DAT093
Comments after lab 1

VHDL basics cont.

Basic VHDL structures

Conditional signal assignment

In concurrent code we have two structures for conditional signal assigment.

The first one, the **WHEN** statement, is simular to what we know as an IF statement from software programming

Example

```
SIGNAL a_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
……
  y_signal <= ’1’ WHEN (a_signal=’0’) ELSE
              ’0’;
```

The parantheses are not necessary but increase the readability

The signal must be assigned a value under all conditions which means that the else clause is necessary

VHDL basics cont.

Basic VHDL structures

WHEN statement cont.

The statement could be expanded

Example

```
SIGNAL y_signal:STD_LOGIC;
SIGNAL a_signal:STD_LOGIC_VECTOR(1 DOWNTO 0);
……
  y_signal <= '1' WHEN (a_signal="00") ELSE
              '1' WHEN (a_signal="01") ELSE
              '0';
```

This could also be rewritten as

```
  y_signal <= '1' WHEN ((a_signal="00") OR
                        (a_signal="01")) ELSE
              '0';
```

I think this is less readable though

VHDL basics cont.

Basic VHDL structures

WHEN statement cont.

Observe that there is nothing to say that the selection condition must be of the same type in all clauses

Example

```
SIGNAL a_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
SIGNAL b_signal:STD_LOGIC_VECTOR(1 DOWNTO 0);
……
  y_signal <= '1' WHEN (a_signal='1') ELSE
              '1' WHEN (b_signal="01") ELSE
              '0';
```

This kind of coding is very confusing and should not be used

VHDL basics cont.

```
SIGNAL a_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
……
  y_signal<='1' WHEN (a_signal='0') ELSE
            '0';
```

Basic VHDL structures

WITH statement

The other concurrent conditional signal assignment is the **WITH** statement.
It has simularities with the CASE statement in software programming

Example

We repeat our first WHEN exampl using the WITH statemente

```
SIGNAL x_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
……
  WITH x_signal SELECT
     y_signal <= '1' WHEN '0',
                 '0' WHEN '1';
```

condition signal

This code won´t compile though it is formally correct and we have covered both the high and the low signal values.

Why?

VHDL basics cont.

```
SIGNAL x_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
……
  WITH x_signal SELECT
     y_signal <= '1' WHEN '0',
                 '0' WHEN '1';
```

Basic VHDL structures

WITH statement cont.

In the statement all possible values of the selector, here `x_signal`, has to be covered and the std_logic variable `x_signal` has nine (9) possible values (U, X, 0, 1, Z, W, L, H, -) that must be handled

VHDL basics cont.

Basic VHDL structures

WITH statement cont.

We rewrite the code

```
WITH x_signal SELECT
   y_signal <= ’1’ WHEN ’0’,
               ’0’ WHEN OTHERS;
```

The OTHERS clause covers all cases when `x/=’0’`

Since the synthesized code only has values `0` and `1` (and Z, but not as an input value), this covers all cases.

The code gets somewhat clearer if we rewrite it as

```
WITH x SELECT
   y_signal <= ’1’ WHEN ’0’,
               ’0’ WHEN ’1’,
               ’0’ WHEN OTHERS;
```

The synthisized result will be the same though

VHDL basics cont.

```
SIGNAL y_signal:STD_LOGIC;
SIGNAL a_signal:STD_LOGIC_VECTOR(1 DOWNTO 0)
……
  y_signal <= ’1’ WHEN (a_signal=”00”) ELSE
              ’1’ WHEN (a_signal=”01”) ELSE
              ’0’;
```

Basic VHDL structures

WITH statement cont.

Let´s rewrite the other WHEN statement, the one with the vector

```
WITH a_signal SELECT
   y_signal <= '1' WHEN "00",
               '1' WHEN "01",
               '0' WHEN OTHERS;
```

In the WITH statement we can only have one selecter so we can not have the mixed condition of the scalar `a` and vector `b` that we had in the WITH case.

The two cases that give the same result could be combined

OR statement

```
WITH a_signal SELECT
   y_signal <= '1' WHEN "00" | "01",
               '0' WHEN OTHERS;
```

For enumerated types we can also give ranges using TO or DOWNTO

We´ll see this for the CASE statement later on

VHDL basics cont.

Basic VHDL structures

Conditional signal assignment in sequential code

The WHEN and WITH statements used in concurrent code can not be used in sequential code.

We have a couple of replacements

VHDL basics cont.

Basic VHDL structures

IF statement

The IF statement has simularities to the WHEN statement

The structure is

```
[IF_label:]
IF condition THEN
   sequential code;
[ELSIF condition THEN
   sequential code;]
[ELSE
   sequential code;]
END IF [IF_label];
```

The IF label is optional but it might encance the readability of the code

VHDL basics cont.

Basic VHDL structures

IF statement cont.

Examples

```
IF (a='0') THEN
   y <= '1';
END IF;
```

Since the behavior at all values of a is not declared a memory element must be used

```
IF (a='0') THEN
   y <= '1';
ELSE
   y <= '0';
END IF;
```

IF structure with complete assignment. When all possibilities are fully declared no memory element is needed

Use this instead!

VHDL basics cont.

Basic VHDL structures

Examples cont.

```
compare_ab:
IF ((a='1') AND (b='0')) THEN
   a_high <= '1';
   b_high <= '0';
   equal <= '0';
ELSIF ((a='0') AND (b='1')) THEN
   a_high <= '0';
   b_high <= '1';
   equal <= '0';
ELSIF..
..
ELSE
   a_high <= '0';
   b_high <= '0';
   equal <= '1';
END IF compare_ab;
```

The IF clause is evaluated first

The ELSIF clause is only evaluated if the IF clause is false

The ELSE clause is only evaluated if the IF and the ELSIF clause(s) are false

We have a priority-encoded structure with dominance for the first IF statement

Try using ELSIF instead of separate IF statements

VHDL basics cont.

Basic VHDL structures

CASE statement

The CASE statement has simularities to the WITH statement.

The structure is

```
[CASE_label:]
CASE selectorSignal IS
   WHEN value1 =>
      sequential code;
   WHEN value1 =>
      sequential code;
  [WHEN value2 =>
      sequential code;]
  [WHEN others =>
      sequential code;]
END CASE [CASE_label];
```

The CASE label is optional but it might encance the readability of the code

If the WHEN cases don´t cover all of the possible values for `selectorSignal` we must include the OTHERS clause. It could actually be there even when it is not needed so make it a habit to include it

All cases have the same priority and they may not overlap

VHDL basics cont.

Basic VHDL structures

```
[Case label:]
CASE selectorSignal IS
   WHEN value1 =>
      sequential code;
   WHEN value1 =>
      sequential code;
  [WHEN value2 =>
      sequential code;]
  [WHEN others =>
      sequential code;]
END CASE [Case label];
```

CASE statement cont.

`selectorSignal` is an input port, a signal or a variable.

The `valueX` could be one single value of `selectorSignal`.

It could also be more than one value if we combine them using the OR symbol |

or it could be a range of values if we use `TO` or `DOWNTO`

VHDL basics cont.

Basic VHDL structures

CASE statement cont.

Examples

```
SIGNAL tal_signal:INTEGER RANGE 0 TO 20;
SIGNAL output_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);

   selector:
   CASE tal_signal IS
      WHEN 1 =>
         output_signal <= "0001";
      WHEN 2 =>
         output_signal <= "0010";
      WHEN OTHERS =>
         output_signal <= "0000";
   END CASE selector;
```

selector

selector values

OTHERS clause

VHDL basics cont.

Basic VHDL structures

Examples cont.

```
SIGNAL tal_signal:INTEGER RANGE 0 TO 20;
SIGNAL output_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);

   CASE tal_signal IS
      WHEN 1 =>
         output_signal <= "0001";
      WHEN 2 TO 4 =>
         output_signal <= "0011";
      WHEN 5 | 9 =>
         output_signal <= "0101";
      WHEN OTHERS =>
         output_signal <= "0000";
   END CASE;
```

Single selector value

Range of selector values

Group of selector values

VHDL basics cont.

Basic VHDL structures

LOOP statement

We have one sequential statement that has no concurrent correspondance, the **LOOP** statement.

The statement has a number of forms

- Infinite loop
- WHILE loop
- FOR loop

The FOR loop is the only LOOP statement that is synthesizable and only under some circumstances that we will get back to

VHDL basics cont.

Basic VHDL structures

Infinite LOOP statement NOTICE! This is not synthesizable

As the name suggests this loop goes on forever

The structure is

```
[LOOP_label:]
LOOP
   sequential code;
END LOOP [LOOP_label];
```

The LOOP label is optional but it might encance the readability of the code

Example

```
VARIABLE counter_variable:NATURAL;
……
count_variable:=0;
counter12:LOOP
   WAIT UNTIL rising_edge(clk);
   count_variable:=
        (count_variable+1) MOD 12;
END LOOP counter12;
```

Infinite loop

Triggered every clock cycle on positive clock edge

The counter counts from 0 to 11 on the rising edge of `clk` and then restarts

VHDL basics cont.

Basic VHDL structures

WHILE LOOP statement NOTICE! This is not synthesizable

The WHILE LOOP goes on while some condition is true

The structure is

```
[LOOP_label:]
WHILE condition LOOP
   sequential code;
END LOOP [LOOP_label];
```

The LOOP label is optional but it might encance the readability of the code

Example

```
VARIABLE sum_variable:NATURAL:=0;
……
add_loop:
WHILE (sum_variable < 100) LOOP
   sum_variable:=sum_variable + 3;
END LOOP add_loop;
```

VHDL basics cont.

Basic VHDL structures

FOR LOOP statement

This loop goes on for some range of an identifier

The structure is

Parentheses not allowed

```
[Loop label:]
FOR identifier IN discrete_range LOOP
   sequential code;
END LOOP [Loop label];
```

The LOOP label is optional but it might enhance the readabillity of the code

Example

To be synthesizable these must be constant values

```
one_fill:
FOR index IN 15 DOWNTO 0 LOOP
   vector(index)<='1';
END LOOP one_fill;
```

A way to fill the vector with ones.

Could be replaced by

```
vector <= (OTHERS=>'1');
```

VHDL basics cont.

Basic VHDL structures

LOOP control

We have a couple of functions to control the loop

With the **EXIT** statement we can break out of the loop and leave it.

The basic form is

```
IF condition THEN
   EXIT;
END IF;
```

The code could be shortened to

```
EXIT WHEN condition;
```

VHDL basics cont.

Basic VHDL structures

EXIT statement cont.

Example

```
VARIABLE count_variable:NATURAL;

……
count_variable:=0;
LOOP
   WAIT UNTIL ((clk='1') OR (reset='1'));
   EXIT WHEN (reset='1');
   count_variable:=
            (count_variable+1) MOD 12;
END LOOP;
```

Start value for the count

Triggered by `clk` or `reset`

Leave the loop and start all over again when `reset` is activated

The process continously counts from 0 to 11 on positive edge of the clock signal and is restarted when `reset` is one (1)

VHDL basics cont.

Basic VHDL structures

Creating a clock for simulation

Example

```
clk_proc:
   PROCESS
   BEGIN
      WAIT FOR 50 ns;
      clk_tb_signal<=NOT(clk_tb_signal);

   END PROCESS clk_proc;
```

This will create a clock with a period time of 100 ns

This will not be synthizisable but it can be used in a test bench to create a clock for simulation

Something is missing though. What?

VHDL basics cont.

Basic VHDL structures

Creating a clock for simulation

```
clk_proc:
   PROCESS
   BEGIN
      WAIT FOR 50 ns;
      clk_tb_signal<=NOT(clk_tb_signal);

   END PROCESS clk_proc;
```


Example cont.

We must set a initial value for the clock signal ond we do this in the signal declaration

```
SIGNAL clk_tb_signal:STD_LOGIC:='0';
```

Setting initial values this way won´t transfer to synthis but this clock is just used for simulation so it will work

VHDL basics cont.

Basic VHDL structures

NEXT statement

The **NEXT** statement breaks the current loop round and moves on to the next round

The basic form is

```
IF condition THEN
   NEXT;
END IF;
```

The code could be shortened to

```
NEXT WHEN condition;
```

VHDL basics cont.

Basic VHDL structures

NEXT statement cont.

Example

```
ones_variable:=0;
FOR index IN WIDTH-1 DOWNTO 0 LOOP
      NEXT WHEN vector(index)='0';
      ones_variable:=ones_variable+1;
 END LOOP;
```

Move to the next bit in the vector when the current bit is zero

The code counts the number of ones (1) in `vector`

Also look at the **Accessing a component from a process** document in the **Reading** folder