VHDL
part 2

So far we´ve only looked at pure logic

To day we will look at clocked designs,
conditional coding and shift operations

We will also look at testbenches

Let´s start with the counter from the introductory lab assignment

We will do the implementation in three different way

- Using a `STD_LOGIC_VECTOR` counting value
- Using an `INTEGER` counting value
- Using our ripple carry adder as component

Let´s start with the entity. This will be the same no matter the implementation

We have a counter that should count from zero to twelve and then start allover again.

The counting is controlled by a clock signal

There is also a reset signal

We will start by making the design work for the given counting range and then make it generic

We have the entity

_n indicates that the signal is active low (’0’)

```
ENTITY counter IS
   PORT (clk:IN STD_LOGIC;
         reset_n:IN STD_LOGIC;
         count:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END counter;
```

The size of the vector is given by the counting range


```
ENTITY counter IS
   PORT (clk:IN STD_LOGIC;
         reset_n:IN STD_LOGIC;
         count:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END counter;
```


The architecture

Since the design is triggered by a clock signal we must have a process with the clock signal in the sensitivity list

The `count` port in the entity is of type `OUT` so it can´t be read but we need to read the value to be able to add one (1) to it.

We introduce a `SIGNAL` as the active counting signal

```
ENTITY counter IS
   PORT (clk:IN STD_LOGIC;
         reset_n:IN STD_LOGIC;
         count:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END counter;
```

The architecture cont.

Let´s see what we have so far

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
   END PROCESS count_process;
END arch_counter_std_logic;
```


```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
   END PROCESS count_process;
END arch_counter_std_logic;
```


The architecture cont.

The process we have written will be activated as soon as the clock signal `clk` changes value, that is on both positive and negative clock flank. This is not what we want.

Besides that: the counter value will be stored in flip-flops between the triggerings and flip-flops can only trigger on one flank, not both.

We must make sure that we trigger on only one flank. Let´s take the positive flank

The architecture cont.

Let´s see what we have so far

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clk) THEN
         …
      END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

This is our first conditional code. We´ll soon talk more about it

Negative flank is called FALLING_EDGE

Note that you can only trigger on one flank, never on both


```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clk) THEN
         …
      END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```


The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock) THEN
         …
      END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

There is also an active low reset signal that should control the design.

We can make the reset signal either syncronous or asyncronous

In the synchronous case the counter will be reset syncronous with the clock that is on the positive flank of the clock signal

In the asynchronous case the counter will be reset as soon as the reset signal is activated

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clk) THEN
         …
      END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

Synchronous implementation of the reset signal

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
   IF RISING_EDGE(clk) THEN
      IF (reset_n='0') THEN
         count_signal<=(OTHERS=>'0');
      ELSE
         …
      END IF;
   END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

The process is only triggered by the clock

Keep the two IF clauses separate

Reset is within the RISING_EDGE structure

A simple way to set all bits in the vector to zero (0)

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock) THEN
         …
      END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

Asynchronous implementation of the reset signal

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF ((reset_n='0') THEN
         count_signal<=(OTHERS=>'0')
      ELSIF RISING_EDGE(clk) THEN
         …
      END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

The process is triggered by both the reset and the clock signal

Reset outside of the RISING_EDGE structure

Let´s keep the asyncronous version

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
   END PROCESS count_process;
END arch_counter_std_logic;
```

Now VHDL doesn´t have any rules for addition to `std_logic_vectors`.

To do aritmetic operations the system must know if the value should be treated as a signed or an unsigned value.

We introduce the `std_logic_vector` subtypes `SIGNED` and `UNSIGNED`.

To do this we must include another library

```
USE ieee.numeric_std.ALL;
```

Don´t use the old libraries `std_logic_signed` and `std_logic_unsigned`. They are not part of VHDL anymore

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:std_logic_vector(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
   IF RISING_EDGE(clock) THEN
   END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

Let´s update

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF ((reset_n='0') THEN
         count_signal<=(OTHERS=>'0')
      ELSIF RISING_EDGE(clk) THEN
         count_signal<=count_signal+1;
      END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

Since we only count positive values we use the UNSIGNED type

`count_signal` is of subtype UNSIGNED in the whole process

The architecture cont.

The counting value will have to be passed on to the output port

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF ((reset_n='0') THEN
         count_signal<=(OTHERS=>'0')
      ELSIF RISING_EDGE(clk) THEN
         count_signal<=count_signal+1;
      END IF;
   END PROCESS count_process;
count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

The value must be interpreted as a `std_logic_vector` when passed on to the output

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock) THEN
          count_signal<=count_signal+1;
      END IF;
   END PROCESS count_process;
END arch_counter_std_logic;
```

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock) THEN
          count_signal<=count_signal+1;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

The transfer to the output is done outside of the process. Can we just as well do it within the process?

No!

If we put it in the process then the value will have to be remembered from one clock pulse to the next so we have to store it in flip-flops and it will also take one extra clock cycle for this value to update

If we put it outside of the process then it will only be wires from the `count_signal`, no extra flip-flops

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock) THEN
          count_signal<=count_signal+1;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

We must make sure that the counter goes back to zero (0) when we have counted up to tha maximal value twelve (12).

We need one more condition

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF ((reset_n='0') THEN
         count_signal<=(OTHERS=>'0');
      ELSIF RISING_EDGE(clk) THEN
         IF (count_signal="1100") THEN
            count_signal<=(OTHERS=>'0');
         ELSE
            count_signal<=count_signal+1;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock) THEN
          count_signal<=count_signal+1;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

Don´t mix the clocking condition (RISING_EDGE)

with any of the other conditions.

The architecture cont.

We could have used a count variable instead of the count signal

```
ARCHITECTURE arch_counter OF counter IS
BEGIN
   count_process:
   PROCESS(clk)
      VARIABLE count_variable:UNSIGNED(3 DOWNTO 0);
   BEGIN
      IF RISING_EDGE(clk)
         IF (reset_n='0') THEN
            count_variable:=(OTHERS=>'0');
         ELSIF (count_variable="1100") THEN
            count_variable:=(OTHERS=>'0');
            ELSE
               count_variable:=count_variable+1;
            END IF;
         END IF;
      END IF;
      count<=STD_LOGIC_VECTOR(count_variable);
   END PROCESS count_process;
END arch_counter_std_logic;
```

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock)
         IF (reset_n='0') THEN
            count_signal<=(OTHERS=>'0');
         ELSIF (count_signal="1100") THEN
            count_signal<=(OTHERS=>'0');
            ELSE
               count_signal<=count_signal+1;
            END IF;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

The variable is local to the process

Notice the variable assignment

Since the variable is local to the process its value must be transfered to the output within the process. This will not give any extra flipflops though

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock) THEN
          count_signal<=count_signal+1;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

Now let´s see how we can use an integer as the counting parameter.

If we just declare the parameter as an integer then it will be of size 32 bits which is a waste of hardware so we limit the range.

Since the counter only count from zero and upwards we can also limit the range by using NATURAL numbers.

The architecture cont.

We could have used an integer instead of std_logic as the count parameter.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:NATURAL RANGE 0 to 12;
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clk)
         IF (reset_n='0') THEN
            count_signal<=0;
         ELSIF (count_signal=12) THEN
            count_signal<=0;
            ELSE
               count_signal<=count_signal+1;
            END IF;
         END IF;
      END IF;
   END PROCESS count_process;
count<=STD_LOGIC_VECTOR(TO_UNSIGNED(count_signal,4));
END arch_counter_std_logic;
```

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clock)
         IF (reset_n='0') THEN
            count_signal<=(OTHERS=>'0');
         ELSIF (count_signal="1100") THEN
            count_signal<=(OTHERS=>'0');
            ELSE
               count_signal<=count_signal+1;
            END IF;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

Limited range of the integer subtype NATURAL

First we go from NATURAL to UNSIGNED and then to `std_logic_vector`

We must give the number of bits for the vector

The architecture cont.

For the third option we will use our earlier ripple carry adder as a component to do the addition.

A component is a fixed instantiation and can not be placed in the sequential code of a process.

We should instantiate the component outside of the process and then assign values to it from within the process using signals

The architecture cont.

We start with the component and other stuff outside of the process

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
      COMPONENT ripple_adder IS
      GENERIC(WIDTH:NATURAL:=4);
      PORT(a:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
           b:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
           y:OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0));
   END COMPONENT ripple_adder;
   SIGNAL adder_a_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL adder_y_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   ripple_adder_comp:
   ripple_adder
      GENERIC MAP(WIDTH=>4)
         PORT MAP(a=>adder_a_signal, b=>"0001",y=>adder_y_signal);
```

Here we can use STD_LOGIC_VECTOR since we never do any addition in the code

We always add one

The architecture cont.

And then the process

```
 PROCESS(reset_n,clk)
   BEGIN
      IF (reset_n='0') THEN
         adder_a_signal<="1111";
      ELSIF RISING_EDGE(clk) THEN
         IF (adder_y_signal="1100") THEN
            adder_a_signal<="1111";
         ELSE
            adder_a_signal<=adder_y_signal;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=adder_y_signal;
END arch_counter;
```

Together with ”0001” in b this will give zero

Transfer from counting signal to output

The architecture cont.

Let´s add one last feature to our design.

We introduce an enable signal so we can start and stop the counter.

This port should also be added to the entity.

```
ENTITY counter IS
   PORT (clk:IN STD_LOGIC;
         reset_n:IN STD_LOGIC;
         enable:IN STD_LOGIC;
         count:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END counter;
```

Let´s use the UNSIGNED version of the counter with an asynchronous reset.

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF (reset_n='0') THEN
         count_signal<=(OTHERS=>'0');
      ELSIF RISING_EDGE(clk)
         IF (enable='1') THEN
            IF (count_signal="1100") THEN
               count_signal<=(OTHERS=>'0');
            ELSE
               count_signal<=count_signal+1;
            END IF;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

If written this way the counter can be reset although it isn´t enabled.

How would you write if you only want to be able to reset it when it´s enabled?

The architecture cont.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(reset_n,clk,enable)
   BEGIN
      IF ((reset_n='0') AND (enable='1')) THEN
         count_signal<=(OTHERS=>'0');
      ELSIF RISING_EDGE(clk)
         IF (enable='1') THEN
            IF (count_signal="1100") THEN
               count_signal<=(OTHERS=>'0');
            ELSE
               count_signal<=count_signal+1;
            END IF;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

The asynchronous reset makes it a bit complicated

Generic counter

```
ENTITY counter IS
   PORT (clk:IN STD_LOGIC;
         reset_n:IN STD_LOGIC;
         enable:IN STD_LOGIC;
         count:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END counter;
```

Let´s look at how to make our designs generic

First the entity.

What we have to do is add a GENERIC and make that control the size of te count port

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.math_real.ALL;

ENTITY counter IS
   GENERIC (COUNT_MAX:NATURAL:=12);
   PORT (clk:IN STD_LOGIC;
         reset_n:IN STD_LOGIC;
         count:OUT STD_LOGIC_VECTOR(INTEGER(CEIL(LOG2(REAL(COUNT_MAX))))-1 DOWNTO 0));
END counter;
```

We calculate the size of the port

LOG2 only works on REAL values and is included in the `math_real` library

Round upwards

From REAL to INTEGER

We have three different implementations so we must look att what needs to be done to them to make them generic.

Let´s look at the versions without enable signal to make it simple.

We start with the version with `std_logic_vector` and `UNSIGNED` vector.

We must make the size of the `count_signal` generic.

We do this in the same way as in the entity but since we need to do this more than once we use it to set a constant that we can reuse

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:UNSIGNED(3 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF ((reset_n='0') THEN
         count_signal<=(OTHERS=>'0');
      ELSIF RISING_EDGE(clk) THEN
         IF (count_signal="1100") THEN
            count_signal<=(OTHERS=>'0');
         ELSE
            count_signal<=count_signal+1;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter_std_logic;
```

```
CONSTANT NO_BITS:NATURAL:=INTEGER(CEIL(LOG2(REAL(COUNT_MAX))));
```

Then we can use this to set the size of the count signal and to set the max value for the calculation

```
ARCHITECTURE arch_counter OF counter IS
   CONSTANT NO_BITS:NATURAL:=INTEGER(CEIL(LOG2(REAL(COUNT_MAX))));
   CONSTANT MAX_VECTOR:UNSIGNED(NO_BITS-1 DOWNTO 0):=
                                  TO_UNSIGNED(COUNT_MAX,NO_BITS);
   SIGNAL count_signal:UNSIGNED(NO_BITS-1 DOWNTO 0);
BEGIN
   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF (reset_n='0') THEN
         count_signal<=(OTHERS=>'0');
      ELSIF RISING_EDGE(clk) THEN
         IF (count_signal=MAX_VECTOR) THEN
            count_signal<=(OTHERS=>'0');
         ELSE
            count_signal<=count_signal+1;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(count_signal);
END arch_counter;
```

Interprat UNSIGNED as STD_LOGIC_VECTOR

No for the version with the NATURAL count variable.

Here the size of the count signal is already given by the GENERIC.

The only thing we need to do is set the number of bits for the count port.

```
ARCHITECTURE arch_counter OF counter IS
   SIGNAL count_signal:NATURAL RANGE 0 to 12;
BEGIN
   count_process:
   PROCESS(clk)
   BEGIN
      IF RISING_EDGE(clk)
         IF (reset_n='0') THEN
            count_signal<=0;
         ELSIF (count_signal=12) THEN
            count_signal<=0;
            ELSE
               count_signal<=count_signal+1;
            END IF;
         END IF;
      END IF;
   END PROCESS count_process;
count<=STD_LOGIC_VECTOR(TO_UNSIGNED(count_signal,4));
END arch_counter_std_logic;
```

```
ARCHITECTURE arch_counter OF counter IS
   CONSTANT NO_BITS:NATURAL:=INTEGER(CEIL(LOG2(REAL(COUNT_MAX))));
   SIGNAL count_signal:NATURAL RANGE 0 TO COUNT_MAX;
BEGIN
   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF (reset_n='0') THEN
         count_signal<=0;
      ELSIF RISING_EDGE(clk) THEN
         IF (count_signal=COUNT_MAX) THEN
            count_signal<=0;
         ELSE
            count_signal<=count_signal+1;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(TO_UNSIGNED(count_signal,NO_BITS));
END arch_counter;
```

No for the version with the adder component.

The changes are simular to the earlier ones.

We split the solution into two slides.

```
ARCHITECTURE arch_counter OF counter IS
   CONSTANT NO_BITS:NATURAL:=INTEGER(CEIL(LOG2(REAL(COUNT_MAX))));
   CONSTANT MAX_VECTOR:STD_LOGIC_VECTOR(NO_BITS-1 DOWNTO 0):=

STD_LOGIC_VECTOR(TO_UNSIGNED(COUNT_MAX,NO_BITS));

   SIGNAL count_signal:UNSIGNED(NO_BITS-1 DOWNTO 0);
   COMPONENT ripple_adder IS
      GENERIC(WIDTH:NATURAL:=8);
      PORT(a:IN STD_LOGIC_VECTOR(NO_BITS-1 DOWNTO 0);
           b:IN STD_LOGIC_VECTOR(NO_BITS-1 DOWNTO 0);
           y:OUT STD_LOGIC_VECTOR(NO_BITS-1 DOWNTO 0));
   END COMPONENT ripple_adder;
   SIGNAL adder_a_signal:STD_LOGIC_VECTOR(NO_BITS-1 DOWNTO 0);
   SIGNAL adder_b_signal:STD_LOGIC_VECTOR(NO_BITS-1 DOWNTO 0);
   SIGNAL adder_y_signal:STD_LOGIC_VECTOR(NO_BITS-1 DOWNTO 0);
   CONSTANT B_CONSTANT:STD_LOGIC_VECTOR(NO_BITS-1 DOWNTO 0):=
                              STD_LOGIC_VECTOR(TO_UNSIGNED(1,NO_BITS));
BEGIN
```

This gives a vector with the value 000…1

```
BEGIN
   ripple_adder_comp:
   ripple_adder
         GENERIC MAP(WIDTH=>NO_BITS)
         PORT MAP(a=>adder_a_signal,
                  b=>B_CONSTANT,
                  y=>adder_y_signal);

   count_process:
   PROCESS(reset_n,clk)
   BEGIN
      IF (reset_n='0') THEN
         adder_a_signal<=(OTHERS=>'1');
      ELSIF RISING_EDGE(clk) THEN
         IF (adder_y_signal=MAX_VECTOR) THEN
            adder_a_signal<=(OTHERS=>'1');
         ELSE
            adder_a_signal<=adder_y_signal;
         END IF;
      END IF;
   END PROCESS count_process;
   count<=STD_LOGIC_VECTOR(adder_y_signal);
END arch_counter;
```

The b input is always one

Together with b we get zero

Conditional coding cont.

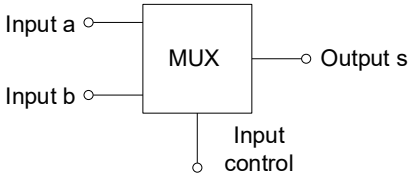
We´ve touched on conditional coding by using IF

Let´s look at it a bit more taking a MUX as an example

We can do this with or without a process.

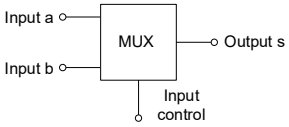
The coding will not be the same though

But the entity is the same




```
ENTITY conditional_MUX IS
   PORT (a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         control:IN STD_LOGIC;
         s:OUT STD_LOGIC);
END conditional_MUX;
```

Conditional coding cont.



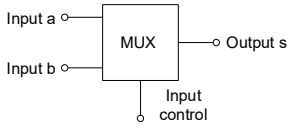
In concurrent code we get

```
ARCHITECTURE arch_conditional_MUX_conc OF
conditional_when IS
 BEGIN
   s<=b WHEN (control='1') ELSE
      a;
END arch_conditional_MUX_conc;
```

Make sure to cover all values for `control`

Remember that there are nine possible values for STD_LOGIC

Conditional coding cont.



In a process we get

```
ARCHITECTURE arch_conditional_MUX_proc OF
conditional_when IS
 BEGIN
 if_proc:
   PROCESS(control)
   BEGIN
      IF (control='1') THEN
         s<=b;
      ELSE
         s<=a;
      END IF;
   END PROCESS if_proc; END
arch_conditional_MUX_proc;
```

The problem is that this won´t work!

Why?

Conditional coding cont.

The process only triggers when `control` changes value and doesn´t care what happens in between. It doesn´t react to changes in a or b

To make it work we must add all signals that influence the output to the sensitivety list

```
ARCHITECTURE arch_conditional_MUX_proc OF
conditional_when IS
 BEGIN
 if_proc:
   PROCESS(control,a,b)
   BEGIN
      IF (control='1') THEN
         s<=b;
      ELSE
         s<=a;
      END IF;
   END PROCESS if_proc; END
arch_conditional_MUX_proc;
```

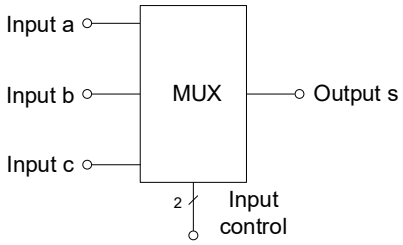
Make sure to cover all values of `control`

```
ARCHITECTURE arch_conditional_MUX_proc OF
                        conditional_when IS
 BEGIN
 if_proc:
   PROCESS(control)
   BEGIN
      IF (control='1') THEN
         s<=b;
      ELSE
         s<=a;
      END IF;
   END PROCESS if_proc;
END arch_conditional_MUX_proc;
```

Conditional coding cont.

Let´s add one input to the MUX.

We need one more input and a two bit control signal.




```
ENTITY conditional_MUX IS
   PORT (a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         c:IN STD_LOGIC;
         control:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         s:OUT STD_LOGIC);
END conditional_MUX;
```

Conditional coding cont.

We get the architecture

```
ENTITY conditional_MUX IS
   PORT (a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         b:IN STD_LOGIC;
         control:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         s:OUT STD_LOGIC);
END conditional_MUX;
```

```
ARCHITECTURE arch_conditional_when_3 OF
                    conditional_when_3 IS
 BEGIN
   s<= a WHEN (control="00") ELSE
       b WHEN (control="01") ELSE
       c;
END arch_conditional_when_3;
```

We have three inputs but the `control` signal has four possibilities so we need do deside what to do with the unused value.

In this code we assign `c` to the output in that case too but there are other possibilities

Conditional coding cont.

We can also use another structure

```
ENTITY conditional_MUX IS
   PORT (a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         b:IN STD_LOGIC;
         control:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         s:OUT STD_LOGIC);
END conditional_MUX;
```

```
ARCHITECTURE arch_conditional_with_select OF
                    conditional_with_select IS
 BEGIN
   WITH control SELECT
      s<=a WHEN "00",
         b WHEN "01",
         c WHEN OTHERS;
END arch_conditional_with_select;
```

It´s simular to a switch structure in software

Conditional coding cont.

Let´s once again use a process

```
ENTITY conditional_MUX IS
   PORT (a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         b:IN STD_LOGIC;
         control:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         s:OUT STD_LOGIC);
END conditional_MUX;
```

```
ARCHITECTURE arch_conditional_if_3 OF
                        conditional_if_3 IS
 BEGIN
   if_proc:
   PROCESS(control,a,b,c)
   BEGIN
      IF (control="00") THEN
         s<=a;
      ELSIF (control="01") THEN
         s<=b;
      ELSE
         s<=c;
      END IF;
   END PROCESS if_proc;
END arch_conditional_if_3;
```

Once again all signals from the assignment must be in the sensitivity list

Conditional coding cont.

```
ENTITY conditional_MUX IS
   PORT (a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         b:IN STD_LOGIC;
         control:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         s:OUT STD_LOGIC);
END conditional_MUX;
```

We have another structure here too

```
ARCHITECTURE arch_conditional_case OF
                  conditional_case IS
 BEGIN
   if_proc:
   PROCESS(control,a,b,c)
   BEGIN
      CASE control IS
         WHEN "00" =>
            s<=a;
         WHEN "01" =>
            s<=b;
         WHEN OTHERS =>
            s<=c;
      END CASE;
   END PROCESS if_proc;
END arch_conditional_case;
```

This is also simular to the software switch structure.

Once again all signals from the assignment must be in the sensitivity list

Shift operations

There are a number of common shift operations

- Logic shift left
- Logic shift right

For bit patterns and unsigned numbers

- Aritmetic shift left
- Aritmetic shift right

For signed numbers

- Rotate

For bit patterns

Shift operations

Logic shifts

In logic shifts we treat the vector that is to be shifted as a number of bits and don´t have to think about any sign of the value.

This also works for unsigned numbers.

A left shift moves the bits a number of steps to the left pushing out the bits that will no longer fit in the vector and fill the new LSB´s with zeros.

Let´s shift the vector `01101101` two steps to the left

`01101101` → `10110100`

`01` are pushed out

`00` are added

Shift operations

Logic shifts cont.

A right shift moves the bits a number of steps to the right pushing out the bits that will no longer fit in the vector and fill the new MSB´s with zeros.

Let´s take the same vector `01101101` and shift it three steps to the right

`01101101` → `00001101`

`101` are pushed out

`000` are added

Shift operations

Aritmetic shifts

In an aritmetic shift we need to take the sign of the number into account, but only when we do a right shift.

We must sign extend the number.

Let´s once again take the number `01101101` and shift it two steps to the right.

The value is positive so extention means inserting zeros

`01101101` → `00011011`

`01` are pushed out `00` are added

A logical and an aritmetic left shift are the same

Shift operations

Aritmetic shifts

Let´s now take another vector `11010110` and shift it two steps to the right.

The value is negative so extention means inserting ones

`11010110` → `11110101`

`10` are pushed out `11` are added

Shift operations

Rotation

In rotation a number of bits are pushed out at one end of the vector and reenters the vector at the other end.
We take the same number as before 01101101 and rotate it two steps to the right.

01101101 → 01011011

01 are pushed out 01 reenters

If we rotate the number three steps to the left we get

01101101 → 01101011

011 are pushed out 011 reenters

Shift operations

Implementation

We can write our own shift and rotate functions.

We assign the value to be shifted to a signal

```
SIGNAL shift_vector: STD_LOGIC_VECTOR(7 DOWNTO 0);
shift_vector<=”01101101”;
```

Three steps shift to the left, aritmetic or logic gives

```
shift_vector<=shift_vector(4 DOWNTO 0) & ”000”;
```

Two steps logic shift to the right gives

```
shift_vector<=”00” & shift_vector(7 DOWNTO 2);
```

Shift operations

Implementation cont.

Two steps aritmetic shift to the right gives

```
shift_vector<=”00” & shift_vector(7 DOWNTO 2);
```

Since the value is positive it´s the same as the logical shift

Lets´s take a negative value

```
shift_vector<=”11011010”;
```

Three steps aritmetic shift to the right gives

```
shift_vector<=”111” & shift_vector(7 DOWNTO 3);
```

Shift operations

Implementation cont.

Now rotation

```
shift_vector<=”11011010”;
```

Three steps rotation to the left gives

```
shift_vector<=shift_vector(4 DOWNTO 0) &
              shift_vector(7 DOWNTO 5);
```

Four steps rotation to the right gives

```
shift_vector<=shift_vector(3 DOWNTO 0) &
              shift_vector(7 DOWNTO 4);
```

Shift operations

Shifts using library functions

We have some ready made shift functions in the `numeric_std` library.

Note that you should not use the functions in the libraries `std_logic_signed` or `std_logic_unsigned`

We have two functions

- `shift_right`
- `shift_left`

Each function has two declarations, one for signed values and one for unsigned values

The two versions are overloaded.

We don´t have to declare which version that is used. The compiler uses the correct version depending on the signal type

Shift operations

Shifts using library functions cont.

If we for example look at `shift_left` we have the function declarations

```
function shift_left ( ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;

function shift_left ( ARG: SIGNED; COUNT: NATURAL) return SIGNED;
```

The vector to shift

The number of steps to shift. Positive for left shift, negative for right shift

Note that we must use the `STD_LOGIC_VECTOR` sub types `SIGNED` and `UNSIGNED`

Shift operations

Shifts using library functions cont.

Now to shift a vector `shift_vector` three steps to the right we write

```
shift_vector<=shift_right(shift_vector,3);
```

If the result will be sign extended or not depends on if the vector `shift_vector` is defined as `SIGNED` or as `UNSIGNED`

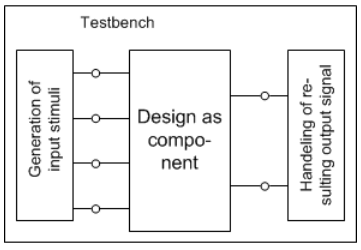
Testbenches

To verify that our design is correct we need to simulate our results.

We will be using the simulator QuestaSim (or ModelSim) from Mentor for this.

To assist in the simulation we can create a kind of test fixture in VHDL, a **testbench**.

This is a top level design where we instantiate our own design as a component and generate input stimuli for the component and watch or check the resulting output signals. We might also check internal signals



Testbenches cont.

We can have three different types of testbenches

- **Type 1** only generates input stimuli and we have to watch the results in the simulator
- **Type 2** generates input stimuli, checks the results and gives an OK signal if the resulting output values are correct
- **Type 3** generates input stimuli and writes a message to the simulator output window if something goes wrong with the simulation results

Testbenches cont.

Example

Let´s take our one bit full adder as an example.

We have the entity

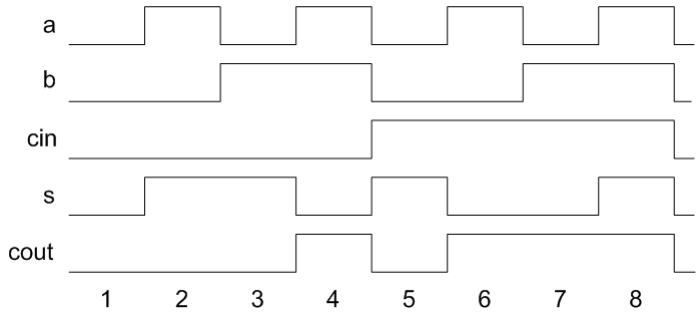
```
ENTITY full_adder IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        cin:IN STD_LOGIC;
        s:OUT STD_LOGIC
        cout:OUT STD_LOGIC);
END full_adder;
```

Testbenches cont.

```
ENTITY full_adder IS
   PORT(    a:IN STD_LOGIC;
            b:IN STD_LOGIC;
          cin:IN STD_LOGIC;
            y:OUT STD_LOGIC
         cout:OUT STD_LOGIC);
END full_adder;
```

Example cont.

We need to generate eight different input stimuli to fully test the circuit



To simplify things we will accept that more than one signal changes it´s value at a given time

Testbenches cont.

Example cont.

Let´s start creating the testbench. We begin with test benchtype 1

In this case we only watch the results from the simulation and we do not need any input or output ports to the testbench.

```
ENTITY full_adder_tb1 IS

END full_adder_tb1;
```

The entity is empty since we have no inputs and no outputs

This causes a problem in QuestaSim since by default signals that don´t connect to outputs are optimized away.

To overcome this we can simulate without optimization. The paper on QuestaSim describes how to do this

Example cont.

```
ENTITY full_adder_tb1 IS
   PORT (   y_tb:OUT STD_LOGIC
          cout_tb:OUT STD_LOGIC);
END full_adder_tb1;
```

In the testbench architecture we instantiate our full adder as a component

```
ARCHITECTURE arch_full_adder_tb1 OF full_adder_tb1 IS
   COMPONENT full_adder IS
      PORT(a:IN STD_LOGIC;
           b:IN STD_LOGIC;
           cin:IN STD_LOGIC;
           s:OUT STD_LOGIC
           cout:OUT STD_LOGIC);
   END COMPONENT full_adder;
   SIGNAL a_signal:STD_LOGIC;
   SIGNAL b_signal:STD_LOGIC;
   SIGNAL cin_signal:STD_LOGIC;
   SIGNAL s_signal:STD_LOGIC;
   SIGNAL cout_signal:STD_LOGIC;
BEGIN
   full_adder_comp:COMPONENT full_adder
      PORT MAP(a=>a_tb,b=>b_tb,cin=>cin_tb,
               s=>s_signal,cout=>cout_signal);
```

Component declaration

Signals to connect to the component

Component Instantiation

Testbenches cont.

Example cont.

We complete the architecture with the input stimuli

```
   a_signal <= '0',
               '1' AFTER 100 ns,
               '0' AFTER 200 ns,
               '1' AFTER 300 ns,
               '0' AFTER 400 ns,
               '1' AFTER 500 ns,
               '0' AFTER 600 ns,
               '1' AFTER 700 ns;
   b_signal <= '0',
               '1' AFTER 200 ns,
               '0' AFTER 400 ns,
               '1' AFTER 600 ns;
   cin_signal <= '0',
                 '1' AFTER 400 ns;
 END arch_full_adder_tb1;
```

This is one of the few times when we can and should use time in our designs

The exact times are not important since we deal with simulation of a design without circuit delays

But we should create all the input signal combinations we want to test for

Testbenches cont.

Example cont.

We need a do file.

Since the instimuli is given in the testbench all the do file need to is to set up signals we like to watch and run the simulation time.

```
-- full_adder_tb1.do

restart -f -nowave
view signals wave
add wave a_signal b_signal cin_signal
add wave s_signal cout_signal
run 730ns
```

Signals to watch

Run the simulation for this time

Testbenches cont.

Example cont.

We move on to test benchtype 2

Here we will need a output signal that signals if something goes wrong with the output signals from the component during simulation.

We add an output to our testbench entity

```
ENTITY full_adder_tb2 IS
   PORT(test_OK:OUT STD_LOGIC);
END full_adder_tb2;
```

Testbenches cont.

```
ENTITY full_adder_tb2 IS
   PORT(test_OK:OUT STD_LOGIC);
END full_adder_tb2;
```

Example cont.

In the architecture we keep the component declaration and do a component instantiation using signals and not outports

```
ARCHITECTURE arch_full_adder_tb2 OF full_adder_tb2 IS
 COMPONENT full_adder IS
      PORT(a:IN STD_LOGIC;
           b:IN STD_LOGIC;
           cin:IN STD_LOGIC;
           s:OUT STD_LOGIC
           cout:OUT STD_LOGIC);
   END COMPONENT full_adder;
   SIGNAL a_signal:STD_LOGIC;
   SIGNAL b_signal:STD_LOGIC;
   SIGNAL cin_signal:STD_LOGIC;
   SIGNAL s_signal:STD_LOGIC;
   SIGNAL cout_signal:STD_LOGIC;
BEGIN
   full_adder_comp:COMPONENT full_adder
      PORT MAP(a=>a_tb,b=>b_tb,cin=>cin_tb,
               s=>s_signal,cout=>cout_signal);
```

Testbenches cont.

Example cont.

We keep the imput stimuli from testbench type 1

```
   a_signal <= '0',
               '1' AFTER 100 ns,
               '0' AFTER 200 ns,
               '1' AFTER 300 ns,
               '0' AFTER 400 ns,
               '1' AFTER 500 ns,
               '0' AFTER 600 ns,
               '1' AFTER 700 ns;
   b_signal <= '0',
               '1' AFTER 200 ns,
               '0' AFTER 400 ns,
               '1' AFTER 600 ns;
   cin_signal <= '0',
                 '1' AFTER 400 ns;
```

Testbenches cont.

Example cont.

We have to complete the code with a test of the output signals

We write the code so that the `test_OK` signal will go low if an error occures and then stay low even if the next stimuli gives a correct result

Testbenches cont.

```
a_signal <= '0',
            '1' AFTER 100 ns,
            '0' AFTER 200 ns,
...
b_signal <= '0',
            '1' AFTER 200 ns,
...
cin_signal <= '0',
              '1' AFTER 400 ns;
```

Example cont.

```
test_proc:PROCESS
BEGIN
   test_OK <= '1';
   WAIT FOR 50 ns; -- 000
   IF ((s_signal/='0') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   WAIT FOR 100 ns; -- 100
   IF ((s_signal/='1') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   WAIT FOR 100 ns; -- 010
   IF ((s_signal/='1') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   ……
END PROCESS test_proc;
END arch_test_bench_type2;
```

Default value for `test_OK`

Wait until the input signals have stabilized

If the result isn´t 00 then set test_OK low

Test for next stimuli

Continue for all eight combinations of input signals

Testbenches cont.

Example cont.

We need a do file here too.

The only difference from the do file for testbench type 1 is that we have added the signal `test_OK` to the signals we watch

```
-- full_adder_tb2.do

restart -f -nowave
view signals wave
add wave a_signal b_signal cin_signal
add wave s_signal cout_signal test_OK
run 730ns
```

Signals to watch

Run simulation

Added signal

The only signal to watch in the test bench is really `test_OK` but it is practical to keep the rest of the signals for debugging

Testbenches cont.

Example cont.

```
test:PROCESS
BEGIN
   test_OK <= '1';
   WAIT FOR 50 ns; -- 000
   IF ((s_signal/='0') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   WAIT FOR 100 ns; -- 100
   IF ((s_signal/='1') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   WAIT FOR 100 ns; -- 010
   IF ((s_signal/='1') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   ……
END PROCESS test;
END arch_test_bench_type2;
```

Observe that as soon as a test sets `test_OK` to zero then it will stay at zero although following tests can be OK

We shouldn´t run this simulation longer than the added WAIT times since the process will restart when it reaches it´s end and then the results will most likely be incorrect

Testbenches cont.

Example cont.

Now over to testbenchtype 3

In this case we don´t need any output signal either since the internal signals are used for our test and since these tests will give the written reports if something is wrong then we have an empty entity

```
ENTITY full_adder_tb3 IS

END full_adder_tb3;
```

We will rewrite the test process, that is replace the test_OK process, but keep the rest of the architecture code

Testbenches cont.

Example cont.

```
a_signal <= '0',
            '1' AFTER 100 ns,
            '0' AFTER 200 ns,
...
b_signal <= '0',
            '1' AFTER 200 ns,
...
cin_signal <= '0',
              '1' AFTER 400 ns;
```

```
test_proc:PROCESS
BEGIN
   WAIT FOR 50 ns; -- 000
   ASSERT ((s_signal='0') AND (cout_signal = '0'))
   REPORT “000 50ns"
   SEVERITY warning;
   WAIT FOR 100 ns; -- 100

   ASSERT ((s_signal='1') AND (cout_signal = '0'))
   REPORT "100 150ns"
   SEVERITY warning;
   WAIT FOR 100 ns; -- 010
   ...
END PROCESS test_proc;
END arch_test_bench_type3;
```

If this condition is true then nothing is wrong with the signals so do nothing

The current simulation time and this text will be written to the simulators Transcript window if the output signals are incorrect

Continue for all eight combinations of input signals

Testbenches cont.

Example cont.

If the **ASSERT** expression is true then the output signals have the correct values

If the expression is false then the test time and the **REPORT** message will be written to the simulators output window

We can have four different levels of **SEVERITY**

- **note**, the message will have the header Note
- **warning**, the message will have the header Warning
- **error**, the message will have the header Error
- **failure**, the message will have the header Failure

The simulation continues

The simulation will stop at current time

The severity levels are given in increasing order

The severity level should be choosen based on the kind of action the error calls for

Example cont.

The assert messages that you write can be simple or very detailed.

You decide!

For a more advanced design the test will be quit extensive and you need to write a lot of code just for the test

Testbenches cont.

Example cont.

Once again we need a do file.

We´re back to the same do file as the one we used for testbench 1 since we have no output.

```
-- full_adder_tb3.do

restart -f -nowave
view signals wave
add wave a_signal b_signal cin_signal
Add wave s_signal cout_signal
run 730ns
```

Signals to watch

Run simulation

Strictly we don´t need to watch any signals since we have the assertions but like in test bench type 2 it is practical to keep the signals for debugging

Test benches

Clocks in test benches

We can create our clocks in the do file like before

```
force clk_tb 0 0,1 50ns -repeat 100ns
```

or create it in the test bench

```
clk_proc:PROCESS
BEGIN
   WAIT FOR 50 ns;
   clk_tb<=NOT(clk_tb);
END PROCESS clk_process;
```

Don´t forget to set a start value for the clock signal

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

In this case this is OK to do this since the signal value will only be used in simulation, not in synthesis