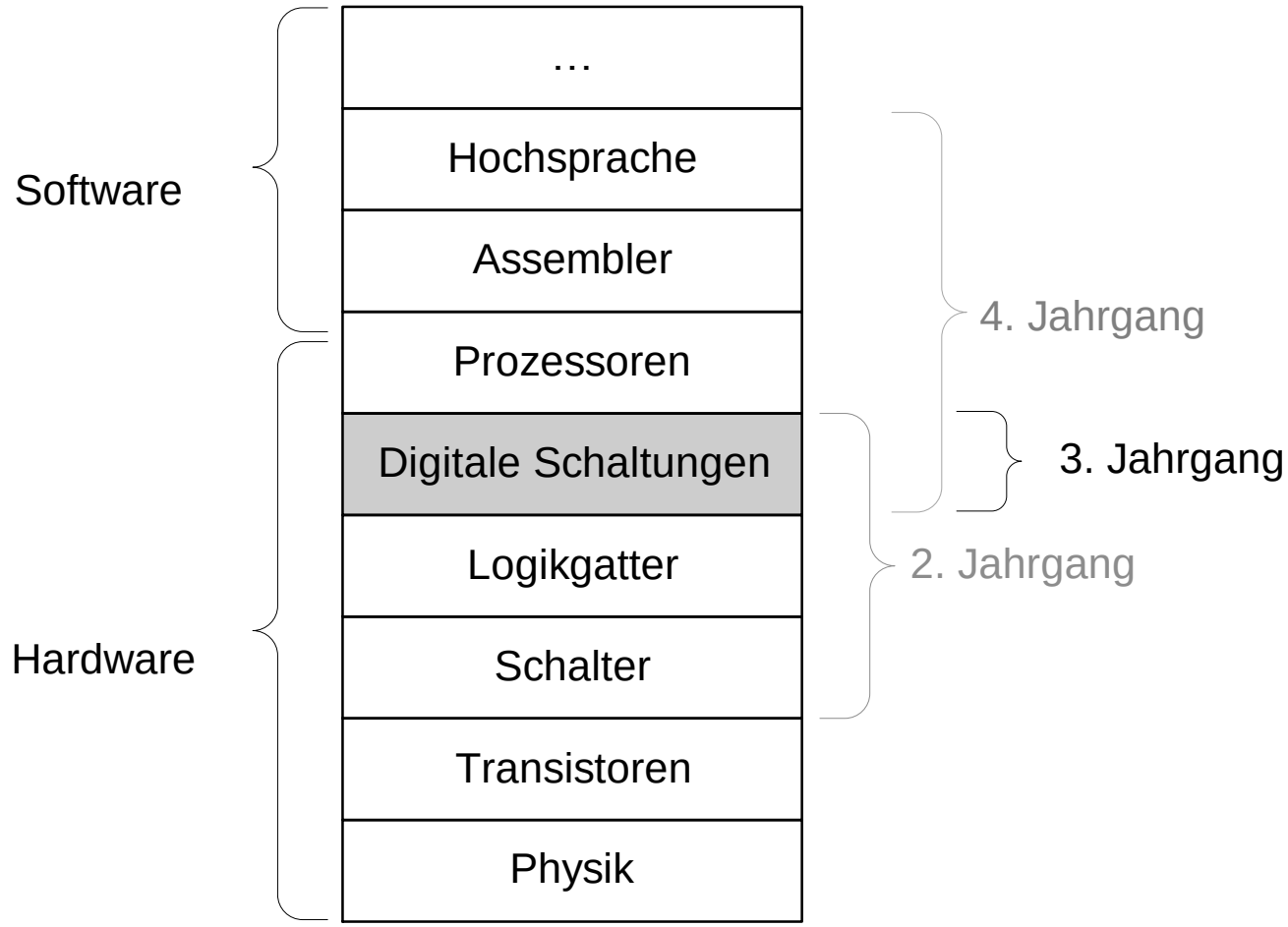
DIGITALE SYSTEME UND COMPUTERSYSTEME

3. Jahrgang (5. und 6. Semester)

Bereich des Unterrichtsgegenstandes im 3. Jahrgang:



1 Zustandsautomaten

Dieses Kapitel ist aus dem Skriptum des 2. Jahrgangs (Kapitel 8.6 – Zustandsautomaten) zu entnehmen und nach dieser Seite einzuordnen.

2 Programmierbare Logikbausteine (PLDs) - Überblick

Zu diesem Inhalt wird ein ergänzender Foliensatz bereitgestellt. Diesen bitte nach dieser Seite einordnen.

3 VHDL – Einführung

VHDL Acronym

VHDL – **V**HSIC **H**ardware **D**escription **L**anguage

VHSIC – **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit

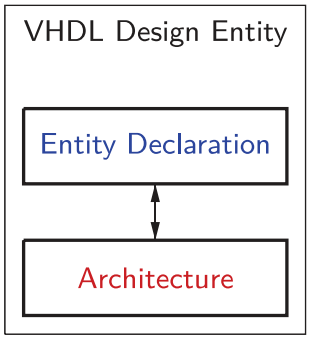
Wofür verwendet?

- Um die Struktur und das Verhalten von digitalen Hardware-Designs zu beschreiben
- Um die Funktionalität eines Designs zu testen

VHDL ist ein internationaler Standard (1076), der von IEEE reguliert wird. Die Definition von VHDL ist nicht-proprietär.

3.1 Modellierung mit VHDL

VHDL erlaubt eine übersichtliche Trennung zwischen externer Ansicht und interner Funktionalität eines digitalen Systems:



Entity

Die Entity-Deklaration beschreibt alle Schnittstellen einer digitalen Funktionseinheit nach außen.

Im Vergleich zu einem Leiterplattenentwurf kann man sich die Entity-Deklaration wie einen bestückten Gehäusesockel vorstellen, der bereits auf dem Board verlötet ist und durch die Art und Bezeichnung der Anschlüsse eindeutig definiert ist.

Architecture

Die Architecture beschreibt die genaue Funktionalität der digitalen Funktionseinheit.

Wenn man es wieder mit dem Leiterplattenentwurf vergleicht, könnte man sich die Architecture als Chip/IC vorstellen, mit dem der obige Gehäusesockel bestückt wird.

Beim Schreiben von VHDL-Code beachten:

VHDL-Code ist im Gegensatz zur Programmiersprache C nicht case-sensitiv, d.h. er ist unabhängig von Groß- und Kleinschreibung. In der Praxis ist es sinnvoll, die selbst definierten Bezeichner durch CamelCase[1] von den vordefinierten VHDL-Syntaxelementen zu unterscheiden, die in der Praxis (und im DIC-Unterricht) durch Kleinschreibung gekennzeichnet sind.

VHDL-Bezeichner müssen aus alphanumerischen Zeichen (mit Ausnahme von Umlauten) oder dem Unterstrich „_“, der wiederum nicht am Anfang oder Ende eines Bezeichners stehen darf, bestehen. Das erste Zeichen alphabetisch sein. Reservierte VHDL-Schlüsselwörter wie z.B. and, or, not, … dürfen natürlich nicht als Bezeichner verwendet werden.

Gültige Bezeichner: z.B.: `ABC, aBc, AB123, AB_cd`

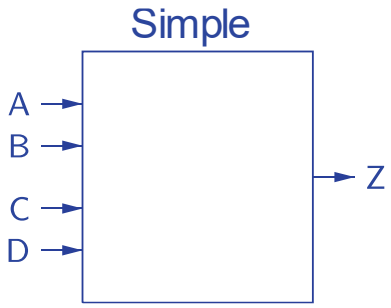
Ungültige Bezeichner: z.B.: `Minus-Zeichen, Zähler, 5Bit_Zaehler, architecture, and`

Dateinamen:

Üblich ist, dass nur ein Entity in eine VHDL-Quelldatei geschrieben wird, wobei der Dateiname den Entity-Namen enhält:

z.B. Name der Entity: `SimpleAdder` → Dateiname: `SimpleAdder.vhd`

3.1.1 Entity-Quelltext



- In der Entity-Deklaration wird zunächst der Name der Entwurfseinheit vergeben, in diesem Fall „Simple“.
- Die Kommunikation mit anderen Entwurfseinheiten erfolgt über `port`-Signale. In der `port`-Deklaration erfolgt die Definition der Ein- und Ausgänge inkl. des entsprechenden Datentyps.

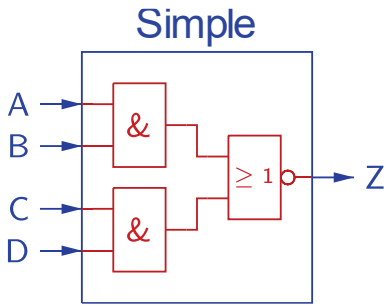
[1] CamelCase: https://en.wikipedia.org/wiki/Camel_case

3.1.2 Architecture-Quelltext

```
-- entity declaration

entity Simple is
  port(A,B,C,D : in  std_logic;
       Z       : out std_logic);
end Simple;

-- architecture body
```



- Der Architecture muss unbedingt ein anderer Name als der Entity gegeben werden
- Ein Bezug mit der Entity muss hergestellt werden (Schlüsselwort `of`)

3.1.3 Gesamte Datei „Simple.vhd“

Der in diesem Beispiel verwendete Datentyp „`std_logic`“ wird aus dem Paket „`std_logic_1164`“ bezogen, daher muss der Quelltext in diesem Beispiel um zwei Zeilen oben erweitert werden:

```
library IEEE;
use IEEE.std_logic_1164.all;

-- entity declaration
entity Simple is
  port(A,B,C,D : in  std_logic;
       Z       : out std_logic);
end Simple;

-- architecture body
architecture Simple_arch of Simple is
begin
  Z <= not((A and B) or (C and D));
end Simple_arch;
```

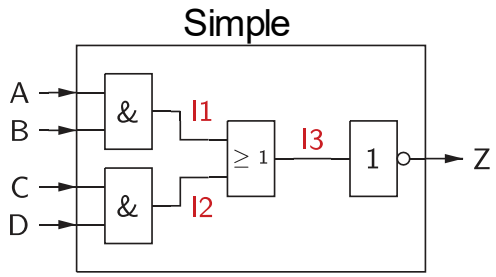
3.1.4 Interne Signale

Soll eine Schaltung als VHDL-Datenflussmodell beschrieben werden, werden interne Signale benötigt:

```
library IEEE;
use IEEE.std_logic_1164.all;

-- entity declaration
entity Simple is
  port(A,B,C,D : in  std_logic;
       Z       : out std_logic);
end Simple;

-- architecture body
architecture Simple_arch of Simple is
  signal I1,I2,I3 : std_logic;
begin
  I1 <= A and B;
  I2 <= C and D;
  I3 <= I1 or I2;
  Z <= not I3;
end Simple_arch;
```



- Interne Signale müssen vor dem `begin` der Architecture deklariert werden
- Jede Signalzuweisung repräsentiert eine nebenläufige Operation

3.1.5 Nebenläufige Signalzuweisungen

Verglichen mit prozeduralen Programmiersprachen wie z.B. C/C++ entsprechen Signalzuweisungen den Wertzuweisungen der Programmiersprache: Auf der linken Seite des Zuweisungsoperators „<=“ befindet sich das Signal, dem ein neuer Signalwert zugewiesen wird.

Alle nebenläufigen Signalzuweisungen einer Architektur werden parallel ausgeführt.

Man unterscheidet drei Arten von nebenläufigen Anweisungen:

Unbedingte Signalzuweisung

Dem Signal wird eine Signalkonstante oder ein Boole'scher Ausdruck zugewiesen.

Allgemein:

```
signal_name <= expression;
```

Beispiel:

```
Y <= A and B;
```

Bedingte Signalzuweisung

Dem Ausgangssignal werden unterschiedliche Signalwerte zugewiesen, je nachdem ob einer der Bedingungsausdrücke, die sich hinter den Schlüsselwörtern `when` bzw. `else when` befinden, erfüllt ist.

Allgemein:

```
signal_name <=
expression when condition else
expression when condition else
expression;
```

Beispiel:

```
Y <= A when S=”00” else
     B when S=”11” else
     C;
```

Selektive Signalzuweisung

Es existiert ein durch das Schlüsselwort `select` bezeichnetes Selektionssignal welches bestimmte, durch das Schlüsselwort `when` bezeichnete Werte, annehmen kann.

Allgemein:

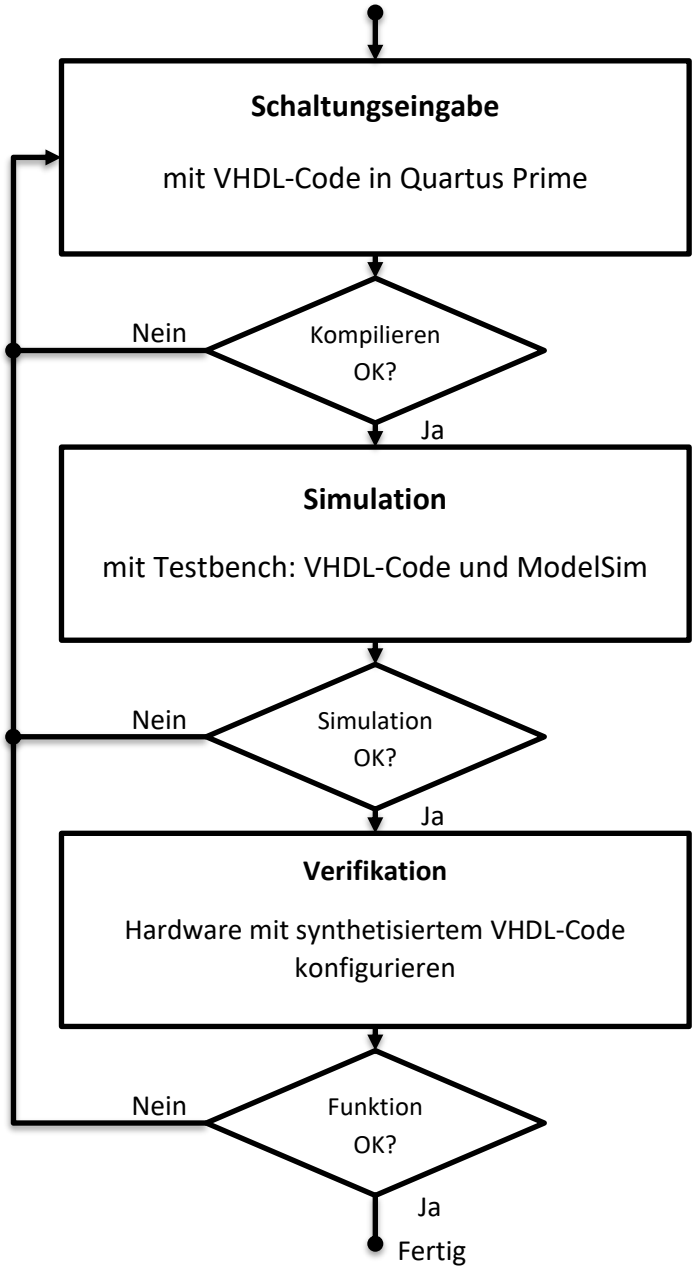
```
with selection select
signal_name <=
expression when choices,
expression when choices,
expression when others;
```

Beispiel:

```
with S select
       Z <= A when “00“,
            B when “11”,
            C when others;
```

4 FPGA Entwurfsablauf

Stark vereinfacht erfolgt der Entwurfsablauf nach folgendem Schema:



Im ersten Schritt wird das Verhalten der gewünschten Digitalschaltung beschrieben, beispielsweise mit der Hardwarebeschreibungssprache VHDL. Dazu verwenden wir das Programmpaket Quartus Prime. Im zweiten Schritt erfolgt die Simulation nach Kriterien, die man selbst festlegt. Die Testkriterien werden in einer sogenannten Testbench definiert, ebenfalls mit der Beschreibungssprache VHDL. Die Simulation erfolgt mit dem Programm ModelSim, welches die Simulationsergebnisse in grafischer Weise aufbereitet. Erst wenn das Ergebnis den Erwartungen entspricht, synthetisiert man den Code aus dem ersten Schritt und konfiguriert den FPGA mit dem Syntheseergebnis. Danach prüft man die Funktion erneut, nun mit der echten Hardware.

5 VHDL-Testbench

6 VHDL-Prozess

Die bisher verwendeten VHDL-Signalzuweisungen waren nebenläufig, d.h. jede Anweisung wurde unabhängig voneinander sofort ausgeführt, wenn sich ein Signal auf der rechten Seite der Signalzuweisung änderte.
Die Reihenfolge, in der die Anweisungen im VHDL-Code standen, hatte keinen Einfluss auf das Ergebnis.

Ausschließlich mit nebenläufigen Anweisungen lassen sich nur einfache Hardwareblöcke realisieren.

Eine sequenzielle Abarbeitung von Anweisungen wird in VHDL mit einem Prozess ermöglicht:

```
[<process_label>]: process [ (sensitivity list) ]
  -- hier können lokale Signale oder Variablen deklariert werden
begin
  -- Sequentielle Anweisungen oder unbedingte Signalzuweisungen
end process [<process_label>];
```

Prozesse weisen folgende Eigenschaften auf:

- Alle Prozesse innerhalb einer `architecture` laufen parallel ab
- Ein Prozess wird aktiviert, wenn sich ein Signal in der Sensitivity-List ändert
- Im Ausführungsteil von Prozessen (zwischen `begin` und `end process`) sind nur unbedingte Signalzuweisungen und spezielle sequenzielle Anweisungen erlaubt. Bedingte und selektive nebenläufige Anweisungen dürfen innerhalb von Prozessen nicht verwendet werden
- Die Zuweisung der tatsächlichen Signalwerte eines Prozesses erfolgt am Ende eines Prozesses, also durch die `end process`-Anweisung. D.h. es ist erlaubt, im Prozess einen bestimmten Signalwert zuzuweisen, der jedoch durch Überschreiben mit einem anderen Signalwert wieder verworfen wird. Der tatsächlich angenommene Wert ist der zuletzt im Prozess zugewiesene.

Man unterscheidet zwei Typen von Prozessen: Kombinatorische und synchron getaktete Prozesse:

6.1 Kombinatorischer Prozess

Kombinatorische Prozesse haben in der Sensitivity-Liste **alle** Eingangssignale und im Inneren deren Verknüpfung.

Beispiel:

```
comb_p: process(A, B, C)
begin
  X <= (A and B) or C;
end process comb_p;
```

Bei kombinatorischer Logik muss sichergestellt sein, dass jede Änderung eines Eingangssignals auch zu einer potentiellen Änderung der Ausgangssignale führt, also den jeweiligen Prozess aktiviert. Dies ist die Aufgabe der Sensitivity-Liste:

Bei VHDL-Prozessen, die kombinatorische Logik beschreiben, müssen alle Signale, die auf der rechten Seite einer Signalzuweisung stehen, in der Sensitivity-Liste des Prozesses vorkommen.

6.1.1 Ungewolltes Speicherverhalten

Die Syntax der `if`-Anweisung erlaubt auch eine unvollständige Spezifikation. d.h. im Code werden z.B. durch Weglassen des `else`-Zweigs nicht alle Verzweigungsmöglichkeiten explizit aufgeführt. Das bedeutet jedoch, dass die Ausgangssignale ihren alten Wert beibehalten, also ein Speicherverhalten aufweisen → ungewolltes Latch.

Bei der Modellierung von kombinatorischer Logik ist ein Speicherverhalten nicht sinnvoll. Leider wird bei unübersichtlich ineinander geschachtelten `if`-Anweisungen häufig der Fehler gemacht, dass einzelne Verzweigungsmöglichkeiten nicht berücksichtigt wurden und ein unerwünschtes Speicherverhalten auftritt. Dies kann vermieden werden, wenn den Ausgangssignalen vor der ersten `if`-Anweisung ein Defaultwert zugewiesen wird:

```
demo_p: process(S, A, B)
begin

  Y <= '0'; -- Defaultzuweisung

  if S = "00" then
    Y <= A and B;
  elsif S="01" then
    Y <= A or B;
  end if;
end process demo_p;
```

Im obigen Prozessbeispiel deckt die if-Anweisung offensichtlich nicht alle Möglichkeiten ab. Ein ungewolltes Speicherverhalten wird jedoch durch die Defaultzuweisung verhindert.

6.2 Synchron getakteter Prozess

Synchron getaktete Prozesse haben in der Sensitivity-Liste nur Takt- und Reset-Signale. Im Inneren wird beschrieben, welches Signal oder auch welche Verknüpfung von Signalen zur aktiven Taktflanke am Ausgang übernommen werden soll.

Beispiel für einen synchron getakteten Prozess (D-FF inkl. asynchronem, low-aktivem Reset):

```
ff1_p: process(nRes, clk)
 begin
   if (nRes ='0') then
      q <= '0';
   elsif (clk'event and clk='1') then
      q <= d;
   end if;
 end process ff1_p;
```

Beispiel für einen synchron getakteten Prozess mit synchronen Steuersignalen (spreset, enable):

```
ff2_p: process (clk)
begin
  if (clk'event and clk = '1') then
    if spreset ='1' then q <= '1';
    elsif enable = '1' then q <= d;
    end if;
  end if;
end process ff2_p;
```

7 Datentypen in VHDL

Im Skriptum wurde bisher ausschließlich der Datentyp `std_logic` verwendet. Dieser Datentyp wird mit dem Package „`ieee.std_logic_1164`“ zur Verfügung gestellt und ist ein Industriestandard.

Der Datentyp `std_logic` ermöglicht 9 unterschiedliche Signalwerte:

Wert	Bedeutung	Verwendung
‘U‘	Nicht initialisiert	Das Signal ist im Simulator (noch) nicht initialisiert
‘X‘	Undefinierter Pegel	Simulator erkennt mehr als einen aktiven Signaltreiber (Buskonflikt)
‘0‘	Starke logische 0	Low-Pegel eines Standardausgangs
‘1‘	Starke logische 1	High-Pegel eines Standardausgangs
‘Z‘	Hochohmig bzw. floatend	Für Busse mit Three-State-Ausgang (4. Jahrgang)
‘W‘	Schwach unbekannt	Simulator erkennt Buskonflikt zwischen schwachen L- und H-Pegeln
‘L‘	Schwacher L-Pegel	Open-Source-Ausgang mit Pull-Down-Widerstand
‘H‘	Schwacher H-Pegel	Open-Drain-Ausgang mit Pull-Up-Widerstand
‘–‘	Don’t-Care	Logikzustand des Ausgangssignals bedeutungslos, kann für Minimierung verwendet werden

In VHDL stehen weitere Datentypen zur Verfügung. Die folgende Übersicht listet die wichtigsten auf:

Datentyp	
`integer`	Ganze Zahl, vorzeichenbehaftet Ohne „range“-Angabe erlaubt sie einen minimalen Wertebereich von $-(2^{31})$ bis $(2^{31}-1)$ z.B.: `signal temp : integer range -16 to 15;`
`natural`	Ganze Zahl >= 0 (Untertyp von integer)
`boolean`	definiert `true` und `false`
`bit`	definiert ‘0‘ und ‘1‘
`bit_vector`	definiert ein ein-dimensionales Array mit Elementen des Datentyps `bit`
`std_logic_vector`	definiert ein ein-dimensionales Array mit Elementen des Datentyps `std_logic`

7.1 Vektoren

Vektoren fassen mehrere Objekte des gleichen Typs zusammen (ein-dimensionales Array) und ermöglichen die Definition wahlweise mit auf- oder absteigendem Index. Der Index eines Vektors ist eine Integerzahl.

Beispiel:

```
signal reg: std_logic_vector(15 downto 0);
signal a:   std_logic_vector(0 to 3);
```

7.1.1 Zuweisungen und Verküpfungen

Mit Vektoren ist es besonders einfach, mit nur einer Anweisung parallel eine Operation auf alle Einzelsignale (oder eine Teilmenge davon) auszuführen. Das Beispiel zeigt sowohl die Deklaration von Vektoren, als auch unterschiedliche Möglichkeiten der Wertzuweisungen.

Beispiele:

```
signal a, b, c: std_logic_vector(3 downto 0);

c <= a or b;                -- bitweises ODER auf alle Elemente
c <= ('1', '0', '1', '0'); -- Zuweisung als Aggregat
c <= "1010";                -- Zuweisung als Bitstring
c <= x"A";                  -- Zuweisung als Hex-Zahl
c <= a(0) & "001";          -- Zusammensetzung mit &
c(2 downto 1) <= "11";      -- Zuweisung zu Teilvektor
c <= a(1 downto 0) & '1' & b(0);-- Auswahl eines Teilvektors
(w,x,y,z) <= c;       -- Zuweisung von Vektor zu Einzelsignalen
c <= (others => '0'); -- allen Einzelsignalen von c wird '0'
                         zugewiesen
c <= (3 downto 2 => "01", 1 => a(3), others => '0'); -- others
                      steht für alle übrigen Indizes und ist
                      als letztes in der Liste platziert.
```

Concatenation Operator:

Um kürzere Vektoren oder Einzelelemente zu einem größeren Vektor zusammenzuführen, steht der Operator „&“ zur Verfügung:

```
data <= "00" & a(7 downto 2);
```

Verschiedene Darstellungen von Zuordnungen:

```
signal d: std_logic_vector(7 downto 0);
d <= "01001001";    -- diese und die beiden folgenden
d <= "0100_1001";   -- Zuweisungen haben gleiche Wirkung
d <= x"49";
```

Übung: Inhalt von a=“0110“, b=“1010“, welchen Wert haben die Zuweisungen in der jeweiligen Zeile?

```
c <= a or b;
c <= ('1', '0', '1', '0');
c <= "1010";
c <= x"A";
c <= a(0) & "001";
c(2 downto 1) <= "11";
c <= a(1 downto 0) & '1' & b(0);
(w,x,y,z) <= c;
c <= (others => '0');
c <= (3 downto 2 => "01", 1 => a(3), others => '0');
```

Übung: Shift Left (SHL), Shift Right (SHR), Rotate Left (ROL), Rotate Right (ROR)

```
signal e: std_logic_vector(7 downto 0);
```

SHL: Bits werden um eine Position nach links verschoben, 0 rückt nach

SHR: Bits werden um eine Position nach rechts verschoben, 0 rückt nach

ROL: Bits werden um eine Position nach links verschoben, rausfallendes Bit rückt nach

ROR: Bits werden um eine Position nach rechts verschoben, rausfallendes Bit rückt nach

7.2 Aufzähltypen

Für z.B. Zustandsautomaten ist es sinnvoll, eigene Aufzähltypen zu definieren.

z.B.:

```
Type M_STATE is (M_reset, M_idle, M_run);
```

Der Compiler ordnet beim Übersetzen entsprechend Bitmuster zu, z.B. M_reset -> „00“,
M_idle -> „01“ und M_run -> „10“.
Die Anzahl der Bits ergibt sich daher aus der Anzahl der zu codierenden Zustände.

7.3 Konvertierungsfunktionen

Um zwischen den einzelnen Datentypen konvertieren zu können, stellt VHDL entsprechende Konvertierungsfunktionen zur Verfügung:

Konversionsfunktion	ARG1	ARG2	Ergebnistyp
`to_integer(ARG1)`	unsigned signed	- -	integer integer
`unsigned(ARG1)`	signed std_logic_vector	- -	unsigned unsigned
`to_unsigned(ARG1, ARG2)`	natural integer	Anzahl der Bits im Ergebnis	unsigned
`signed(ARG1)`	unsigned std_logic_vector	- -	signed signed
`to_signed(ARG1, ARG2)`	integer	Anzahl der Bits im Ergebnis	signed
`resize(ARG1, ARG2)`	signed unsigned	Anzahl der Bits im Ergebnis	signed unsigned
`to_bit(ARG1)`	std_ulogic std_logic	- -	bit bit
`to_stdulogic(ARG1)`	bit	-	std_ulogic std_logic
`to_bitvector`	std_ulogic_vector std_logic_vector	- -	bit_vector
`to_stdulogicvector(ARG1)`	bit_vector std_logic_vector	- -	std_ulogic_vector
`to_stdlogicvector`	bit_vector std_ulogic_vector	- -	std_logic_vector

Quelle: DI Wolfgang Bodner

8 Operatoren

Die folgende Tabelle gibt einen Überblick, welche Operationen in VHDL zur Verfügung stehen und bei welchen Datentypen sie angewendet werden können:

Operator	Description	Data type of a	Data type of b	Data type of result
`a ** b`	exponentiation	int	int	int
`abs a`	absolute value	int		int
`a * b`	multiplication	int	int	int
`a / b`	division	int	int	int
`a mod b`	modulo	int	int	int
`a rem b`	remainder	int	int	int
`+ a`	identity	int		int
`- a`	negation	int		int
`a + b`	addition	int	int	int
`a - b`	subtraction	int	int	int
`a & b`	concatenation	array, element	array, element	array
`a sll b`	shift left log.	bitv	int	bitv
`a srl b`	shift right log.	bitv	int	bitv
`a sla b`	shift left arith.	bitv	int	bitv
`a sra b`	shift right arith.	bitv	int	bitv
`a rol b`	rotate left	bitv	int	bitv
`a ror b`	rotate right	bitv	int	bitv

a = b	equal to	any	same as a	boo
a /= b	not equal to	any	same as a	boo
a < b	less than	scalar, array	same as a	boo
a <= b	l. th. or eq. to	scalar, array	same as a	boo
a > b	greater than	scalar, array	same as a	boo
a >= b	g. th. or eq. to	scalar, array	same as a	boo
not a	negation	boo, bit, bitv		same as a
a and b	and	boo, bit, bitv	same as a	same as a
a or b	and	boo, bit, bitv	same as a	same as a
a xor b	and	boo, bit, bitv	same as a	same as a
a nand b	and	boo, bit, bitv	same as a	same as a
a nor b	and	boo, bit, bitv	same as a	same as a
a xnor b	and	boo, bit, bitv	same as a	same as a

Quelle: Dr. Wess (TGM)

9 State Machines in VHDL

10 Metastabilität

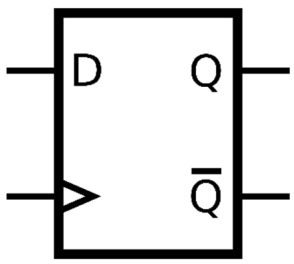
Metastabilität ist ein unerwünschtes Phänomen von synchronen Schaltungen.

10.1 Problematik

Alle digitalen Speicherelemente (Flip-Flops) haben bestimmte Anforderungen an die zeitliche Abfolge der Eingangssignale, damit diese korrekt verarbeitet werden.

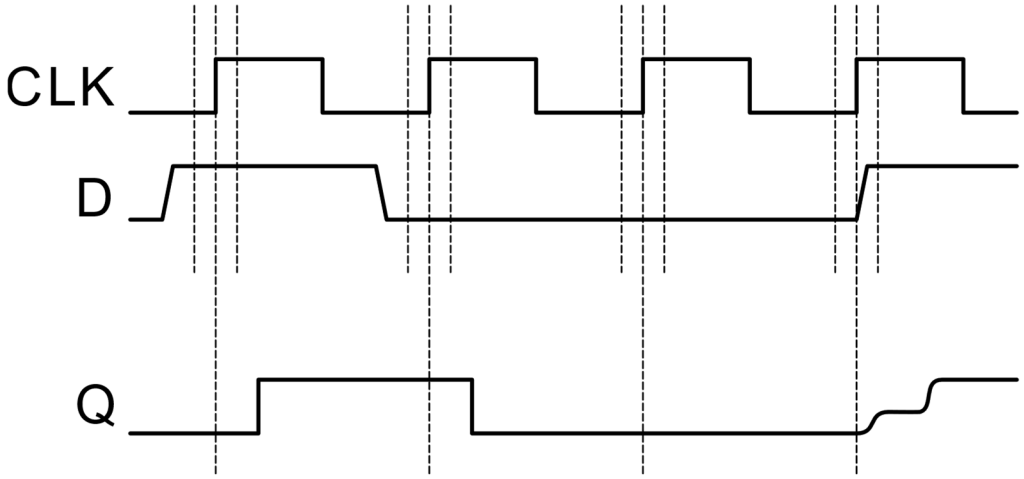
Die Timing-Angaben am Beispiel eines D-Flip-Flops:

- Daten am D-Eingang müssen schon kurze Zeit vor der aktiven CLK-Flanke stabil sein, diese Dauer wird mit der „Register Setup Time“ bzw. $\mathbf{t_{su}}$ angegeben
- Daten am D-Eingang müssen auch nach der steigenden CLK-Flanke kurzzeitig Zeit stabil sein, diese Dauer wird der „Register Hold Time“ bzw. $\mathbf{t_H}$ angegeben
- Der Ausgang Q ändert sich erst kurze Zeit nach der aktiven Clock-Flanke, diese Dauer wird mit der „Clock-to-Output Time“ bzw. $\mathbf{t_{CO}}$ angegeben



Quelle: de.wikipedia.org

Beispielhaftes Timing-Diagramm:



Bei den aktiven Taktflanken 1-3 wurden die Anforderungen an das Eingangssignal eingehalten, somit nimmt das Ausgangssignal Q einen definierten Zustand ein. Bei der vierten aktiven Taktflanke wurde die Timing-Anforderungen an das Eingangssignal nicht eingehalten, daher nimmt das Ausgangssignal Q kurzzeitig einen undefinierten Zwischenzustand ein, auch metastabiler Zustand genannt. Nachfolgende Logikgatter geben höchstwahrscheinlich ebenfalls undefinierte Werte aus, was dazu führen kann, dass beispielsweise ein Zustandsautomat einen Zustand einnimmt, aus dem kein Weg mehr herausführt, außer durch Anlegen eines Reset-Signals.

10.2 Auswirkungen

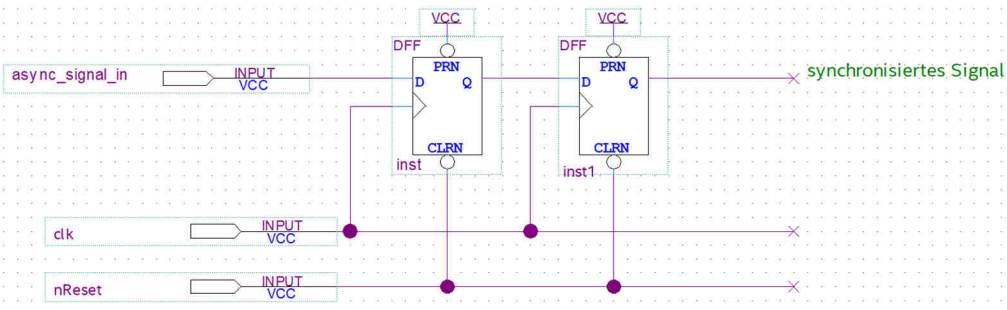
Eingangssignale von Zustandsautomaten, wie beispielweise die Signale eines Inkrementalgebers, können jederzeit den Zustand wechseln, ohne Berücksichtigung der Timing-Anforderungen bezüglich des Clock-Signals und der internen Flip-Flops einer Digitalschaltung. Aus Sicht des Zustandsautomaten sind diese Eingangssignale asynchron, mit der nun bekannten Problematik der Metastabilität.

10.3 Abhilfe

Abhilfe kann geschaffen werden, indem die zuerst asynchronen Signale in einen korrekten zeitlichen Bezug zum Clock-Signal gesetzt werden. Eine geeignete Schaltungsmaßnahme ist eine zweistufige Synchronisierungskette, auch Synchronisation Chain genannt.

Die Funktionsweise beruht darauf, dass ein eventuell metastabiles Ausgangssignal des ersten (linken) D-Flip-Flops bei der nächsten aktiven Taktflanke schon längst wieder einen stabilen Wert angenommen hat, so dass am zweiten (rechten) D-Flip-Flop zum Zeitpunkt der aktiven Taktflanke nur stabile Werte anliegen, unter Einhaltung der Timing-Anforderungen.

Für jedes asynchrone Eingangssignal muss eine Synchronisierungskette vorgesehen werden.



11 Addierschaltungen

11.1 Ripple Carry Adder

11.1.1 Wiederholung

Ein N-bit Ripple Carry Adder wird mit N hintereinander geschalteten Volladdierern realisiert.

Zur Wiederholung des Volladdierers und des Ripple-Carry-Addierers siehe DIC-Skriptum 2. Klasse.

Zu beachten ist, dass jedes Gatter eine Signallaufzeit hat, z.B. 6 ns, abhängig von der verwendeten Halbleitertechnologie. Bei einem Volladdierer liegen bis zu drei Gatter in Serie, also benötigt ein Volladdierer 18ns, bis das richtige Ergebnis am Ausgang anliegt.

Würde man aus dem obigen Volladdierer beispielsweise einen 64-bit Ripple Carry Addierer aufbauen, wäre die Signallaufzeit, bis das endgültige Ergebnis vom letzten Volladdierer ausgegeben wird, 64*3*6ns = 1,15µs (eine Ewigkeit in der Digitaltechnik).

11.1.2 Signallaufzeit in VHDL

In VHDL kann man die Signallaufzeit berücksichtigen, indem man folgende Syntax benutzt:

```
Y <= A and B
```

Diese Anweisung bewirkt, dass das Ergebnis von (A and B) um 6 ns verzögert dem Ausgang Y zugewiesen wird.

Wichtig ist zu beachten, dass VHDL-Code, welcher das Schlüsselwort „after“ benutzt, nicht synthetisierbar ist. Genauso verhält es sich mit den Schlüsselwörtern „wait“ bzw. „wait for“. Das bedeutet, dass der Quellcode zwar simuliert werden kann, aber nicht in eine FPGA-Hardware eingespielt werden kann.

11.1.3 Projekt: RippleCarryAdder

Die Aufgabe besteht darin, einen 4-bit Ripple Carry Adder in VHDL zu realisieren. Als erster Schritt muss der Full Adder mit VHDL beschrieben werden (FullAdder.vhd), wobei die Zuweisung von jedem Rechenschritt um 6 ns verzögert werden soll.

Dateibezeichnungen

Projektname:	RippleCarryAdder
Top Sheet:	RippleCarryAdder.bdf
VHDL:	FullAdder.vhd
	tb_RippleCarryAdder.vhd
DO:	tb_RippleCarryAdder.do

VHDL-Code eines FullAdders:

I/O-Bezeichnungen

FullAdder.vhd:

Eingänge:	A, B, CI
Ausgänge:	SUM, CO

RippleCarryAdder.bdf:

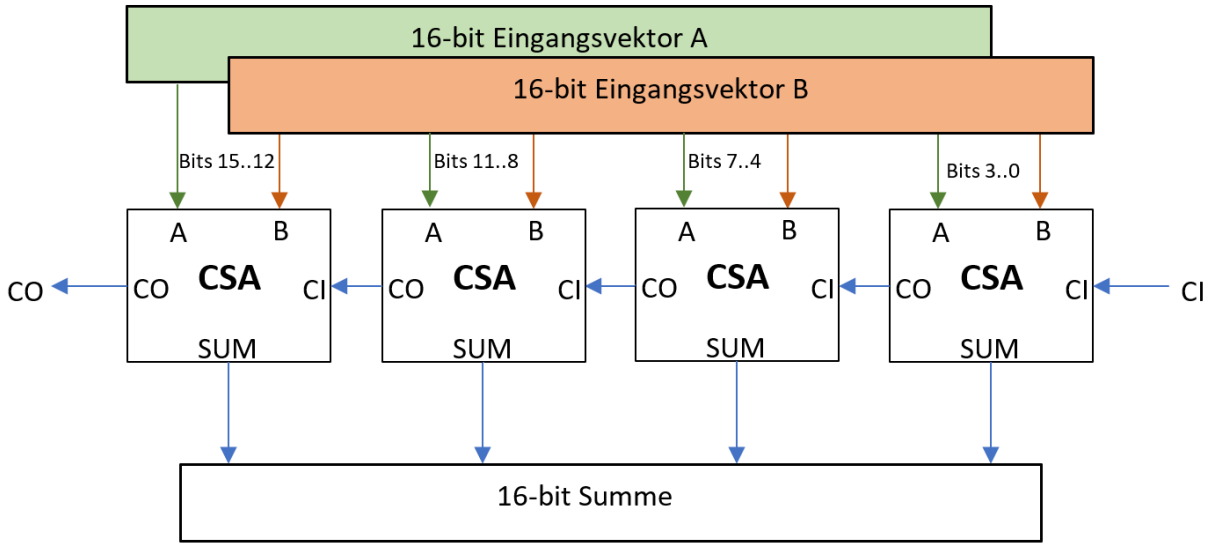
Eingänge:	A3 A2 A1 A0
	B3 B2 B1 B0
	CI (Carry-In der ersten Stufe)
Ausgänge:	SUM3 SUM2 SUM1 SUM0
	CO (Carry-Out der letzten Stufe)

11.2 Carry Select Adder

Ein Carry-Select-Addierer (CSA) unterteilt die zu addierenden Wortbreiten in kleinere Blöcke, und berechnet parallel die Teilsummen für CI = 0 sowie CI = 1. Das Ergebnis (SUM, CO) wird dann entsprechend dem tatsächlichen Wert von CI ausgegeben.

Dadurch wird die Gesamtsignallaufzeit gegenüber einem reinen Ripple-Carry-Addierer deutlich reduziert, weil alle 4-bit-Teilsummen sowohl mit CI = 0 als auch mit CI = 1 parallel berechnet werden.

Prinzipskizze eines 16-bit Addieres, bestehend aus 4x 4-bit Carry-Select-Addierern:

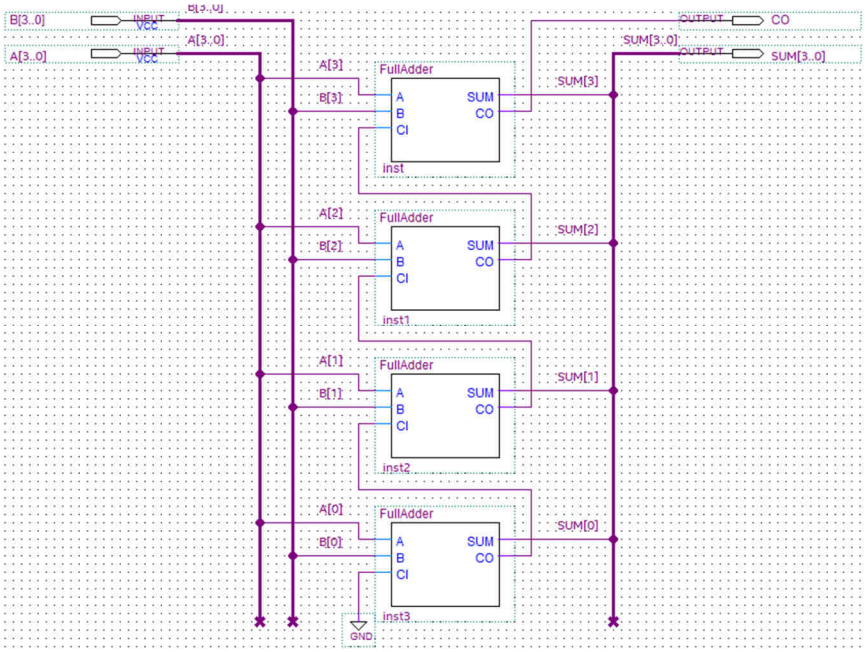


11.2.1 Innerer Aufbau eines 4-bit CSA-Blocks

Ein 4-bit Carry-Select-Adder Block besteht aus zwei Ripple-Carry-Addierern, welche zwei 4-bit Zahlen addieren. Ein Ripple-Carry-Adder rechnet mit CI = 0, der andere Ripple-Carry-Adder rechnet mit CI = 1.

12 VHDL-Instantiierung

Bisher haben wir mit Quartus Prime immer ein grafisches Top-Sheet gezeichnet, wenn ein Projekt aus mehreren VHDL-Modulen zusammengesetzt wurde, wie z.B. der RippleCarryAdder:



Dieser 4-Bit-Ripple-Carry-Addierer verwendet vier Instanzen des FullAdders, welche in geeigneter Weise verdrahtet sind.

Abbildung 1: Grafische Darstellung eines 4-bit-Ripple-Carry-Addierers

Aber es besteht auch die Möglichkeit, mittels VHDL Code andere, bestehende Module einzubinden. Anders gesagt, was man in Quartus Prime mit der grafischen Darstellung machen kann, kann man ebenfalls rein in VHDL beschreiben.

Als Beispiel wollen wir den oben dargestellten 4-bit-Ripple-Carry-Addierer nicht grafisch, sondern rein in VHDL realisieren. Wie gewohnt schreibt man zuoberst einen Headerblock und danach die Library-Einbindung. Dann beginnt man wie üblich mit der Entity-Beschreibung, welche die Schnittstellen nach außen definiert:

```
entity RippleCarryAdder IS
    port (
        A     :   in    std_logic_vector(3 downto 0);
        B     :   in    std_logic_vector(3 downto 0);
        CI    :   in    std_logic;
        SUM   :   out   std_logic_vector(3 downto 0);
        CO    :   out   std_logic
    );
end entity RippleCarryAdder;
```

Bis jetzt also nicht neues. Weiter geht's mit der architecture:

```
architecture RippleCarryAdder_arch of RippleCarryAdder is
```

Nun kommt eine Neuigkeit. Wir müssen auf den bestehenden FullAdder und dessen Anschlussbelegung verweisen. In der Praxis macht man das so, dass man im Code-Editor die Entity-Deklaration des FullAdders in die zu schreibende Architecture rüberkopiert und die Schlüsselwörter „entity" durch „component" ersetzt:

```
    entity component FullAdder is
        port (
            A, B, CI    :     in    std_logic;
            SUM, CO     :     out   std_logic
        );
     end component FullAdder;
```

Nun brauchen wir noch die drei Carry-Leitungen, die intern das Carry-Signal von einem FullAdder zum Nächsten weiterleiten, damit ist der Teil vor „begin“ fertig:

```
    signal  carry01 :  std_logic; -- Carry-Signal von FA0 zu FA1
    signal  carry12 :  std_logic; -- Carry-Signal von FA1 zu FA2
    signal  carry23 :  std_logic; -- Carry-Signal von FA2 zu FA3

begin
```

Nun erstellen wir die einzelnen Instanzen der FullAdder, und verbinden sie mit den jeweiligen Eingängen / Ausgängen / Signalen:

```
      FA0 : FullAdder port map(
            A => A(0),
            B => B(0),
            CI => '0',
            SUM => SUM(0),
            CO => carry01
      );
```

Dieser Code ist so zu verstehen:

- Erzeuge eine Instanz von FullAdder mit dem Namen FA0 und beginne mit dem „Verdrahten“:

  ```
  FA0 : FullAdder port map
  ```

- Verdrahte den Anschluss A des FullAdders mit dem Bit 0 des Busses A der RippleCarryAdder-Entity:

  ```
  A => A(0),
  ```

- Das selbe mit dem Anschluss B:

  ```
  B => B(0),
  ```

- Das Carry-In Signal des 0. FullAdders liegt dauerhaft auf LOW:

  ```
  CI => '0',
  ```

- Die Summe wird auf Bit 0 des SUM-Busses des RippleCarryAdders gelegt:

  ```
  SUM => SUM(0),
  ```

- Das Carry-Out-Signal wird auf die Leitung „carry01“ gelegt:

  ```
  CO => carry01
  ```

- Abschließen mit Klammer-zu / Strichpunkt:

  ```
  );
  ```

Nun die weiteren FullAdder-Instanzen:

```
    FA1 : FullAdder port map(
          A => A(1),
          B => B(1),
          CI => carry01,
          SUM => SUM(1),
          CO => carry12
    );

    FA2 : FullAdder port map(
          A => A(2),
          B => B(2),
          CI => carry12,
          SUM => SUM(2),
          CO => carry23
    );

    FA3 : FullAdder port map(
          A => A(3),
          B => B(3),
          CI => carry23,
          SUM => SUM(3),
          CO => CO
    );
```

Und damit ist der RippleCarryAdder vollständig beschrieben. Der Code-Teil nach „begin“ der RippleCarryAdder-Architektur besteht also nur aus Instanziierungen von FullAddern und deren Verdrahtung. Abschließend braucht es noch

```
end architecture RippleCarryAdder_arch;
```

13 Generics

Angenommen, man benötigt in einem Design mehrere Multiplexer für Bussignale mit unterschiedlicher Bitanzahl: 4, 9, 16, 20 und 35 Bit. Nun könnte man für jede Busbreite einen eigenen Multiplexer schreiben, also würde man 5 unterschiedliche Entities und Architectures benötigen. Mit Hilfe von sogenannten Generics braucht man nur einen Multiplexer zu beschreiben. Erst bei der Instanziierung, egal ob grafisch oder mit VHDL, gibt man dann an, mit welcher Bitanzahl der jeweilige Multiplexer realisiert werden soll.

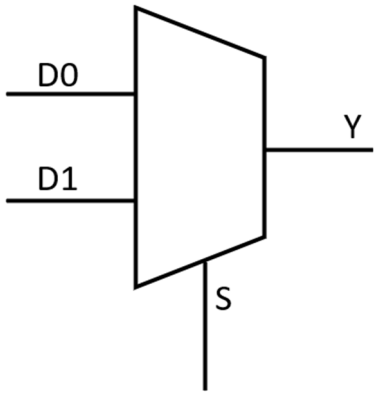


Abbildung 2: 2-zu-1 Multiplexer

Als erstes Beispiel betrachten wir einen Multiplexer mit zwei Eingängen, „D0“ und „D1“. Aus wie vielen Bit die Eingänge D0 und D1 bestehen soll, soll frei definierbar sein. Mit dem „S“-Eingang wird gewählt, welchen der beiden Eingangswerte der Multiplexer auf dem Ausgang „Y“ ausgibt.

Im folgenden VHDL-Code für den generischen 2-zu-1-Multiplexer wird mit dem Schlüsselwort **generic** der Generic-Wert „N“ vom Datentyp integer mit dem Standardwert von 4 innerhalb der Entity definiert. Dieser Wert wird bei der Definition der Vektorbreiten für die Ein- und Ausgänge D0, D1 und Y verwendet:

```
entity Mux2to1_Nbit is

      generic(N : integer := 4); -- Default-Wert von N ist 4

      port(
            D0, D1      :     in    std_logic_vector(N-1 downto 0);
            S           :     in    std_logic;
            Y           :     out   std_logic_vector(N-1 downto 0)
      );
end entity Mux2to1_Nbit;


architecture Mux2to1_Nbit_arch of Mux2to1_Nbit is
begin

      Y <= D0 when S = '0' else D1;

end architecture Mux2to1_Nbit_arch;
```

13.1 Grafische Instanziierung

Verwendet man diesen Multiplexer nun in Quartus Prime in seiner grafischen Darstellung, erscheint ein zusätzlicher Block, welcher den aktuellen Wert für N anzeigt. Die Linie zeigt an, zu welcher Instanz der Parameterblock gehört.

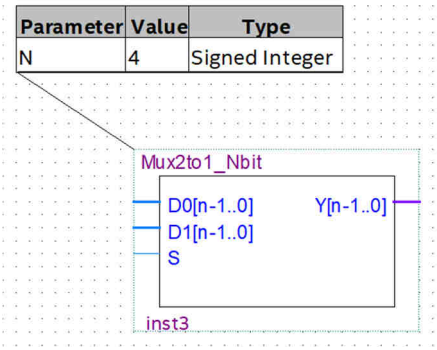


Abbildung 3: Instanz mit Parameterblock

Um den Standardwert von 4 abzuändern, führt man entweder einen Doppelklick auf den Parameterblock aus, oder Rechtsklick / Properties. Im erscheinenden Fenster wechselt man auf den dritten Tab, „Parameters“, und kann hier den Wert ändern:

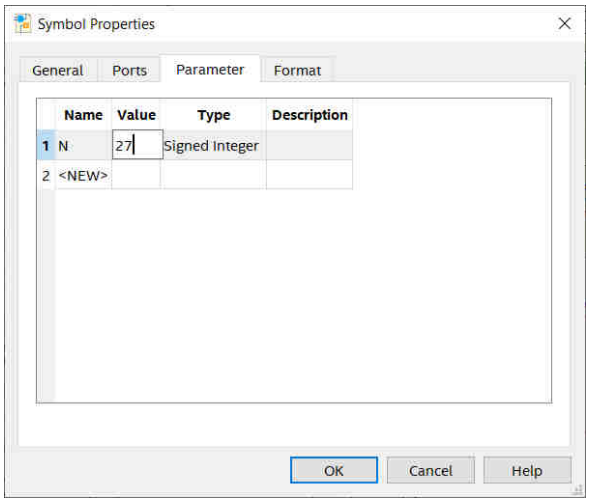


Abbildung 4: Ändern des Standardwerts für den Generic-Wert N

13.2 VHDL-Instanziierung

Will man den Multiplexer in VHDL instanziieren (d.h. verwenden), ist folgende Syntax erforderlich:

- Zwischen **architecture** und **begin** muss auf den bestehenden Mux2to1_Nbit verwiesen werden (Entity von Mux2to1_Nbit kopieren und 2x die Schlüsselwörter „**entity**“ durch „**component**“ ersetzen
- Bei der Instanziierung (nach **begin**) benötigt man zusätzlich ein „**generic map**“ Kommando, wo man den gewünschten Wert für N angibt

```
architecture ... of ... is

	component Mux2to1_Nbit is
		generic(N : integer := 4);	-- Default-Wert von N ist 4
		port(
			D0, D1		:	in	std_logic_vector(N-1 downto 0);
			S		:	in	std_logic;
			Y		:	out	std_logic_vector(N-1 downto 0)
		);
	end component Mux2to1_Nbit;

begin
	-- Hier werden zwei Instanzen von Mux2to1_Nbit erstellt:
	Mux27bit : Mux2to1_Nbit generic map(N => 27) port map(
		D0 => ...,
		D1 => ...,
		S => ...,
		Y => ...
	);

	Mux12bit : Mux2to1_Nbit generic map(N => 12) port map(
		D0 => ...,
		D1 => ...,
		S => ...,
		Y => ...
	);

	...

end architecture ...;
```