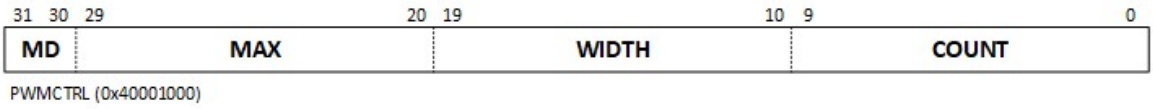
Numerik in VHDL

1 Einleitung

In Schaltungen der Digitalelektronik werden häufig Zahlen benötigt. Typische Anwendungen sind Zähler oder Filter (digitale Signalverarbeitung). Die Besonderheit bei der Verwendung von Zahlen in VDHL besteht darin, dass zwischen dem *Zahlenwert* und der *technischen Realisierung* einer Zahl unterschieden wird.

Manchmal möchte man aber sowohl auf den Zahlenwert als auch auf die einzelnen Stellen der zugehörigen Dualzahl zugreifen können. Ein typisches Beispiel ist ein Peripherieregister eines Mikrocontrollers, das mehrere Felder enthält (Abbildung 1).



PWMCTRL (0x40001000)

Position	Field Name	Description
31:30	MD	Mode
	00	PWM off (counter stopped, output set to 0)
	01	PWM off (counter set to 1 and stopped, output depends on WIDTH)
	10	PWM on (counter running, output depends on WIDTH)
	11	reserved
29:20	MAX	Maximum
	1-1000	The counter will be reset to 1 after reaching the MAX value
19:10	WIDTH	Pulse Width
	0-1000	PWM output set to 1 when COUNT<WIDTH, else set to 0
9:0	COUNT	Current Counter Value
	1-1000	Contains the current value of the counter (read only)

Abbildung 1: Register PWMCTRL (Pulse Width Modulation Control)

In diesem Beispiel besteht das Register aus 32 Bit, von denen die obersten beiden (31 und 30) zusammen ein Feld bilden, das der Steuerung dient. Diesem Feld ist kein Zahlenwert zugeordnet. Die drei anderen Felder (MAX, WIDTH und COUNT) werden bei der Funktionsbeschreibung als Zahlenwerte interpretiert. Für die Programmierung ist zusätzlich die Position der einzelnen Werte innerhalb des Registers wichtig. Eine mögliche Lösung zum Einstellen einer neuen Pulsweite zeigt Listing 1.

```
void set_pulsewidth(uint16_t w)
{
   uint32_t* p=(uint32_t*)0x40001000;
   uint32_t  r=*p;
   r &= ~(0x3ff)<<10;
   r |= w<<10;
   *p = r;
}
```

```
// w contains the new pulse width

// let p point to the peripheral register
// get current values
// clear the WIDTH field only
// set the WIDTH field to new value
// write new values back to the register
```

Listing 1: C-Funktion zum Setzen einer neuen Pulsweite

Kapitel 2 beschreibt allgemein, wie in VHDL ein Zahlenwert einer technischen Realisierung zugeordnet werden kann, so dass man auf beides zugreifen kann.

In Kapitel 0 wird das Beispiel auf VHDL-Seite fortgesetzt.

2 Zahlendarstellung

Für VHDL gibt es im Sprachumfang enthaltene Typen (z.B. *bit*, *integer*) als auch Typen, die mit Hilfe eines Packages definiert werden. In der Vergangenheit gab es dazu proprietäre Lösungen, die später durch eine Standardnumerik ersetzt wurde. Hier wird nur die Verwendung der Standardnumerik behandelt.

2.1 Packages

Für die Datentypen *bit* und *std_logic* sind zwei Standardpackages verfügbar (Tabelle 1).

Package	Inhalt
use ieee.numeric_std.all;	definiert Numerik für den Typ std_logic
use ieee.numeric_bit.all;	definiert Numerik für den Typ bit

Tabelle 1: Packages für Arithmetik mit den Datentypen bit und std_logic

In diesen Packages sind die nachfolgend verwendeten Typen *signed* und *unsigned* sowie zugehörige Funktionen und Operatoren deklariert und definiert. In der Praxis wird fast immer mit dem Typ *std_logic* gearbeitet. Aus diesem Grund bezieht sich die Beschreibung auf das Package *numeric_std*. Nur dieses Package wird dann auch im VHDL-Quelltext eingebunden. Sämtliche Zahlen werden technisch als Dualzahlen realisiert.

2.2 Datentypen

Tabelle 2 zeigt Beispiele für die Deklaration von Datenobjekten, die als ganze Zahlen interpretiert werden können.

Datentyp	Beispiel	Bemerkung
integer	*signal i: integer range -2 to 2;* *signal j; integer;*	ganze Zahl, vorzeichenbehaftet Wenn ohne Range deklariert, dann mindestens [-2^{31}, $2^{31}-1$]
natural	*signal i: natural range 0 to 15;*	ganze Zahl ≥ 0 (Untertyp des Integer)
positive	*signal i: positive range 2 to 6;*	ganze Zahl > 0 (Untertyp des Integer)
signed	*signal i: signed (15 downto 0);*	Feld, aber Zahleninterpretation als vorzeichenbehaftete Dualzahl möglich
unsigned	*signal i: unsigned (31 downto 0);*	Feld, aber dann Zahleninterpretation als vorzeichenlose Dualzahl möglich

Tabelle 2: Datentypen für die Zahlendarstellung

Bei den reinen Zahlentypen ist eine Bereichsangabe sinnvoll, weil der Compiler dann auch nur Dualzahlen mit der minimal nötigen Länge verwendet. Bei der späteren Schaltungssynthese wird dann auch nur Hardware für diese Länge erzeugt. Das spart nicht nur Kosten, es kann auch zu einer höheren Abarbeitungsgeschwindigkeit führen. Zudem können bei der Simulation Bereichsüberschreitungen erkannt werden.

Falls der Bereich nicht angegeben wird, wird mindestens eine Dualzahl mit 32 Bits unterstützt.

2.3 Operatoren

Für die Typen *signed* und *unsigned* sind einige Operatoren (Tabelle 3) definiert. Man sollte beachten, dass dabei jeweils eine ganze Schaltung erzeugt wird, d.h. für die Operation *y<=y+1;* wird ein ganzer Addierer mit n Stellen erzeugt (n: Anzahl der zur Darstellung von y benötigten Stellen). Ebenso erzeugt der Vergleich *if (y<6) ...* eine ganze Vergleichsschaltung für n Stellen. Das kann sehr schnell zu großen Schaltungen führen. Manche Operationen (Division) können (je nach VHDL-Compiler) gar nicht in eine Schaltung umgesetzt werden. Sie erfordern dann Speziallösungen. In einem Programm für einen µC ist dagegen die Schaltung für die Operationen nur einmal vorhanden, sie wird dann nacheinander benutzt.

Operator	Bedeutung	Beispiel
+, -	Addition/Subtraktion	i:=i+1; j=j-k;
*	Multiplikation	i:=i*2;
/, rem, mod	Division, Rest, Modulo	i.d.R. nur zur Simulation
<, <=, >,>=,=,/=	Vergleiche (Ergebnistyp *boolean*)	y <='1' when a /= b;

Tabelle 3: Operatoren für die Datentypen signed und unsigned bei numeric_std

Die Multiplikation kommt sehr häufig in der digitalen Signalverarbeitung vor. Dafür gibt es FPGAs, die einen begrenzten Vorrat von fertigen Multiplikationsschaltungen in Hardware haben. Das könnten (Stand 2020) z. B. 20 Multiplizierer sein, die sehr schnell je zwei 16 Bit Zahlen als Operanden haben können.

Dann kann der Compiler (er muss das natürlich wissen), bis zu 20 Multiplikationen auf diese Multiplizierer legen.

Divisionen können meist vermieden werden, daher gibt es dafür in der Regel auch keine Unterstützung in FPGAs.

2.4 Konversionen

Der Wechsel zwischen der technischen Realisierung und dem Zahlenwert erfolgt über Konversionsfunktionen. Abbildung 2 zeigt den Zusammenhang. Links ist nur eine technische Realisierung in Form eines Feldes mit 8 Elementen gegeben. Diesem Feld ist kein Zahlenwert zugeordnet, dafür kann auf jedes Element einzeln zugegriffen werden (rote Werte).

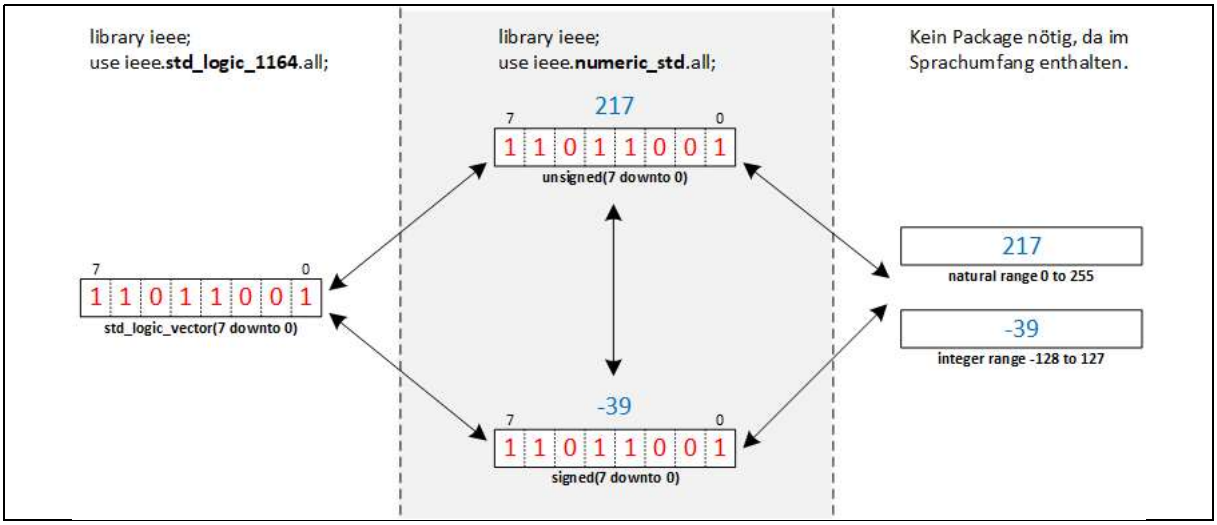


Abbildung 2: Konversion zwischen Datentypen

Rechts sind reine Zahlenwerte (blau) gegeben. Mit diesen Werten kann gerechnet werden, aber ein Zugriff auf einzelne Stellen der Zahl ist nicht möglich.

In der Mitte sind mit den neuen Typen *unsigned* und *signed* technische Realisierungen gegeben, aber der Feldinhalt kann als Zahl interpretiert werden. Daher kann jetzt auf einzelne Stellen zugegriffen werden und zusätzlich sind Rechnungen mit den Zahlenwerten möglich. Die in Tabelle 3 definierten Operatoren können ohne vorherige Umwandlung in einen Zahlentyp verwendet werden.

Da VHDL eine Sprache mit strenger Typisierung ist, muss jede Umwandlung eines Typs in einen anderen explizit beschrieben werden. Dazu gibt es eine Reihe von Konversionsfunktionen (Pfeile in Abbildung 2). Die Benutzung dieser Funktionen führt nicht zu mehr Schaltungsaufwand, es wird nur die Interpretation (*signed*, *unsigned*) bzw. die Sichtweise (Vektor, Zahl) geändert.

Tabelle 4 zeigt die Syntax der Konversionsfunktionen.

von	nach	Funktion
std_logic_vector	unsigned	unsigned()
std_logic_vector	signed	signed()
unsigned	std_logic_vector	std_logic_vector()
unsigned	signed	signed()
unsigned	integer, natural, positive	to_integer()
signed	std_logic_vector	std_logic_vector()
signed	unsigned	unsigned()
signed	integer	to_integer()
integer, natural, positive	unsigned	to_unsigned(,)
integer, natural, positive	signed	to_signed(,)

Tabelle 4: Typkonversionen für Zahlendarstellungen

Falls *s* ein std_logic_vector ist, der in ein Signal *u* des den Typ unsigned gleicher Länge gewandelt werden soll, schreibt man also: *u <= unsigned(s);*

Bei der Umwandlung von einer Ganzzahl in einen Vektor muss man noch die Anzahl der Stellen mit angeben, die von der Ganzzahl verwendet werden sollen. Die Anweisung `y<=to_unsigned(i,4);` legt die untersten 4 Stellen der Ganzzahl *i* in dem vierstelligen Vektor *y* ab.

2.5 Resize

Manchmal möchte man eine Zahl in einem Vektor *x* in einen Vektor *y* mit anderer Länge kopieren, um dann mit der neuen Stellenzahl weiterzuarbeiten. Dazu gibt es die Funktion *resize(src, n)*. Der Parameter *src* ist dabei die Zahlendarstellung in der Quelle (*signed*, *unsigned*) und *n* die Zahl der Stellen im Ziel. Das Ergebnis hat denselben Typ wie die Quelle. Dabei können vier Fälle auftreten (Tabelle 5).

Fall	Operation
unsigned, Ziel hat mehr Stellen als die Quelle	Kein Problem, die neuen Stellen werden mit 0 gefüllt, der Zahlenwert bleibt erhalten.
signed, Ziel hat mehr Stellen als die Quelle	Kein Problem, die neuen Stellen werden mit der obersten Stelle (Vorzeichen) der Quelle gefüllt. Der Zahlenwert bleibt erhalten.
unsigned, Ziel hat weniger Stellen als die Quelle	Die überzähligen Stellen werden einfach verworfen. Falls diese Stellen nicht alle 0 sind, geht der Zahlenwert verloren.
signed, Ziel hat weniger Stellen als die Quelle	Das Vorzeichen der Quelle wird in das Vorzeichen des Ziels übernommen. Der Absolutwert der Quelle geht verloren, wenn die Zahl zu groß bzw. zu klein für das Ziel ist.

Tabelle 5: Fallunterscheidung bei resize

2.6 Verschieben und Rotieren

Manchmal möchte man einen Vektor um eine bestimmte Anzahl an Stellen nach links oder rechts verschieben. Die Verschiebung nach links entspricht einer Multiplikation mit 2 pro Stelle. Die Verschiebung nach rechts entspricht einer Division durch 2 pro Stelle.

Das kann man in VHDL formulieren, indem man selber für das „Umsetzen“ der Bits sorgt (Listing 2).

```
signal z: std_logic_vector(15 downto 0);

z <= z(13 downto 0) & "00"; -- Verschieben von z um zwei Stellen (z=z*4)
```

Listing 2: Manuelles Verschieben um 2 Stellen nach links

Bei der Interpretation als Zahl muss man aber beim Verschieben nach rechts unterscheiden, ob die Zahl ein Vorzeichen hat oder nicht: Bei vorzeichenlosen Zahlen werden Nullen auf der linken Seite hineingeschoben, bei vorzeichenbehafteten Zahlen wird die oberste Stelle (das ist das Vorzeichen) hineingeschoben (ggf. mehrfach).

Einfacher ist es daher, gleich die vordefinierten Funktionen aus *numeric_std* zu verwenden.

Funktion	Bemerkung
shift_left(), shift_right()	Arbeitet auf *signed* und *unsigned*. Bei *signed* unter Beachtung des Vorzeichens.
rotate_left(), rotate_right()	Auf der einen Seite herausgeschobene Stellen werden auf der anderen Seite hineingeschoben

Tabelle 6: Funktionen zum Schieben und Rotieren

Diese Funktionen arbeiten auf den Datentypen *signed* und *unsigned*. Sie behandeln damit das Vorzeichen automatisch richtig. Daher wird im Beispiel (Listing 3) der *std_logic_vector z* zunächst in eine vorzeichenlose Dualzahl gewandelt und das Ergebnis dann wieder in einen *std_logic_vector* zurückgewandelt. Das kostet zwar Schreibarbeit, aber keinerlei Ressourcen in der Schaltung.

```
use ieee.numeric_std.all; -- hier sind die Funktionen deklariert und definiert

-- z muss vor der Nutzung in signed/unsigned gewandelt werden
-- das Ergebnis wird dann wieder in einen std_logic_vector gewandelt
z <= std_logic_vector(shift_left(unsigned(z), 2));
```

Listing 3: Verschieben um 2 Stellen nach links mit Funktion

Achtung: Die früher vorhandenen Operatoren *srl* etc. gibt es inzwischen im Sprachumfang nicht mehr.

2.7 Bereichsangabe bei integer/natural/positive

Die Angabe des Zahlenbereichs bei der Definition einer *integer* (*natural*, *positive*) dient dem Compiler dazu, die Zahl der mindestens benötigten Stellen in einer Dualzahl zu bestimmen. Der Compiler kann außerdem bei der Verwendung von Konstanten prüfen, ob sie im zulässigen Bereich liegen und der Simulator kann Bereichsüberschreitungen erkennen.

Der Zahlenbereich wird aber in der resultierenden Schaltung **nicht** automatisch eingehalten. Die Definition *signal i: integer range 0 to 70;* führt zur Anlage eines Vektors mit 7 Stellen, weil für diesen Bereich eine Dualzahl mit mindestens 7 Stellen benötigt wird. Eine siebenstellige Dualzahl kann aber die Werte von 0 bis 127 annehmen. Die Anweisung *i <= i+1*; weist in der Schaltung i den Wert 71 zu, wenn i vorher den Wert 70 hatte. Der Überlauf zu 0 erfolgt erst bei i=127. Man muss also den vorzeitigen Überlauf selber erzwingen, indem man mit entsprechenden Anweisungen prüft, ob der (neue) Zahlenwert noch innerhalb des gewählten Bereichs liegt.

3 Beispiel PWM in VHDL

In Listing 4 werden die Konversionsfunktionen eingesetzt, um die einzelnen Bitfelder des Registers PWMCTRL (Abbildung 1) in Zahlenwerte zu wandeln bzw. um die neuen Zahlenwerte wieder in die zugehörigen Felder im Register zu wandeln. Die Funktion (PWM) ist hier nicht enthalten. Man kann natürlich auf die explizite Angabe der Zwischensignale vom Typ *unsigned* auch verzichten. Es wäre auch möglich, gleich mit den *unsigned*-Typen zu rechnen, dann würde man auf die Integertypen verzichten.

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

entity pwm_module is
  port
  (
    -- interface to
    data_out:   out std_logic_vector(31 downto 0);
    data_in:    in std_logic_vector(31 downto 0);

    -- i/o
    pwm_out:    out std_logic
  );
end;

architecture demo of pwm_module is
  signal pwmctrl_max:   unsigned(9 downto 0);
  signal pwmctrl_pwidth:unsigned(9 downto 0);
  signal pwmctrl_cnt:   unsigned(9 downto 0);
  signal pwmctrl_md:    std_logic_vector(1 downto 0);

  signal max:           positive range 1 to 1000;
  signal cnt:           positive range 1 to 1000; -- counter value, assignment not included here
  signal pwidth:        natural  range 0 to 1000;

begin
  -- read data (from µC), separate into fields
  pwmctrl_max    <= unsigned(data_in(29 downto 20));
  pwmctrl_pwidth <= unsigned(data_in(19 downto 10));
  pwmctrl_md     <= std_logic_vector(data_in(31 downto 30));

  -- for convenience: convert to integer and check range
  max <= 1000 when to_integer(pwmctrl_max) > 1000 else
            1 when to_integer(pwmctrl_max) = 0 else
         to_integer(pwmctrl_max);

  pwidth <= 1000 when to_integer(pwmctrl_pwidth) > 1000 else
            to_integer(pwmctrl_pwidth);

  --  write data (to µC) at the correct positions
  data_out(31 downto 30) <= pwmctrl_md;

  -- convert numbers back to fields at the correct position
  data_out(29 downto 20) <= std_logic_vector(to_unsigned(max,10));
  data_out(19 downto 10) <= std_logic_vector(to_unsigned(pwidth,10));
  data_out( 9 downto  0) <= std_logic_vector(to_unsigned(cnt,10));

  -- now the functional description may follow. example: determine value of the pwm output signal
  pwm_out <= '0' when pwmctrl_md = "00" else
             '1' when cnt < pwidth else
             '0';
end;
```

Listing 4: VHDL-Beispiel zu Konversionsfunktionen