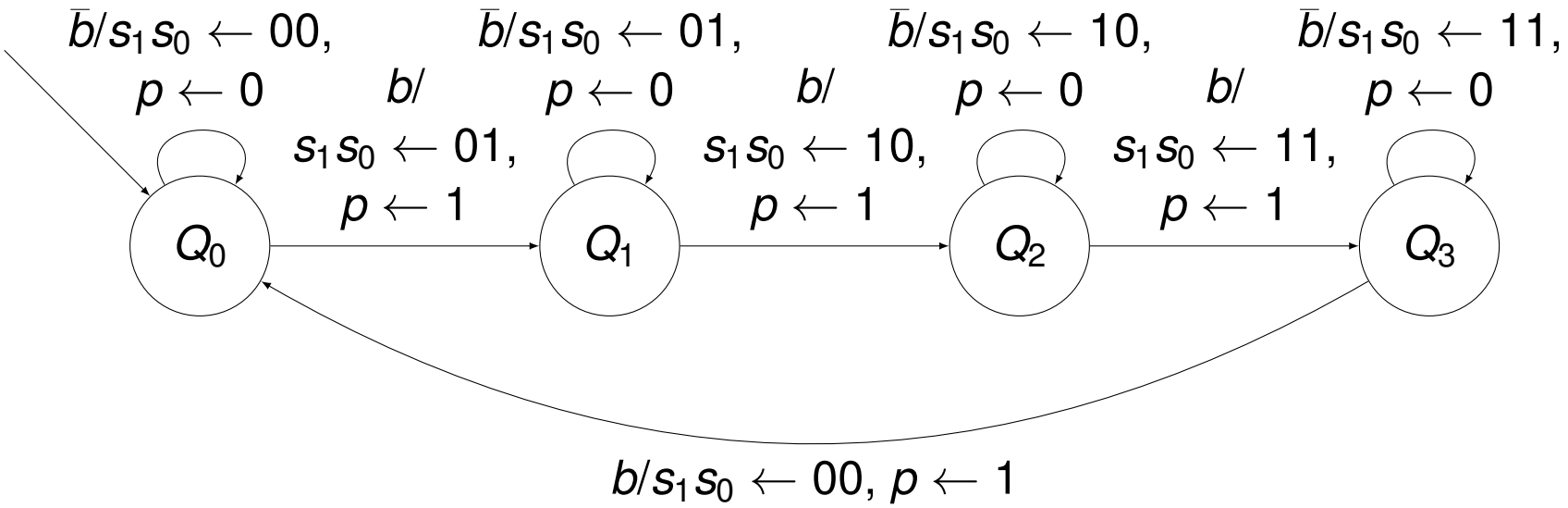


FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

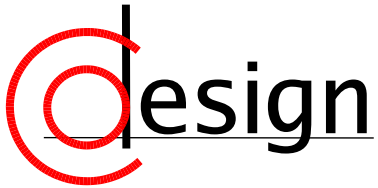
TECHNISCHE FAKULTÄT

Aufgabe 3 – VHDL: Automaten

Implementieren Sie den folgenden Mealy-Automaten, der die Armbanduhr aus Übung 11 beschreibt, in VHDL. Der Automat soll mit einem synchronen Reset-Signal in den Anfangszustand zurückgesetzt werden können.



Aufgabe 4 – Komparator



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Aufgabe 4 – Komparator

a) Entwickeln Sie einen 1-Bit-Komparator, der zwei Bits a und b miteinander vergleicht und auf den Ausgängen $<$, $>$ und = die Gültigkeit der drei Relationen $a < b$, $a > b$ und $a = b$ ausgibt (1 entspreche wahr). Achten Sie darauf, dass stets genau einer der drei Ausgänge aktiv ist.

Aufgabe 4 – Komparator

b) Entwerfen Sie nun eine digitale Schaltung, deren Eingänge die Ausgänge $K_0 = (<_0, >_0, =_0)$ und $K_1 = (<_1, >_1, =_1)$ zweier Komparatoren sind und die diese *lexikographisch* auf die Ausgabe $(<, >, =)$ reduziert. Dabei soll K_0 nieder- und K_1 höherwertig sein.

Aufgabe 4 – Komparator: lexikographisch?

Lexikographisch

Gegeben sei ein quasigeordnetes Alphabet $(\Sigma, \leq)$, d. i. eine Menge von Zeichen $a_i, b_j \in \Sigma$. Eine Zeichenkette $a = (a_1, a_2, \ldots)$ ist lexikographisch kleiner als eine Zeichenkette $b = (b_1, b_2, \ldots)$, das heißt a liegt in der Sortierung vor b, wenn beim komponentenweisen Vergleich Zeichen für Zeichen ...

(entnommen aus: *Wikipediaeintrag zu „Lexikographischer Ordnung“*)

Aufgabe 4 – Komparator: lexikographisch?

Lexikographisch

Gegeben sei ein quasigeordnetes Alphabet $(\Sigma, \leq)$, d. i. eine Menge von Zeichen $a_i, b_j \in \Sigma$. Eine Zeichenkette $a = (a_1, a_2, \ldots)$ ist lexikographisch kleiner als eine Zeichenkette $b = (b_1, b_2, \ldots)$, das heißt a liegt in der Sortierung vor b, wenn beim komponentenweisen Vergleich Zeichen für Zeichen ...

1. das Zeichen a_i von a mit dem niedrigsten Index i, in dem sich die beiden Zeichenketten unterscheiden, (echt) kleiner ist als das entsprechende Zeichen b_i von b,

(entnommen aus: *Wikipediaeintrag zu „Lexikographischer Ordnung“*)

Aufgabe 4 – Komparator: lexikographisch?

Lexikographisch

Gegeben sei ein quasigeordnetes Alphabet $(\Sigma, \leq)$, d. i. eine Menge von Zeichen $a_i, b_j \in \Sigma$. Eine Zeichenkette $a = (a_1, a_2, \ldots)$ ist lexikographisch kleiner als eine Zeichenkette $b = (b_1, b_2, \ldots)$, das heißt a liegt in der Sortierung vor b, wenn beim komponentenweisen Vergleich Zeichen für Zeichen ...

1. $a_i \leq b_i \wedge b_i \not\leq a_i$,

(entnommen aus: *Wikipediaeintrag zu „Lexikographischer Ordnung“*)

Aufgabe 4 – Komparator: lexikographisch?

Lexikographisch

Gegeben sei ein quasigeordnetes Alphabet $(\Sigma, \leq)$, d. i. eine Menge von Zeichen $a_i, b_j \in \Sigma$. Eine Zeichenkette $a = (a_1, a_2, \ldots)$ ist lexikographisch kleiner als eine Zeichenkette $b = (b_1, b_2, \ldots)$, das heißt a liegt in der Sortierung vor b, wenn beim komponentenweisen Vergleich Zeichen für Zeichen ...

1. $a_i \leq b_i \wedge b_i \not\leq a_i$,
2. oder wenn a ein Präfix von b (d. h. $a_i \leq b_i \wedge b_i \leq a_i$ für alle verfügbaren i), aber kürzer ist.

(entnommen aus: *Wikipediaeintrag zu „Lexikographischer Ordnung“*)

Aufgabe 4 – Komparator: lexikographisch?

Lexikographisch *bei Wörtern mit festen Längen*

Gegeben sei ein quasigeordnetes Alphabet $(\Sigma, \leq)$, d. i. eine Menge von Zeichen $a_i, b_j \in \Sigma$. Ein geordnetes Paar $(a_1, a_2) \in \Sigma^2$ ist *lexikographisch kleiner* als ein geordnetes Paar $(b_1, b_2) \in \Sigma^2$, wenn ...

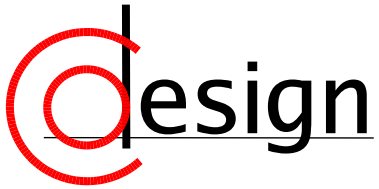
1. $a_1 < b_1$ oder
2. $a_1 = b_1$ und $a_2 < b_2$

(entnommen aus: *Wikipediaeintrag zu „Lexikographischer Ordnung“*)

Aufgabe 4 – Komparator

c) Betrachten Sie schließlich die in a) und b) entworfenen Schaltungen jeweils als Blackbox mit den gegebenen Schaltsymbolen.
Entwerfen Sie ausschließlich mit diesen Komponenten einen Komparator für vorzeichenlose 4-Bit-Binärzahlen. Welche Möglichkeiten gibt es, die Komponenten zusammenzuschalten, und welche Auswirkungen hat dies auf die benötigte Fläche und den kritischen Pfad?

Korrektur und Besprechung der ersten Miniklausur



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT