Felder und Strukturen in VHDL

1 Einleitung

Sehr häufig hat man es bei Schaltungen der Digitalelektronik im Zusammenhang mit Feldern, d.h. einer Menge von Elementen gleichen Typs zu tun. Ganz offensichtlich ist das bei einem Speicher der Fall, der einfach eine linear nummerierte Menge von Speicherstellen, z.B. Bytes, darstellt. Ein zweites Beispiel wäre ein Registerfeld in einem Peripheriebaustein für ein Mikroprozessorsystem. Auch hier kann der Mikroprozessor oft über eine Adresse auf einzelne Register in bestimmten Modulen (Funktionseinheiten) des Peripheriebausteins zugreifen. Häufig enthält ein Modul dann eine zusammengehörige Menge von Registern. Bei einem PWM-Modul könnte es sich um ein Register für die Frequenz, ein Register für die Pulsbreite und ein Register für die Steuerung (Ein-/Aus, Interrupterzeugung, ...) handeln. Diese Zusammenfassung würde dann beispielsweise in C als Struktur beschrieben. Natürlich lässt sich das kombinieren, so dass ein Peripheriebaustein mit 8 PWM-Kanälen dann einfach ein Feld mit 8 Elementen enthält, wobei jedes Element durch eine Struktur mit den (hier) drei Registern verwaltet wird. Sieht man genauer hin, dann ist ja auch ein Byte (oder ein n-Bit-Register) nichts anderes als ein Feld, das aus 8 bzw. n Bits besteht.

Solche Felder wurden bereits mit den Datentypen *bit_vector* bzw. *std_logic_vector* eingeführt. Tatsächlich sind diese Datentypen nichts anderes als bereits im Standard vordefinierte Felder, für die dann weiterführende Möglichkeiten (Arithmetik) ebenfalls im Standard vordefiniert sind. Das sind aber keine eigenständigen Sprachelemente.

Im Folgenden wird anhand eines Spiels[1] kurz, d.h. nicht vollständig, die Verwendung von Feldern und Strukturen in VHDL beschrieben.

Das Spielfeld ist die 5x7-Matrixanzeige. Alle Positionen werden durch ein Koordinatenpaar (x,y) bestimmt, wobei x von -2 bis 2 und y von 1 bis 7 laufen kann (Abbildung 1 links).

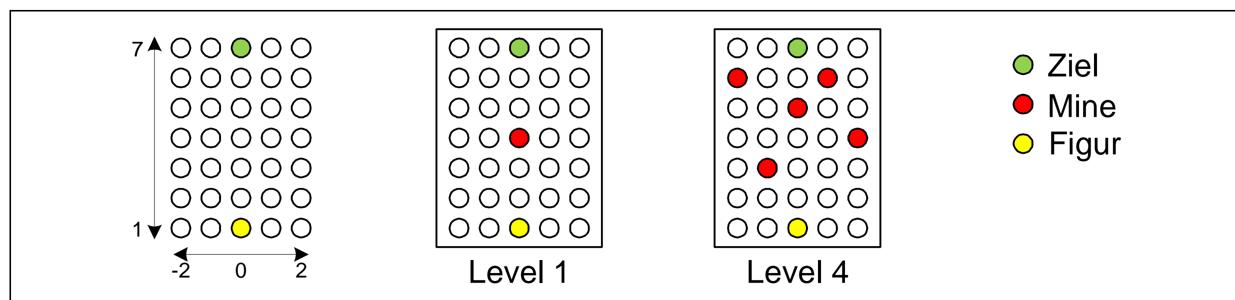

Abbildung 1: Spiel "Minenfeld"

Der Spieler muss mit Hilfe der vier Tasten (auf, ab, links, rechts) die Figur zum Zielpunkt bringen. Zu Beginn eines Spiels startet der Spieler immer auf Position (0,1) und das Ziel liegt fest auf Position (0,7). Das Spiel hat mehrere Level, wobei pro Level eine unterschiedliche Anzahl von Minen auf unterschiedlichen Positionen liegen (Abbildung 1 mitte und rechts). Die Minen sind natürlich nicht sichtbar. Betritt der Spieler eine Mine, dann muss er erneut mit der Startposition beginnen.

[1] "Minenfeld", in VHDL beschrieben von Herrn Rudoy, Student MFM WS13/14

2 Aggregate

Sowohl bei Feldern als auch bei Strukturen hat man es mit Datentypen zu tun, die mehr als ein Element enthalten. In VHDL kann man aus einzelnen Elementen ein Aggregat bilden, das dann in einem Ausdruck als eine Gesamtheit verwendet werden kann. Dabei müssen dann auf beiden Seiten die Elemente in Typ und Länge übereinstimmen. Die Zuordnung geschieht hier über die Stelle.

Beispiel	**Bemerkung**
(Element 1, Element 2, ...)	Allgemeiner Aufbau eines Aggregats aus Elementen
(3,7)	Beide Elemente sind hier eine Zahl, z.B. integer. Im Spiel handelt es sich dann um die Position (3,7).
('0',15,true)	Element 1 vom Typ bit/std_logic, Element 2 eine Zahl, Element 3 vom Typ Boolean
((2,3),(-1,4), others => (0,0))	Jedes Element ist wieder ein Aggregat, hier als Position im Spiel interpretiert. Ab dem dritten Element wird das Aggregat wird auf die benötigte Länge mit den Positionen (0,0) aufgefüllt.

Abbildung 2: Aggregate (positionelle Zuordnung)

Eine Besonderheit stellt die letzte Zeile dar. Die Länge des Aggregats muss hier bekannt sein. Das kann z.B. dadurch sichergestellt sein, dass dieses Aggregat einem Feld mit bekannter Länge zugewiesen wird. Mit dem Schlüsselwort *others* wird hier angegeben, dass bisher noch nicht definierte Elemente mit einem bestimmten Wert - hier (0,0) - belegt werden. Falls dieses Aggregat einem Feld mit einer Länge von 9 Spielpositionen (x, y) zugewiesen wird, dann wird das erste Feldelement auf (2,3) gesetzt, das zweite auf (-1,4) und alle übrigen Feldelemente bekommen die Spielposition (0,0).

3 Records (Strukturen)

Ein Record ist ein Datentyp, der als Elemente unterschiedliche andere Datentypen enthalten kann. Ein Record wird in einem *package* (extern) oder direkt in der Architektur vor der ersten Verwendung deklariert. Die Elemente können durch *neuer_typ.elementname* (wie in C) einzeln herausgegriffen werden. Zudem ist es möglich, die Elemente mit einem Aggregat insgesamt anzusprechen. Im Beispiel wird ein Record deklariert, der eine Spielposition speichern kann (Abbildung 3).

Allgemein

```
type neuer_typ is
record
  Elementname : Elementtyp
  ...         : ....
end record;
```

Beispiel

```
type mp is record
  x: integer range -2 to 2;
  y: integer range 0 to 7;
end record;
```

Abbildung 3: Recorddeklaration mit Beispiel

Der neue Datentyp heißt im Beispiel *mp*. Der Wert für *y* kann hier auch die 0 annehmen. Damit werden später Positionen bezeichnet, die nicht auf dem Spielfeld liegen. In Abbildung 4 zeigt die linke Spalte ein Beispiel für den Zugriff auf ein einzelnes Recordelement. In der rechten Spalte wird mit einem Aggregat auf alle Elemente des Record zugleich zugegriffen.

Einzelzugriff über .

```
architecture rudoy of spiel is
  type position is record
    x: integer range -2 to 2;
    y: integer range 0 to 7;
  end record;

 signal spieler: position;
begin
  if (rising_edge(clk))
```

Gesamtzugriff über Aggregat

```
architecture rudoy of spiel is
  type position is record
    x: integer range -2 to 2;
    y: integer range 0 to 7;
  end record;

 signal spieler: position;
begin
  if (rising_edge(clk))
```

```
  then

    if (aufwaerts)
    then
      if (spieler.y<7)
      then
        spieler.y <= spieler.y+1;
      end if;
    end if;
```

```
  then

    if (start)
    then
      spieler <= (0,1);
    end if;
```

Abbildung 4: Anwendung im Spiel

Die Deklaration des neuen Typs wird hier in der Architekture vorgenommen. Falls der Typ auch in der entity oder ohnehin öfter in andern Modulen verwendet wird, dann nimmt man die Typdefinition in einem *package* vor. Damit ist dann auch die Konsistenz in den einzelnen Modulen sichergestellt.

4 Felder

Während bei einem Record die einzelnen Elemente unterschiedliche Typen haben können und über ihre Namen angesprochen werden, sind die Elemente in einem Feld vom gleichen Typ und werden in der Regel über ihre Position angesprochen. Auch ein Feld ist zunächst nur ein neuer Datentyp. Allerdings erlaubt VHDL bereits bei der Deklaration deutlich mehr an Einstellmöglichkeiten als C.

Man kann insbesondere:

- Die Feldlänge schon vorgeben oder noch offen lassen
- die Laufrichtung des Index (Position) einstellen
- einen beliebigen Wert als erste bzw. letzte Position verwenden.

Einige Beispiele sollen das verdeutlichen:

```
-- Deklaration eines Feldtyps (byte) mit immer gleichem Indexbereich und Richtung
type byte is array (7 downto 0) of bit;

-- Deklaration eines Feldtyps (register)
-- mit später festzulegendem Indexbereich (hier ab 1, da positive)
-- und später festzulegender Richtung
type register is array (positive range <>) of bit;

Nun können Signale oder Variablen als Felder definiert werden:
signal data, command:  byte;                    -- 2 Bytes mit den Namen data und command
signal adc_value:      register (11 downto 1);  -- Ein 12-Bit-Feld, abwärtszählender Index
signal flags:          register(2 to 4);        -- ein 3-Bit-Feld mit aufwärtszählendem Index
```

Abbildung 5: Typische Deklarationen eines Feldes

Für das Spiel könnte man pro Level ein Feld vorsehen, das dann alle auf diesem Level aktiven Minenpositionen enthält. Aufbauend darauf enthält ein zweites Feld für jeden Level ein solches Minenfeld (Abbildung 6).

VHDL-Code	**Bemerkung**
type minenfeld is array(1 to 9) of position;	Feld für 9 Positionen (Records), für einen Level
type minenrom is array(0 to 3) of minenfeld;	Feld für 4 Level (die Elemente sind ebenfalls Felder)

Abbildung 6: Felddeklarationen für das Spiel

5 Initialisierung (Konstanten)

Oft haben Felder einen konstanten Inhalt. Ein Beispiel wäre der Zeichensatz bei einer Punktmatrix-anzeige. Die Muster für jedes Zeichen stehen ja schon fest. Dazu kann man in VHDL beliebige Daten-typen zu Konstanten erklären und sie einmalig initialisieren. Konstanten werden immer mit der Zuweisung := initialisiert. Das geschieht im Kopf der Architecture oder wieder in einem *package*. In Abbildung 7 werden zunächst einzelne Felder für jeden Level deklariert und auch gleich initialisiert. Diese Felder heißen hier *minenfeld1*, ..., *minenfeld4*.

```
-- Initialisierung der Minenfelder (y-position 0 -> keine Mine)
   constant minenfeld1: minenfeld := ((-2,2),              others => (0,0) );
   constant minenfeld2: minenfeld := ((-1,3), (2,2),       others => (0,0) );
   constant minenfeld3: minenfeld := ((-1,4), (0,4),       others => (0,0) );
   constant minenfeld4: minenfeld := ((1,3), (2,3),(-1,5), others => (0,0) );

-- Initialisierung des Gesamtfeldes
   constant minen: minenrom := (minenfeld1, minenfeld2, minenfeld3, minenfeld4);
```

Abbildung 7: Konstanten für das Spiel (Felder)

Danach wird das Feld für alle Level mit dem Namen *minen* deklariert und mit den Einzelfeldern initialisiert. Die Typen der Felder (*minenfeld*, *minenrom*) wurden ja schon zuvor deklariert. Auf der rechten Seite werden jeweils Aggregate verwendet, bei denen die nicht verwendeten Positionen auf (0,0) gesetzt werden. Dies entspricht nicht vorhandenen Minen.

6 Schleifen (hier: for-Schleife)

In dem Spiel muss man nach jedem Schritt des Spielers prüfen, ob die neue Spielerposition einer Minenposition entspricht. Die einzelnen Minenpositionen stehen ja bereits linear angeordnet in einem Feld. Um nun eine gleichartige Operation auf alle (oder einen Teil) der Feldelemente anwenden zu können kennt VHDL Schleifen. Hier wird zunächst das Beispiel gegeben und anschließend folgen allgemeinere Erläuterungen. Der rote Code gehört zur Syntax der for-Schleife.

Nr.	VHDL-Code	Kommentar
1	hit := false;	Die Variable hit dient der Speicherung eines Treffers (Mine / Spieler)
2	mf := minen(level);	Aus dem gesamten Feld wird das Minenfeld mf geholt, das zum Level gehört
3	for i in 1 to 9	Die Laufvariable i zählt von 1 bis 9: das sind die Indizes der Minenpositionen
4	loop	Beginn des Schleifenkörpers
5	mine := mf(i);	Aus dem Minenfeld wird die Position einer Mine ausgewählt: mine
6	if (spieler=mine)	Vergleich der Minenposition (der ganze Record!) mit der Spielerposition
7	then	
8	hit := true;	Falls gleich: Treffer in hit merken
9	exit;	und Schleife vorzeitig verlassen (ein Treffer genügt)
10	end if;	
11	end loop;	Sonst den nächsten Schleifendurchlauf starten

Abbildung 8: Listing zu einer for-Schleife

Diese Schleife sieht fast so aus, wie man sie aus einer sequentiellen Programmiersprache kennt. Eine *for*-Schleife deklariert automatisch eine Laufvariable, hier i. Diese Variable wird bei jedem Durchlauf hochgezählt. Man kann auf die Variable lesen zugreifen, aber nur in der Schleife. Man kann ihr keine Werte in der Schleife zuweisen.

Innerhalb des durch *loop*/*end loop* begrenzten Schleifenkörpers sind nur die Anweisungen erlaubt, die auch in der sequentiellen Umgebung erlaubt sind.

Man kann eine Schleife mit *exit* vorzeitig beenden. Der zu diesem Zeitpunkt gültige Laufindex ist jedoch außerhalb der Schleife nicht verfügbar, daher muss man sich den vorzeitigen Abbruch anders merken. Hier geschieht das über die Variable *hit*.

Die gesamte Schleifenkonstruktion ist eigentlich leicht verständlich. Es gibt aber einen wesentlichen Unterschied zu Schleifen in einer sequentiellen Programmiersprache: Die Schleife wird vom Compiler bei der Schaltungserstellung durchlaufen und für jeden Durchlauf wird Hardware erzeugt!

Konkret heißt das, dass insgesamt 9 Vergleicher (Zeile 6) erzeugt werden, die dann in der Schaltung parallel arbeiten. Jeder Vergleicher vergleicht die aktuelle Spielerposition (x und y) mit je einem Element aus dem Feld. Das *exit* wird durch eine Oder-Funktion aller 9 Vergleichsausgänge ersetzt: Wird ein Vergleich wahr, dann ist der Gesamtausgang *hit* auch wahr. Welcher Vergleicher den Treffer meldet, ist am Ausgang des Oder unbekannt. Damit wird auch klar, warum die Laufvariable i außerhalb der Schleife nicht sichtbar ist: Es gibt sie in der fertigen Schaltung gar nicht mehr! Sie hat nur geholfen, die 9 Teilschaltungen richtig zu verbinden. Damit wird auch das zweite potentielle Problem bei Schleifen in VHDL sichtbar: die Verdrahtungskosten können sehr schnell ansteigen.

Eigenschaft	**VHDL (Hardware)**	**C (Software)**
Schleifendurchlauf wann	Beim Kompilieren	Zur Laufzeit
Aufwand bei n Durchläufen	n mal die Hardware, die für einen Durchlauf benötigt wird	1 mal die Hardware, die für einen Durchlauf benötigt wird
Zeitbedarf zur Laufzeit	1 Takt (in einem process)	n mal die Takte, die für einen Durchlauf benötigt werden

Abbildung 9: Vergleich von Schleifen in Hardware / Software

Eine Schleife, die zur Laufzeit ausgeführt werden soll, erfordert in Hardware (VHDL) einen getakteten Prozess.

7 Attribute zu Arrays

Attribute sind Eigenschaften, die ein Objekt aufweisen kann. Sie werden in VHDL mit einem einfachen Hochkomma an ein Signal oder ein Objekt angehängt. Ein schon bekanntes Attribut zu einem Signal ist *'event*. Es wird bei einer Signaländerung (in der Simulation) wahr.

Arrays möchte man oft über Schleifen bearbeiten, um sich Schreibarbeit zu sparen. Damit man Code schreiben kann, der automatisch die benötigte Schleifengrenzen verwendet, gibt es zu Arrays unter anderem die folgenden Attribute:

Attribut	**Bedeutung**
'low	Untere Feldgrenze (range)
'high	Obere Feldgrenze (range)
'length	Anzahl der Elemente

Abbildung 10: Ausgewählte Attribute

Man kann damit die Zeile 3 aus Abbildung 8 wie folgt neu schreiben: *for i in mf.low to mf.high* .

Damit passt sich die for-Schleife immer der aktuellen Deklaration des Feldes mf an.

8 Anhang: Code des Spiels

```
--Autor: Alexander Rudoy (MFM)
--Datum: 23.11.2013
--Programmname: Minenfeld

--------------------------------------------------------------------------

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

entity Minenfeld is
port
(
  clk: in std_logic;                                        -- Oszillator 50 MHz
  tasteR, tasteL, tasteU, tasteD: in std_logic;             -- Steuertasten
  level2, level3, level4 : in std_logic;                    -- Leveleinstellung
  row1, row2, row3, row4, row5, row6, row7: out std_logic; -- Reihen der 5x7-Anzeige
  col1, col2, col3, col4, col5 : out std_logic;             -- Spalten der 5x7-Anzeige
  LED1, LED2, LED3, LED4, LED5, LED6, LED7, LED8 : out std_logic -- Verbrauchte Leben
);
end;

architecture Spiel of Minenfeld is

  -- Record für eine einzelne Minenposition
  type position is record
        x: integer range -2 to 2;
        y: integer range 0 to 7;
  end record;

  -- Feld für alle Minen eines Levels (hier aktuell maximal 3 Minen)
  type minenfeld is array(1 to 3) of position;
  -- Initialisierung der Minenfelder (y-position 0 -> keine Mine, da 0 keine legale Position)
  constant minenfeld1: minenfeld := ((-2,2), others => (0,0));
  constant minenfeld2: minenfeld := ((-1,3), (2,2), others => (0,0));
  constant minenfeld3: minenfeld := ((-1,4), (0,4), others => (0,0));
  constant minenfeld4: minenfeld := ((1,3), (2,3),(-1,5));

  -- Der gesamte Speicher (pro Level ein Minenfeld)
  type minenrom is array(0 to 3) of minenfeld;
  -- und hier die Initialisierung
  constant minen: minenrom := (minenfeld1, minenfeld2, minenfeld3, minenfeld4);

  signal t1: std_logic;
  signal row: std_logic_vector(1 to 7);
  signal col: std_logic_vector(1 to 5);
  signal LED: std_logic_vector(1 to 8);

  signal level: natural range 0 to 3;

begin

  -- Umrechnung der Schaltereingabe in Level
  level <= 3 when level4='0' else
           2 when level3='0' else
           1 when level2='0' else
           0;


  --Prescaler
  process(clk)
       variable i: integer range 0 to 3000000;
  begin
       if(clk'event and clk='1')
       then
              i := i+1;
              if(i=3000000)
              then i := 0;
                     if(t1='0')
                     then
                            t1 <= '1';
                     elsif(t1='1')
                            then t1 <= '0';
                     end if;
```

```
            end if;
      end if;
end process;


      process(t1)
            variable c: std_logic_vector(1 to 5);
            variable r: std_logic_vector(1 to 7);
            variable l: std_logic_vector(1 to 8);
            variable mine, spieler: position;
            variable mc: integer range 0 to 8;
            variable mf: minenfeld;
            variable trigger: boolean;
      begin

      if(t1'event and t1='1')
      then

            --Steuerung X-Position
            if(tasteR='0' and spieler.x<2)
            then
                  spieler.x:=spieler.x+1; --Schritt nach Rechts
            end if;

            if(tasteL='0' and spieler.x>-2)
            then
                  spieler.x:=spieler.x-1; --Schritt nach Links
            end if;

            case spieler.x is
                  when -2 => c := "10000";
                  when -1 => c := "01000";
                  when 0 => c := "00100";
                  when 1 => c := "00010";
                  when others => c := "00001";
            end case;

            -- Steuerung Y-Position
            if(tasteU='0' and spieler.y<7)
            then
                  spieler.y:=spieler.y+1; --Schritt nach Oben
            end if;

            -- Steuerung X-Position
            if(tasteD='0' and spieler.y>1)
            then
                  spieler.y:=spieler.y-1; --Schritt nach Unten
            end if;

            case spieler.y is
                  when 1 => r := "0000001";
                  when 2 => r := "0000010";
                  when 3 => r := "0000100";
                  when 4 => r := "0001000";
                  when 5 => r := "0010000";
                  when 6 => r := "0100000";
                  when others => r := "1000000";
            end case;


            -- jetzt die aktuelle Position mit dem passenden Minenfeld vergleichen
            trigger := false;
            -- Bestimmen des Minenfelds zum aktuellen Level
             mf := minen(level);
            -- nun über alle Minenpositionen gehen
            for i in minenfeld'low to minenfeld'high loop
              mine := mf(i); -- mine holen
              if (spieler=mine) -- und koordinaten vergleichen
              then
                trigger := true; -- draufgetreten -> Abbruch
                  exit ;
              end if;
            end loop;

            -- Test, ob Mine explodiert
            if (trigger)
            then
             mc:= mc+1;
```

```
            spieler:=(0,1);
              r:="1111111";
              c:="11111";
            else
              -- Test, ob Ziel erreicht
              if (spieler=(0,7))
              then
                   mc:=0; --Reset der verbrauchten Leben
                   spieler:=(0,1);
                   r:="1010101";
                   c:="10101";
              end if;
            end if;

            --Ausgabe der verbrauchten Leben auf den LEDs
            case mc is
                   when 0 => l := "00000000";
                   when 1 => l := "10000000";
                   when 2 => l := "11000000";
                   when 3 => l := "11100000";
                   when 4 => l := "11110000";
                   when 5 => l := "11111000";
                   when 6 => l := "11111100";
                   when 7 => l := "11111110";
                   when others => l := "11111111";
            end case;

            row <= r;
            col <= c;
            LED <= l;

      end if;
    end process;


    (row1, row2, row3, row4, row5, row6, row7) <= row;
    col1<=not(col(1));
    col2<=not(col(2));
    col3<=not(col(3));
    col4<=not(col(4));
    col5<=not(col(5));

    (LED1, LED2, LED3, LED4, LED5, LED6, LED7, LED8) <= not(LED);
end;
```