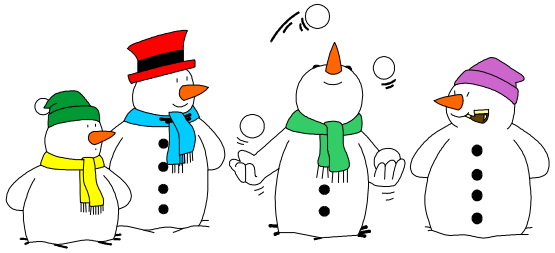
Unterprogramme



Unterprogramme

- Komplexes Verhalten kann modular mit Hilfe von Unterprogrammen beschrieben werden
- Es gibt zwei Arten von Unterprogrammen:
  - Prozeduren
    - werden wegen ihres Effekts ausgeführt
    - Verallgemeinerung eines Befehls
  - Funktionen
    - werden zum Berechnen eines Ergebnisses ausgeführt
    - Verallgemeinerung eines Ausdrucks

Unterprogramme (2)

- Deklaration von Unterprogrammen kann in allen Deklarationsbereichen erfolgen (Architecture, Process, Unterprogramm, etc.)
- Aufruf eines Unterprogramms kann im concurrent und im sequential context erfolgen
- Funktionen können auch in Deklarationsbereichen verwendet werden

Prozeduren

- **procedure** name [(interface_list)] **is**
  { subprogram_declarative_part }
  begin
  { sequential_statement }
  end [name] ;
- Deklarationen können Typen, Subtypen, Konstanten, Variablen und Unterprogramme sein
- Im Gegensatz zu Prozessen verlieren Variablen ihren Wert nach Beenden des Unterprogramms

Beispiel

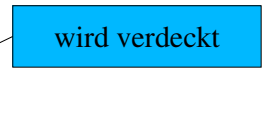
```
architecture rtl of nonsense is
   type stype is array(0 to 15) of integer;
   signal samples : stype;
   signal total   : integer;

   procedure compute_sum is
      variable tmp : integer;
   begin
      tmp := 0;
      for i in samples'range loop
         tmp := tmp + samples(i);
      end loop;
      total <= tmp;
    end compute_sum;
begin ... compute_sum; ...
```


Sichtbarkeit

- Auf externe Objekte kann zugegriffen werden, sofern sie sichtbar sind
- Beispiel:

```
signal s : bit;
...
p : process(s)
   constant val : integer;
   procedure proc is
      variable val : bit;
   begin
      s <= val;
   end proc;
...
```



Return Statements

- Verlassen der Prozedur
-
```
instruction_interpreter : process is
  ...
  procedure read_memory is
  begin
    address_bus <= mem_address_reg;
    mem_read <= '1';
    mem_request <= '1';
    wait until mem_ready='1' or reset = '1';
    mem_data_reg := data_bus_in;
    mem_request <= 0;
    wait until mem_ready = '0';
  end read_memory;
```


Wait Statements

- In Prozeduren dürfen auch wait-Statements vorkommen
  - Unterprogramm wird angehalten
  - aufrufender Prozess wird angehalten (wenn es einen gibt)
- In Funktionen dürfen keine wait-Statements vorkommen

Parameter (1)

```
procedure do_op
    (op          : in func_code;
     arg1, arg2  : in integer;
     result      : out integer) is
begin
   case op is
      when add => result := arg1 + arg2;
      when sub => result := arg1 - arg2;
   end case;
end do_op;
```

- Ähnlich zu Entity/Architecture, aber ...

Parameter (2)

- Parameter können Konstanten (**constant**), Variablen (**variable**) und Signale (**signal**) sein und können die Richtung in, out und inout haben
  - Wird für Parameter mit Richtung "in" nichts angegeben, so sind sie konstant.
    Beim Aufruf kann ein beliebiger Ausdruck (mit passendem Typ) übergeben werden
  - Bei Richtung "out", "inout" wird **variable** angenommen (wenn nichts anderes angegeben ist)

Beispiel

```
procedure negate(a : inout bit) is
begin
   a := not a;
end negate;
...
variable v : bit;
...
negate(v);
```

- Beim Aufruf von negate wird v zunächst nach a kopiert, dann wird a invertiert, und zuletzt wird a nach v geschrieben

Beispiel (2)

```
procedure vec(a : in integer) is
constant c : integer := a;
variable b : bit_vector(0 to a);
begin
   b(a) := '0';
end vec;
...
variable v : integer;
...
v := 7;
vec(v);
```

Signal Parameter

- Für Signal Parameter sind alle drei Richtungen erlaubt
- Signal Parameter stellen eine Referenz auf ein Signal dar
  - Nach einer wait-Anweisung kann ein Signal-Eingang einen anderen Wert haben
  - Signal-Ausgänge werden wie Signale in einem Prozess behandelt

Default Werte

- Default Werte können angegeben werden für Parameter mit
  - Richtung "in"
  - Typ "Constant" oder "Variable"
- ```
  procedure do_op
      (op          : in func_code := add;
       arg1, arg2  : in integer := 0;
       result      : out integer) is
  begin ...
  ```
- Beim Aufruf sollte "open" angegeben werden

Unconstrained Array Parameter

- Parameter können auch unconstrained arrays sein
- Die Größe des Arrays wird durch die Breite des Actuals beim Aufruf bestimmt
- Tatsächliche Grenzen und Größe des Arrays können durch Attribute bestimmt werden

Beispiel

```
procedure contains_one(v     : in  bit_vector;
                       found : out boolean) is
begin
   for i in v'range loop
      if v(i) = '1' then
         found := true;
         return;
      end if;
   end loop;
   found := false;
end contains_one;
...
variable vec : bit_vector(7 downto 0);
variable res : boolean;
...
contains_one(vec, res);
```
```

Funktionen

```
function contains_one_f(v : in bit_vector)
   return boolean
   is
begin
   for i in v'range loop
      if v(i) = '1' then
         return true;
      end if;
   end loop;
   return false;
end contains_one;
...
variable vec : bit_vector(7 downto 0);
variable res : boolean;
...
res := contains_one_f(vec);
```


Funktionen (2)

- Funktionen können "pure" und "impure" sein, Default ist "pure".
- Funktionen, die "pure" sind, dürfen nur Objekte lesen, die
  - lokal deklariert sind oder
  - als Parameter übergeben wurden,
- Funktionen, die "impure" sind, dürfen wie Prozeduren alle sichtbaren Objekte lesen.

Parameterliste

- Parameter einer Funktion müssen die Richtung "in" haben
- variable ist als Parameterklasse nicht erlaubt
- ... ansonsten wie bei Prozeduren

⇒ Parameter können nur gelesen werden

Return Statements

- Eine Return-Anweisung beendet die Funktion und definiert den Rückgabewert
- Es ist ein Fehler, wenn eine Funktion ohne Return-Anweisung beendet wird

Überladen von Unterprogrammen

- Verschiedene Unterprogramme können den gleichen Namen haben (overloading)
- Der Aufruf eines Unterprogrammes ist mehrdeutig (ambiguous) und damit ein Fehler, wenn er sich auf verschiedene Definitionen beziehen kann

Entscheiden der Mehrdeutigkeit

- Es wird berücksichtigt
  - der Name des Unterprogramms,
  - die Zahl der Parameter,
  - die Typen der Parameter,
  - die Namen der formalen Parameter (bei "named association")
  - der Typ des Rückgabewertes (bei Funktionen)

Beispiel

```
procedure inc(a : inout integer;
              n : in integer := 1) is ...
procedure inc(a : inout bit_vector;
              n : in bit_vector := B"1") is...
procedure inc(a : inout bit_vector;
              n : in integer := 1) is ...
variable i  : integer := 2;
variable bv : bit_vector(7 downto 0) := X"03";

inc(i);           -- legal
inc(bv, 2);       -- legal
inc(bv, X"007"); -- legal
inc(bv);          -- illegal, weil mehrdeutig
```


Operatoren überladen

- Alle Operatoren können mit selbst definierten Operator-Funktionen überladen werden
- Eigenschaften von Operatoren:
  - werden als pure function deklariert
  - Name der Operator-Funktion ist "<Symbol>"
  - Parameter entsprechen den Operator-Argumenten

Beispiel

```
function "+"(a, b : in bit_vector)
      return bit_vector is
   variable s : bit_vector(a'range);
   variable c : bit;
begin
   c := '0';
   for i in a'range loop
      s(i) :=  (a(i) xor b(i)) xor c;
      c    := ((a(i) xor b(i)) and c)
            or (a(i) and b(i));
   end loop;
   return s;
end "+";
```


Unterprogramme und Synthese

- Vorsicht bei Verwendung von "unconstrained array" Parametern: Indexbereich und Richtung sind unbekannt
  - Verwendung von Assertions, wenn nur eine Richtung erlaubt sein soll
  - Kopieren in lokale Variablen

```
variable intern : bit_vector(0 to inp'length-1);
```

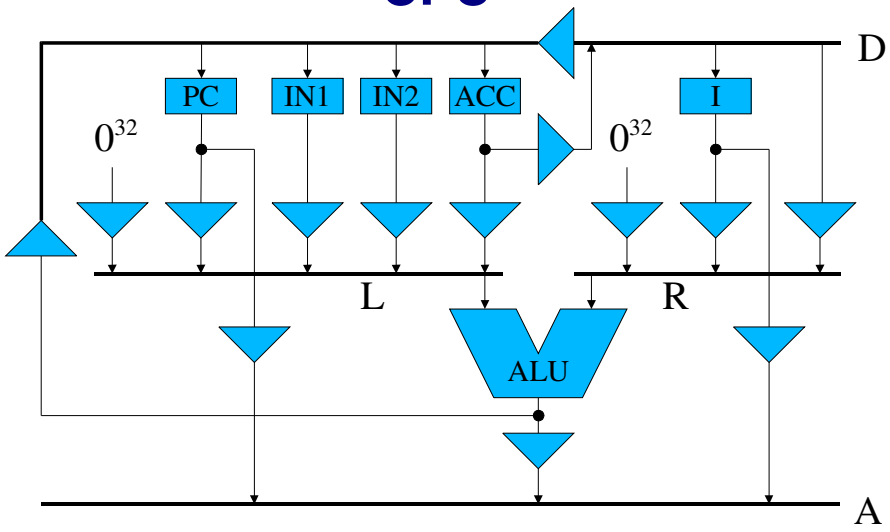

Unterprogramme und Synthese (2)

- Interface-Listen sollten immer vollständig sein (Seiteneffekte!)
- Funktionen sollten nie "impure" sein
- wait-Anweisungen sollten vermieden werden, sofern möglich

Einschub: Resolvierte Signale

- Normalerweise werden Signale nur von einem Prozess geschrieben
- Was passiert, wenn ein Signal mehrere Treiber hat?
  - Dies ist nur erlaubt, wenn eine Resolutionsfunktion angegeben ist
  - Diese Funktion bestimmt dann den Wert des Signals
  - Notwendig zum Beispiel zur Modellierung von Bussen

Datenpfade einer einfachen CPU



Resolutionsfunktionen

- Die Resolutionsfunktion ist eine frei definierte Funktion
  - Sie muss "pure" sein
  - Einziger Parameter ist ein Feld von Werten, für die ein resultierender Signalwert bestimmt werden soll
  - Rückgabewert ist der resultierende Signalwert

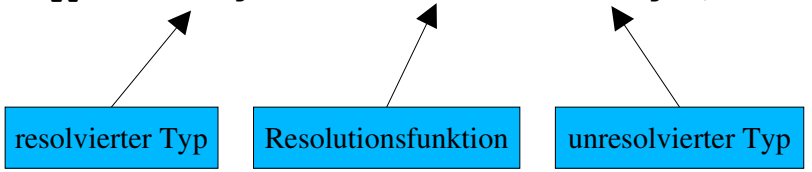
Beispiel: std_logic

```
type std_ulogic is ('U', 'X', '0', '1', 'Z',
                    'W', 'L', 'H', '-');

type std_ulogic_vector is array
        ( natural range <> ) of std_ulogic);

function resolved(s : std_ulogic_vector)
        return std_ulogic;

subtype std_logic is resolved std_ulogic;
```



Resolutionsfunktion

```
function resolved(s : std_ulogic_vector)
                return std_ulogic is
   variable res : std_ulogic := 'Z';
begin
   if (s'length = 1)
   then return s(s'low);
   else
      for i in s'range loop
         res := resolution_table(res, s(i));
      end loop;
   end if;
   return res;
end resolved;
```

Resolutionsfunktion (2)

```
type stdlogic_table is
  array(std_ulogic, std_ulogic) of std_ulogic;

constant resolution_table : stdlogic_table := (
--'U'  'X'  '0'  '1'  'Z'  'W'  'L'  'H'  '-'
 ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'), --U
 ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), --X
 ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'), --0
 ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'), --1
 ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'), --Z
 ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'), --W
 ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'), --L
 ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'), --H
 ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'));---
```


Resolution und Synthese

- Resolutionsfunktionen sollen das Verhalten der Hardware wiederspiegeln, sie werden nicht synthetisiert
- Resolutionsfunktionen sollten immer symmetrisch sein
- Es ist möglich, auch für zusammengesetzte Datentypen Resolutionsfunktionen zu definieren. Dies sollte nicht verwendet werden