Sequential Statements

Sequential Statements

- Werden in Prozessen und Sub-Programmen (Funktionen, Prozeduren) verwendet
- Dienen oft zur Beschreibung von Kontrollstrukturen
- Es gibt Konstrukte ähnlich wie in Programmiersprachen
- Die Beschreibung muss nicht notwendigerweise synthetisierbar sein

If Statements

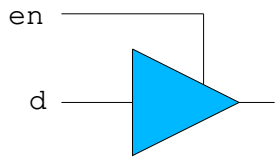
```
if en = '1' then
   temp := data_in;
   stored_value := temp;
end if;
```

```
if mode = immediate then
   operand := immed_operand;
elsif opcode = load or opcode = add or
      opcode = subtract then
   operand := memory_operand;
else
   operand := address_operand;
end if;
```


Tristate-Treiber

```
entity tri is
   port(en, d : in  std_logic;
        y     : out std_logic);
end tri;
```

- Wert des Ausgangs ist d, wenn en = '1', sonst 'Z' (Zustand hoher Impedanz)



Tristate-Treiber (2)

```
architecture rtl of tri is
begin
   p : process(en, d)
   begin
      if en = '1' then
         y <= d;
      else
         y <= 'Z';
      end if;
   end process;
end rtl;
```


Latch

- Speichern von din bei en = '1' ("Transparenz")

```
entity latch is
   port(en, d : in  std_logic;
        q     : out std_logic);
end latch;
```


Latch (2)

```
architecture rtl of latch is
begin
   p : process(en, d)
   begin
      if en = '1' then
         q <= d;
      end if;
   end process;
end rtl;
```


Latch mit Reset

```
entity latch_r is
   port(en, d, reset : in  std_logic;
        q            : out std_logic);
end latch_r;
```

```
...
   p : process(en, d, reset)
   begin
      if reset = '1' then
         q <= '0';
      elsif en = '1' then
         q <= d;
      end if;
   end process;
```

Flipflop

```
entity dff is
   port(clk, d : in  std_logic;
        q      : out std_logic);
end dff;
```

```
architecture rtl of dff is
begin
   p : process(clk)
   begin
      if clk'event and clk = '1' then
         q <= d;
      end if;
   end process;
end rtl;
```

Synthese darf Sensitivitätsliste vervollständigen

Flipflop mit synchronem Reset

```
architecture rtl of dff_sr is
begin
   p : process(clk)
   begin
      if clk'event and clk = '1' then
         if reset = '1' then
            q <= '0';
         else
            q <= d;
         end if;
      end if;
   end process;
end rtl;
```


Flipflop mit asynchronem Reset

```
architecture rtl of dff_ar is
begin
   p : process(clk, reset)
   begin
      if reset = '1' then
         q <= '0';
      elsif clk'event and clk = '1' then
         q <= d;
      end if;
   end process;
end rtl;
```


Synthese getakteter Schaltungen

- Positive (clk'event and clk='1') und negative Flanken (clk'event and clk='0') sollten nicht gemischt werden
- Oft muss eine Anweisung einem bestimmten Schema entsprechen, um richtig erkannt zu werden
  - Beispiel:
    clk'event and clk = '1'
    führt zum Teil zu einer anderen Schaltung als
    clk'event and clk = '1' and clk'last_value = '0'

Case Statements

- `type alu_func is (pass1, pass2, add, sub);`
-
```
case func is
   when pass1 =>
      result := operand1;
   when pass2 =>
      result := operand2;
   when add =>
      result := operand1 + operand2;
   when subtract =>
      result := operand1 - operand2;
end case;
```
  selector expression
- ähnlich wie selected signal assignment im concurrent scope

Case Statements (2)

- Typ der Alternativen muss gleich sein und zum selector expression passen
- Für jeden Wert muss es genau eine Alternative geben
- Alternativen müssen konstant sein
- Zusammenfassen von Alternativen mit
  `when load | add | subtract => ...`
- Alle restlichen Fälle mit
  `when others => ...`

Case Statements (3)

- Angabe eines diskreten Bereichs:

```
variable v, w : std_logic;
...
case v and w is
   when '0' to 'H' =>
      p := '1';
      q := '0';
   when others =>
      p := '0';
      q := '1';
end case;
```


JK-Flipflop

- Funktionstabelle:

J & K	Ausgang
"00"	hält den Wert
"01"	wird 0
"10"	wird 1
"11"	wird invertiert

JK-Flipflop (2)

```
architecture rtl of jkff is
   signal ff : std_logic;
begin
  p : process(clk, j, k)
  variable jk : std_logic_vector(1 downto 0);
  begin
    if clk'event and clk = '1' then
       jk := j & k;
       case jk is when "01" => ff <= '0';
                  when "10" => ff <= '1';
                  when "11" => ff <= not ff;
                  when others => null;
       end case;
end if; end process; q <= ff; end rtl;
```


Null Statements

- In manchen Fällen will man angeben, dass nichts getan werden soll
- Beispiel:

```
case opcode is
   when add => acc := acc + operand;
   when sub => acc := acc - operand;
   when nop => null;
end case;
```

- Frage: warum gibt es kein entsprechendes concurrent statement?

Loop Statements

- [label:]
  loop
  { sequential statement }
  end loop;
- Implementierung einer Endlosschleife

Beispiel: Zähler

```
entity counter is
   port(clk : in bit; count : out natural);
end counter;
```

```
architecture behavior of counter is begin
   incrementer : process
      variable value : natural := 0;
   begin
      count <= value;
      loop
         wait until clk = '1';
         value := (value + 1) mod 16;
         count <= value;
      end loop;
end process; end behavior;
```

While Schleifen

- [label :] **while** condition **loop**
  { sequential_statement; }
  end loop [label] ;
- Der Schleifenkörper wird solange ausgeführt, wie die Bedingung erfüllt ist.

Beispiel: gcd

- Berechnung des größten gemeinsamen Teilers (greatest common divisor)
- ```
  entity gcd is
    port (a, b : in  natural;
          gcd  : out natural);
  end gcd;
  ```


Beispiel: gcd (2)

- ```
  architecture behave of gcd is
  begin
     process(a, b)
     variable x, y, temp : natural;
     begin
        x := a; y := b;
        while (x > 0) loop
           if (x < y) then
              temp := y; y := x; x := temp;
           end if;
           x := x - y;
        end loop;
        gcd <= x;
  end process; end behave;
  ```


For Schleifen

- [label :] **for** identifier **in** discrete_range **loop**
  { sequential_statement; }
  end loop [label] ;
- Der Schleifenparameter nimmt jeden Wert des Bereichs an, beginnend von links
- Beispiel:
  ```
  sum := 0;
  for i in 0 to 127 loop
     sum := sum + i;
  end loop;
  ```

For Schleifen (2)

- Innerhalb der Schleife ist der Schleifenparameter konstant
- Der Schleifenparameter wird nicht deklariert
- Der Schleifenparameter ist nur innerhalb der Schleife definiert.
- ```
  for loop_param in 10 downto 1 loop
     loop_param := 5;    -- falsch
  end loop;
  j := loop_param;       -- falsch
  ```


For Schleifen (3)

- Der Schleifenparameter verdeckt andere Definitionen mit dem gleichen Namen
- Iterationsgrenzen lassen sich mit 'range und 'reverse_range angeben
- ```
     variable a, b : integer;
     variable x : std_logic_vector(2 to 7);
  ...
     a := 10;
     for a in x'range loop
        b := x(a);
     end loop;
     -- a = 10, and b = x(7)
  ...
  ```


Exit Statements

- **exit** [loop_label] [**when** boolean_expression] ;
- Verlassen einer Schleife, wenn Bedingung erfüllt
  - Wenn kein label angegeben, verlassen der direkt umgebenden Schleife
  - Sonst verlassen der Schleife mit diesem label

Exit Statements (2)

- ```
  ...
  outer : loop
     ...
     inner : loop
        ...
        exit outer when cond-1; -- go to B
        ...
        exit when cond-2;       -- go to A
        ...
     end loop inner;
     ...                        -- A
     exit outer when cond-3;    -- go to B
     ...
  end loop outer;
  ...                           -- B
  ```
  ```

Next Statements

- **next** [loop_label] [**when** boolean_expression] ;
- Fortfahren mit der nächsten Iteration (der angegebenen Schleife).
- Beispiel:

```
loop                        loop
   statement1;                 statement1;
   next when cond;             if not cond then
   statement2;                    statement2;
end loop;                      end if;
                            end loop;
```


Schleifen und Synthese

- Endlosschleifen sind nicht synthetisierbar
- While-Schleifen sind nicht synthetisierbar[1]
- Schleifen sollten ein label bekommen
- Schleifenvariablen einer for-Schleife sollten keine externen Objekte überdecken
- Manche Synthese-Tools erwarten, dass Schleifenvariablen vom Typ integer sind

[1] mit zur Zeit gebräuchlichen Synthese-Tools

Schleifen und Synthese (2)

- Grenzen einer For-Schleife müssen in der Regel konstant sein
- Als Grenzen von For-Schleifen sollte, wenn möglich, über Attribute ('range, etc.) angegeben werden.

Assertions

- Wie im concurrent context können auch im sequential context Assertions verwendet werden
- Im Gegensatz zu concurrent assertions brauchen sequentielle Assertions nur im jeweiligen sequentiellen Kontext zu gelten