

HARDWARE- BESCHREIBUNGSSPRACHEN

Hardwareentwurf mit VHDL

21. Oktober 2021
Revision: b941727 (2021-01-16 01:57:51 +0100)

Steffen Reith

Theoretische Informatik
Studienbereich Angewandte Informatik
Hochschule **RheinMain**



Notizen

Notizen

EINE (ZU KURZE) EINFÜHRUNG IN VHDL

LIBRARY DEKLARATIONEN

Mit Hilfe von Bibliotheken kann weitere Funktionalität eingebunden werden:

```
1 library library_name;
2 use library_name.package_name.package_parts;
```

Beispiel:

```
1 library ieee;
2 use ieee.std_logic_1164.all; -- Definiert logische Datentypen
3 use ieee.numeric_std.all; -- Rechnen mit Bitvektoren (std_logic)
```

Die Bibliotheken `std` und `work` werden automatisch eingebunden. Eigene Bibliotheken sind möglich:

```
1 library cpu;
2 use cpu.UARTTypes.all;
```

23

Notizen

ENTITY - SCHNITTSTELLENBESCHREIBUNGEN

Ein `entity` ist die vollständige Spezifikation aller Ein- und Ausgabepins (`ports`) eines Schaltkreises:

```
1 entity entity_name is
2 port(port_name : mode signal_type;
3       port_name : mode signal_type;
4       ....);
5 end entity_name
```

Für `mode` sind `in`, `out` und `inout` zulässig, sowie `buffer`. `buffer` verwendet man, wenn in einem Schaltkreis ein Signal **gelesen und geschrieben** werden muss.

Xilinx empfiehlt `buffer` zu **vermeiden**! Dies kann man erreichen, indem man **Zwischenresultate** (Signale) verwendet.

24

Notizen

ENTITY - EIN BEISPIEL (BESCHREIBUNG EINES SPEICHERS)

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  library cpu;
5  use cpu.CPUTypes.all;
6
7  entity MemoryBank is
8
9      port (clk      : in  std_logic;
10
11           enable   : in  std_logic;
12           writeMem : in  std_logic;
13
14           adr      : in  cpuAdrReal_t;
15
16           dataIn   : in  cpuWord_t;
17           dataOut  : out cpuWord_t);
18
19  end MemoryBank;

```

25

Notizen

ARCHITECTURE - FUNKTIONALITÄT EINES SCHALTKREISES

Mit Hilfe einer Architekturbeschreibung legt man die Funktionalität eines Schaltkreises fest:

```

1  architecture architecture_name of entity_name is
2
3      (Deklarationen von Konstanten, Typen und Signalen)
4
5  begin
6
7      (Code)
8
9  end architecture_name;

```

Hinweis: Es kann zu einem **entity** **mehrere Architekturen** (mit verschiedenen Namen) geben. Dadurch kann man z.B. die Funktionalität eines Schaltkreises speziell für eine Simulation beschreiben, verschiedene Implementationen gegeneinander testen oder für die Synthese optimieren.

26

Notizen

BEISPIEL: SYNCHRONES FLIP-FLOP

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity FF is
5      port(d      : in  std_logic; -- Eingabebit
6           clk     : in  std_logic; -- Clock
7           reset   : in  std_logic; -- Reseteingang
8           q       : out std_logic); -- Ausgabebit
9  end FF;
10
11 architecture behavioral of FF is
12 begin
13     process (clk, reset) -- Sensitivitätsliste
14     begin
15         if (reset = '1') then
16             q <= '0'; -- Zurücksetzen
17         elsif (rising_edge(clk)) then -- Teste auf steigende Flanke
18             q <= d; -- Daten übernehmen
19         end if;
20     end process;
21 end behavioral;

```

27

Notizen

GENERISCHE SCHALTKREISE

Ein **entity** kann durch eine **generic**-Anweisung parametrisiert werden, wodurch das Design leichter wiederverwendet wird:

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Adder is
5
6      generic (width : integer := 8);
7
8      port (a      : in  std_logic_vector(width - 1 downto 0);
9           b      : in  std_logic_vector(width - 1 downto 0);
10          cin : in  std_logic;
11
12          s      : out std_logic_vector(width - 1 downto 0);
13          c      : out std_logic);
14
15 end Adder;

```

28

Notizen

SKALARE DATENTYPEN

Das Bibliothek `std` stellt die Datentypen `bit`, `boolean`, `integer`, `character` und `real` bereit. Der Datentyp `real` kann **nicht synthetisiert** werden (\Rightarrow nur für Simulationen).

Das Package `std_logic_1164` liefert die Datentypen `std_logic` und `std_ulogic`, wobei

```

1 type std_logic is ('U', -- Uninitialized
2                   'X', -- Forcing Unknown
3                   '0', -- Forcing zero
4                   '1', -- Forcing one
5                   'Z', -- High Impedance (tri-state)
6                   'W', -- Weak Unknown
7                   'L', -- Weak zero
8                   'H', -- Weak one
9                   '-'); -- Don't care

```

Im Package `numeric_std` finden sich die Datentypen `signed` und `unsigned`, sowie Rechen- und Konvertierungsfunktionen.

29

Notizen

PHYSIKALISCHE DATENTYPEN

VHDL kennt physikalische Datentypen, d.h. ein Wert hat eine Einheit:

```

1 type length is range 0 to 1E9
2   units
3     um;           -- Primäre Einheit: Micron
4     mm = 1000 um; -- 1000 Micron sind ein Millimeter
5     m  = 1000 mm; -- 1000 Millimeter sind ein Meter
6     inch = 25400 um; -- Imperiale Einheiten
7   end units length;

```

Somit sind `0.1 inch` und `2.54 mm` gleich.

Achtung: Addition und Subtraktion funktioniert auf physikalischen Datentypen, aber bei der Multiplikation und Division kann es Probleme geben (m vs. m^2).

30

Notizen

PHYSIKALISCHE DATENTYPEN (II)

Es gibt auch einen vordefinierten Datentyp für die Zeit, der bei der Simulation eine wichtige Rolle spielt:

```

1  -- Clock period definition
2  constant clk_period : time := 10 ns;
3
4  -- Clock process definitions (simulation starts with raising clock)
5  clk_process : process
6  begin
7
8      clk <= '1';
9      wait for clk_period/2;
10
11     clk <= '0';
12     wait for clk_period/2;
13
14 end process;
```

31

Notizen

FILES

VHDL kennt auch Files. Diese sind allerdings **nicht synthetisierbar** und dienen z.B. dem automatisierten Testen, wobei die Testvektoren aus einer Datei ausgelesen werden.

```

1
2  -- Welcher Datentyp ist im File gespeichert?
3  type typeName is file of typeInFile;
4
5  -- Identifier fuer Zugriff
6  file identifier: typeName [[open openMode] is expression];
```

Beispiel:

```

1  type bFile is file of bit;
2  file file01 : bFile is "data.bit";
```

Im Package **textio** der Bibliothek **std** finden sich Routinen zum Lesen und Schreiben, sowie allgemeine Hilfsfunktionen für Files.

32

Notizen

USER DEFINIERTE SKALARE DATENTYPEN

Von **integer** können Datentypen abgeleitet werden:

```
1 type typeName is range specification;
```

Die Intervallgrenzen müssen zwischen $\text{integer}'\text{low}$ ($= -2^{31} + 1$) und $\text{integer}'\text{high}$ ($= 2^{31} - 1$) liegen:

```
1 type natural is range 0 to integer'high;
2 type negative is range integer'low to -1;
3 type positive is range 1 to integer'high;
4 type temperature is range -273 to 100;
```

Zählt man alle gültigen Werte eines Datentyps auf, so bekommt man **Aufzählungstypen**:

```
1 type bit is ('0', '1');
2 type boolean is (false, true);
3 type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
4 type stateFSM is (idle, exec, transmit, receive);
```

33

Notizen

ARRAYS

Vergleichbar mit Programmiersprachen sind **Arrays** eine Sammlung von gleichen Datentypen:

```
1 type typeName is array (rangeSpecs) of element_t;
```

In VHDL gibt es wichtige vordefinierte Arraytypen:

```
1 type bit_vector is array (natural range <>) of bit;
2 type boolean_vector is array (natural range <>) of boolean;
3 type integer_vector is array (natural range <>) of integer;
4 type string is array (positive range <>) of character;
```

und Arrays mit Logik-Typen (Package `std_logic_1164`):

```
1 type std_ulogic_vector is array (natural range <>) of std_ulogic;
2 type std_logic_vector is array (natural range <>) of std_logic;
```

34

Notizen

ARRAYS - BEISPIELE

Arrays mit **integer**-Typen:

```
1 type t1_t is array (positive range <>) of integer;
2 constant c1 : t1_t(1 to 4) := (42, -7, 11, 0);
3
4 type t2_t is array (0 to 3) of natural;
5 constant c2 : t2_t := (2, 0, 9, 4);
```

Arrays mit Aufzählungen

```
1 type t3_t is array (natural range <>) of std_logic;
2 constant c3 : t3_t(4 downto 1) := "z100";
3
4 type t4_t is array (7 downto 0) of bit;
5 constant c4 : t4_t := "10011001";
```

Die Indizes eines Arrays brauchen keine **integer**-Typen zu sein:

```
1 type row_t is range 1 to 3;
2 type col_t is ('x', 'y', 'z');
3
4 type matrix is array (row_t, col_t) of std_logic;
```

35

Notizen

SLICES

Mit Hilfe von Slices kann man **Teile** eines Arrays kopieren / bearbeiten:

```
1 signal word      : std_logic_vector(15 downto 0);
2 signal lowWord   : std_logic_vector( 7 downto 0);
3 signal highWord  : std_logic_vector( 7 downto 0);
4
5 -- Teile aus dem Array rausschneiden
6 highWord <= word(15 downto 8);
7 lowWord  <= word(7  downto 0);
```

Möchte man nur auf einen Teil eines Arrays zugreifen, so kann man auch **Aliases** definieren:

```
1 alias highWord is word(15 downto 8);
2 alias lowWord  is word(7  downto 0);
```

Achtung: **alias** wird von **vielen Synthesetools nicht unterstützt** und kann so die Portierbarkeit beeinträchtigen.

36

Notizen

RECORDS

Möchte man verschiedene Datentypen zu einem neuen Typ gruppieren, so verwendet man **Records**, die mit `struct` in C vergleichbar sind:

```
1 type ireg_t is record
2   irq : std_logic;
3   pend : std_logic_vector(0 to 7);
4   mask : std_logic_vector(0 to 7);
5 end record;
6
7 signal irq_r : ireg_t;
```

Der Zugriff auf die Komponenten eines `records` funktioniert dann ganz normal mit

```
1 irq_r.mask <= "00110011"
```

37

Notizen

SUBTYPEN

Aufgrund der Codequalität, Testbarkeit oder Klarheit macht es Sinn **Typen einzuschränken**. Mit `integer`-Typen funktioniert das wie folgt:

```
1 subtype natural is integer range 0 to integer'high;
2 subtype positive is integer range 1 to integer'high;
```

Dieser Mechanismus funktioniert auch mit Aufzählungstypen

```
1 type std_logic is ('X', '0', '1', 'Z', 'W', 'L', 'H', '-');
2 subtype simpleLogic is std_logic range 'X' to 'Z';
3
4 type color_t is (red, green, blue, pink, yellow);
5 subtype additiveColor is color_t range red to blue;
```

oder mit Arrays deren Grenzen noch nicht festgelegt wurden

```
1 -- Width of datapath / registers
2 constant cpuDataWidth : integer := 32;
3
4 subtype cpuWord_t is std_logic_vector(cpuDataWidth - 1 downto 0);
```

38

Notizen

SIGNALE UND VARIABLEN

Ein **Signal** dient dazu, Werte in oder aus einem Schaltkreis zu führen, d.h. ein Signal repräsentiert eine **Verbindung / Leitung** innerhalb von Schaltkreisen:

```
1 signal signal_name : data_type [:= expression];
```

Ganz ähnlich können auch Konstanten definiert werden:

```
1 constant signal_name : data_type := expression;
```

Neben Signalen bietet VHDL auch **Variablen**. Diese entsprechen (in der Regel) keinen Verbindungen zwischen Schaltkreisen. Sie dienen der Speicherung von Werten **innerhalb** von **Prozessen**, **Funktionen** und **Unterprogrammen**:

```
1 variable variable_name : data_type [:= expression];
```

Variablen repräsentieren nur **lokale** Informationen!

39

Notizen

SIGNALE UND VARIABLEN (II)

Achtung: Variablen dürfen **nicht** direkt nach einer **architecture**-Vereinbarung verwendet werden.

Der Zuweisungsoperator für Variablen und Signale unterscheidet sich:

Signal: a <= "0010";

Variable: a := "0010";

Die Zuweisung an ein **Signal** wird **nicht sofort** aktiv, sondern erst nach einer gewissen Verzögerungszeit (z.B. erst nach verlassen eines Prozesses). Deshalb ist auch

```
1 clk <= '0' after 5ns;
```

möglich. Dies ist nützlich, um Signallaufzeiten für eine Simulation zu modellieren. Bei der **Synthese** wird **after** **ignoriert**.

Die Zuweisung an eine **Variable** wird immer **somit aktiv**.

40

Notizen

OPERATOREN

VHDL kennt die üblichen Operatoren:

- Logik: **not**, **and**, **nand**, **or**, **nor**, **xor** und **xnor**
- Arithmetik: **+**, **-**, *****, **/**, ****** (Exponentiation), **abs** (absoluter Wert), **mod** und **rem** (ganzzahlige Division)
- Vergleiche: **=** (Equal), **/=** (Not equal), **<** (Less than), **>** (Greater than), **<=** (Less than or equal) und **>=** (Greater than or equal).

Das Package **numeric_std** implementiert die obigen Operatoren auch für **std_logic_vector** (evtl. nicht alle synthesefähig).

Wichtig und oft hilfreich (z.B. Schieberegister) ist der **Konkatenationsoperator (&)**.

Sei nun:

```
1 constant one: bit := '1';
2 constant tri: std_logic := 'Z';
```

41

Notizen

OPERATOREN (II)

Beispiel:

```
1 signal y: bit_vector(1 to 4);
2 signal z: std_logic_vector(7 downto 0);
3
4 y0 <= (one & "010"); -- ergibt "1010"
5 y1 <= one & "000"; -- ergibt "1000"
6 z0 <= (tri & tri & "111" & tri); -- ergibt ZZ111Z
7 z1 <= ('0' & "101" & tri); -- ergibt 0101Z
```

Weiterhin bietet **std_logic_vector** Schiebeoperationen:

- Shift left logic (**sll**) - Bits auf der rechten Seite mit '0' füllen
- Shift right logic (**srl**) - Bits auf der linken Seite mit '1' füllen
- Shift left arithmetic (**sla**) - Bits rechts mit LSB füllen
- Shift right arithmetic (**sra**) - Bits links mit MSB füllen
- Rotate left (**rol**) - Ringshift links
- Rotate right (**ror**) - Ringshift rechts

42

Notizen

BEDINGTE ANWEISUNGEN

Die **when-Anweisung** ist vergleichbar mit einer nebenläufigen Variante der if-Anweisung. **when** darf deshalb nicht innerhalb einer **process**-Anweisung stehen:

```
1 q <= '0' when (reset = '1') else
2   d when (clk = '1');
```

Das **when**-Statement verlangt nicht, dass alle Eingabekombinationen abgedeckt sind (aber: gute Praxis für die Synthese!). Bei der ähnlichen **select**-Anweisung ist das notwendig:

```
1 signal s1, s0 : std_logic;
2 ...
3 s <= s1 & s0; -- Concat
4 with (s) select
5   output <= "000" when "00" | "01",
6             "111" when "10",
7             "111" when "11",
8             "ZZZ" when others; -- mehrwertige Logik
```

43

Notizen

ANWENDUNG: 4-TO-1 MULTIPLEXER

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity Multiplexer4to1 is
5
6   -- Breite des Multiplexers
7   generic (w: integer := 8);
8
9   port (d0, d1, d2, d3 : in std_logic_vector(w - 1 downto 0);
10        sel           : in bit_vector(1 downto 0);
11        output        : out std_logic_vector(w - 1 downto 0));
12
13 end entity;
14
15 architecture whenArch of Multiplexer4to1 is
16 begin
17   output <= d0 when sel = "00" else
18             d1 when sel = "01" else
19             d2 when sel = "10" else
20             d3;
21 end architecture;
```

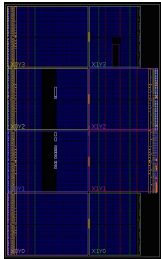
44

Notizen

SEQUENTIELLER CODE

Um **sequentiellen** Code in VHDL (oder Schaltkreisen) erzeugen zu können, braucht man ein **Speicherkonzept**, da sonst keine Zwischenergebnisse „aufgehoben“ werden können.

Das in der Vorlesung verwendete Artix-7 FPGA hat in jedem Slice gleich mehrere Flip-Flops zu diesem Zweck:



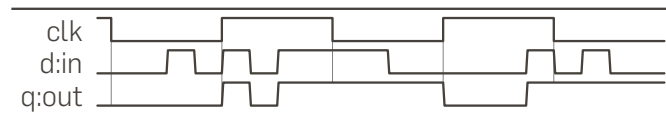
Block-RAMs stellen größere Mengen von Speicher (z.B. 36kB pro Block) zur Verfügung.

45

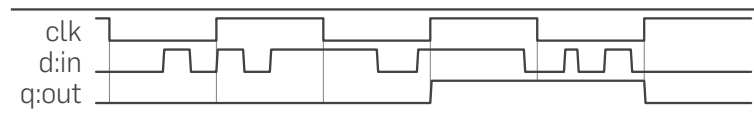
Notizen

SPEICHERELEMENTE

Ein **Latch** (engl. Schnappverschluss / Riegel) ist ein 1-Bit Speicher, dessen Ausgang den Wert des Eingang direkt wiedergibt, solange das Clock/Enable-Signal '1' ist. Dieses Verhalten heißt **level-sensitiv**:



Ein Flip-Flop dagegen ist **edge-sensitiv**, d.h. die Daten werden nur an (steigenden) **Flanken** übernommen.



46

Notizen

DIE PROCESS-ANWEISUNG

Eine `process`-Anweisung wird **sequentiell** abgearbeitet und darf die **üblichen Kontrollstrukturen** enthalten (`if`, `case` oder `loop`).

Zusätzlich ist die `wait`-Anweisung erlaubt, mit der man auf ein **Ereignis** (\triangleq **Veränderung**) eines (Clock)Signals **warten** kann.

```
1 [label:] process [(sensitivity_list)] is
2   [declarative_part]
3 begin
4   sequential_statements_part
5 end process [label];
```

Im **deklarativen** Teil einer `process`-Anweisung dürfen z.B. **Variablen vereinbart** werden.

In der **Sensitivitätsliste** werden alle Signale aufgeführt, die bei einer **Änderung** den **Ablauf des Prozesses** bewirken.

47

Notizen

BEISPIEL: FLIP-FLOP MIT ASYNCHRONEM RESET

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity ff is
5   port ( clk : in std_logic;
6         reset : in std_logic;
7         d : in std_logic;
8         q : out std_logic);
9 end entity;
10
11 architecture Behavioral of FF is
12 begin
13   ff_pro : process (clk, reset)
14   begin
15     if (reset = '1') then -- Asynchroner reset^^I
16       q <= '0'; -- Ausgang resettet^^I
17     elsif ((clk'event) and (clk = '1')) then -- steigende Flanke
18       q <= d; -- Eingang übernehmen
19     end if;
20   end process;
21 end architecture;
```

48

Notizen

EINIGE BEMERKUNGEN ZU WAIT

Die Anweisung

```
1 elsif ((clk'event) and (clk = '1')) then -- Warte steigende Flanke
```

kann man auch abkürzen:

```
1 elsif (rising_edge(clk)) then -- Warte steigende Flanke
```

Für Simulationszweck ist auch die Variante

```
1 clk_process : process
2 begin
3   clk <= '1';
4   wait for clk_period / 2;
5
6   clk <= '0';
7   wait for clk_period / 2;
8 end process;
```

erlaubt. Dies ist aber **nicht** synthetisierbar.

49

Notizen

CODEPARTITIONIERUNG

Mit Hilfe von Package, Component, Function und Procedure kann der Code **strukturiert** werden:

```
1 package packageName is
2   Deklarationen;
3 end [package] [packageName];
4
5 package body packageName is
6   [Unterprogramme;]
7   [zurückgestellte Konstanten]
8 end [package body] [packageName];
```

Im ersten Teil eines Package sind **nur** Deklarationen zulässig, wogegen im zweiten (optionalen) Teil die Implementationen von Function und Procedure untergebracht sind.

50

Notizen

BEISPIEL (AUSSCHNITT): PACKAGE

```

1 package CPUTypes is
2   -- Holds an op-code of any instruction
3   subtype instrOp_t is std_logic_vector(opCodeWidth - 1 downto 0);
4
5   -- Codes for RType operations
6   constant opRType : instrOp_t := "000000";
7
8   -- Access the opcode
9   function GetOp (signal word : in cpuWord_t) return instrOp_t;
10
11 end CPUTypes;
12
13 package body CPUTypes is
14
15   function GetOp (signal word : in cpuWord_t) return instrOp_t is
16   begin
17     -- Simply slice out the operands field
18     return instrOp_t(word((dW - 1) downto (dW - opCodeWidth)));
19   end GetOp;
20
21 end CPUTypes;

```

51

Notizen

WIEDERVERWENDUNG VON CODE: COMPONENT

Für größere Schaltkreise muss es möglich sein, Schaltkreise (mehrmals) wieder verwenden zu können. Dies kann man mit der **component**-Anweisung erreichen:

```

1 component componentName is
2   [generic (
3     constName1 : const_t := value1;
4     constName2 : const_t := value2;
5     ...);]
6   port (
7     portPame : mode signal_t;
8     portPame : mode signal_t;
9     ...);
10 end component [componentName];

```

Mit **port** werden die Ein- und Ausgänge des zu benutzenden Schaltkreises beschrieben (vergleichbar mit **entity**).

Die **component**-Anweisung wird **innerhalb** von **architecture** verwendet.

52

Notizen

WIEDERVERWENDUNG VON CODE: COMPONENT (II)

Hat man eine Komponente in einer Architektur bekannt gemacht, dann kann man diese auch (mehrfach) **benutzen** / **instanziiieren**:

```
1 label: [component] componentName
2       [generic map (genericList)]
3       port map (portList);
```

Mit Hilfe von **generic map** kann man die generischen **Parameter** der **speziellen Instanz festlegen**.

Mit **port map** kann man die Signale der konkreten Instanz mit Signalen der Architektur **verdrahten**.

53

Notizen

BEISPIEL: EIN XOR MIT DREI EINGÄNGEN

Angenommen man hat ein XOR mit zwei Eingängen und es soll draus eines mit drei Eingängen konstruiert werden:

```
1 entity XOR3 is
2   port(i1,i2,i3 : in std_logic;
3         o       : out std_logic);
4 end XOR3;
5
6 architecture Structural of XOR3 is
7   signal tmp : std_logic;
8
9   component XOR2 is
10    port(x1,x2 : in std_logic;
11          y    : out std_logic);
12  end component;
13
14 begin
15
16  -- Zwei XOR2 Gatter (Instanzen) verwenden
17  XOR2_1 : XOR2 port map(x1=>i1, x2=>i2, y=>tmp);
18  XOR2_2 : XOR2 port map(x1=>tmp, x2=>i3, y=>o);
19 end Structural;
```

54

Notizen
