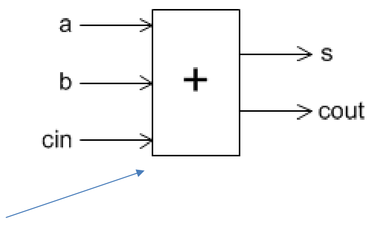
VHDL
part 1

Let´s look at the first assignment from the introductory lab

Full adder



a	b	cin	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

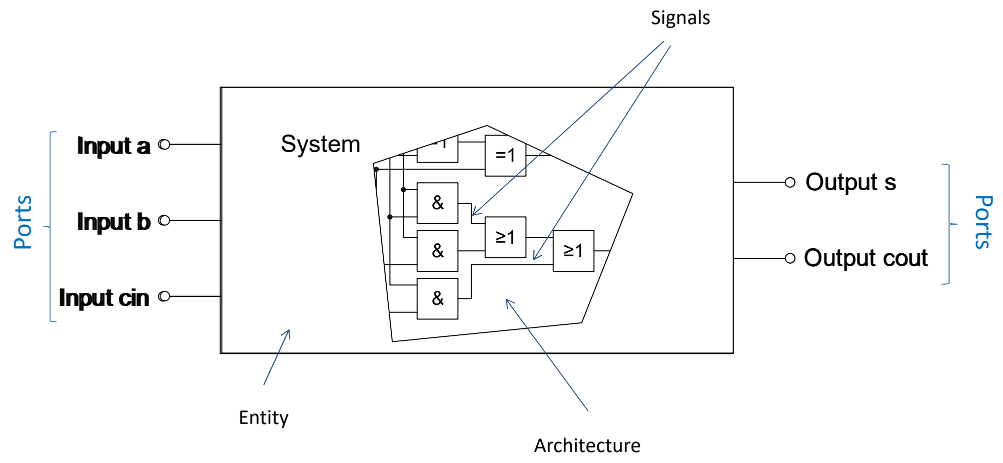
We have the user perspective of the design, the interface

In VHDL we call this an entity

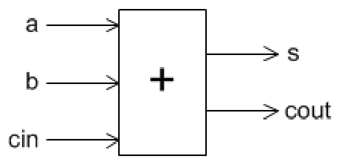
We have the functionallity, the internals

In VHDL we call this an architecture

Full adder cont.



Full adder cont.



Now to VHDL

Let´s look at the entity

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

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;
```

Basic library functions

Name of the entity

Input ports

Output ports

Name of the entity again

The entity

Formal view

Entity

The external, visual part of a VHDL design is the entity that defines the connections (ports) in and out of the design.

The entity can also contain generics, attributes that are used to control the design, for example the width of vectors.

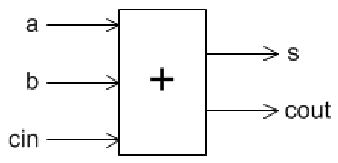
The entity has the following structure

```
ENTITY entity_name IS
  [GENERIC (generic_name:data_type[:=value]);]
  PORT(port_name1:connection_type datatype;
       port_name2:connection_type datatype);
END entity_name;
```

We will get back to this

Observe where the semicolon (;) separators are placed

Full adder cont.



Let´s look at the entity cont.

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

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;
```

Port type

Ports

Formal view

Ports

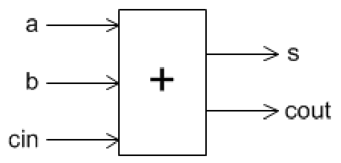
The ports are our connections in and out of the design

We can have four different connection types for the ports

IN	data path directed into the design
OUT	data path directed out of the design
INOUT	bidirectional data path
BUFFER	a readable output

Avoid using this

Full adder cont.



Now to VHDL

Let´s look at the entity cont.

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

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;
```

Signal type

VHDL basics cont.

Data types

Scalar types

Type declarations

```
TYPE ubyte IS RANGE 0 TO 255;
TYPE nibble IS RANGE -8 TO 7;
```

Placed in the architecture before the first `BEGIN`

Signal names

Signal declarations

```
SIGNAL xint_signal:INTEGER;
SIGNAL xubyte1_signal:ubyte;
SIGNAL xubyte2_signal:ubyte;
SIGNAL xnibble_signal:nibble;
```

Predefined type with range $-2^{31} - (2^{31}-1) =$ $= -2,147,483,648 - 2,147,483,647$

Our declared types

What about assignments? The integer type could represent all the values in the `ubyte` range so

```
xint_signal <= xubyte1_signal;
```

would be OK, wouldn´t it?

No! They are two different types and VHDL is strictly typed. To go between types we need conversion functions

```
xubyte1_signal <= xubyte2_signal;
```

is OK though. They are of the same type

VHDL basics cont.

```
TYPE ubyte IS RANGE 0 TO 255;
TYPE nibble IS RANGE -8 TO 7;
```

Scalar types cont.

Why not just use INTEGER as in software?

Our VHDL code will be synthesized to hardware and this hardware must be able to handle all possible values of a signal.

In the hardware our signals are represented by binary bits.

An integer will have to be represented by 32 bits to cover all possible values and that would have to be the width of our signal paths then.

If we only use a fraction of the integer range that would be a waste of hardware.

Even worse if the signal is to be stored along the signal path. In every place where we want to store the signal we would have to include 32 flip-flops to do this.

We can restrict the integer range though. `SIGNAL xint_signal:INTEGER RANGE 10 TO 20;`

The `ubyte` type would take 8 bits and the `nibble` type only 4 bits.

A word of warning. The simulator will give an error if we try to use values outside of the range of the type but the hardware won´t

VHDL basics cont.

Scalar types cont.

More on integers

There are a couple of sub types to integer

SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER’HIGH ← Positive values including zero

SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER’HIGH ← Positive values excluding zero

VHDL basics cont.

Scalar types cont.

Enumeration types

Symbolic names for the values of a signal

```
TYPE weekday IS (sun,mon,tue,wed,thu,fri,sat);
TYPE washing_machine IS (pre_wash,wash,rinse,dry);
```

Typically used to name the states in a state machine,
like the phases, the states, of a traffic light (green, yellow, red)

Some useful predefined enumeration types

```
TYPE boolean IS (false,true);
```

Useful in conditional code

```
TYPE bit IS ('0','1');
```

Logical values. Not recommended
use std_logic

The '-signs indicate that these values
are actually characters

VHDL basics cont.

Scalar types cont.

Enumeration types cont.

Standard logic unsigned

The rest of the line is a comment

```
TYPE std_ulogic IS ('U',  -- uninitialized
                    'X',  -- forcing unknown
                    '0',  -- forcing zero
                    '1',  -- forcing one
                    'Z',  -- high impedance
                    'W',  -- weak unknown
                    'L',  -- weak zero
                    'H',  -- weak one
                    '-'); -- don´t care
```

Always relevant ('0', '1', 'Z')

Only relevant in simulation ('U', 'X', 'W', 'L', 'H')

Only relevant at compilation ('-')

Standard logic (`std_logic`) is a type that is formed from `std_ulogic`

Signed or unsigned has no meaning for single bits

Standard logic is our recommended type for all binary signals

For multi-bit signals it´s expanded to std_logic_vector

Full adder cont.

a	b	cin	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Now to VHDL

Let´s look at the architecture

```
ARCHITECTURE arch_full_adder OF
                         full_adder IS
BEGIN
   s<=(a XOR b) XOR cin;
   cout<=(a AND b) OR
         (a AND cin) OR
         (b AND cin);
END arch_full_adder;
```

Name of architecture

The name could be anything but I use the entity name headed by `arch_` The name can´t start with a digit though

Name of entity

The new line doesn´t mean anything but improves readability

Name of architecture again

Full adder cont.

Now to VHDL

Let´s look at the architecture cont.

```
ARCHITECTURE arch_full_adder OF
                      full_adder IS
BEGIN
   s<=(a XOR b) XOR cin;
   cout<=(a AND b) OR
         (a AND cin) OR
         (b AND cin);
END arch_full_adder;
```

This is a structural description where we use logical blocks (AND, XOR and OR) to describe the functionality.

We can also use a behavior description where we describe what should happen in logical manner.

The behavior description is not suited for this specific design.

Structural and behavior design

Let´s illustrate the two description types by a very simple example: an AND gate.
First the structural design.

```
-- AND gate
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and2 IS
   PORT(a:IN STD_LOGIC;
        b:IN STD LOGIC;
        y:OUT STD_LOGIC);
END and2;

ARCHITECTURE arch_and2 OF and2 IS
BEGIN
   y<=a AND b;
END arch_and2;
```

Inputs

Output

An AND building block

Structural and behavior design cont.

Now the bahavior design.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY and2 IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        y:OUT STD_LOGIC);
END and2;
ARCHITECTURE arch_and2_behavioral OF and2 IS
BEGIN
   y <= '1' WHEN (a='1') AND
                 (b='1') ELSE
        '0';
END arch_and2_behavioral;
```

Behavioral description

```
-- AND gate
LIBRARY ieee;
USE
ieee.std_logic_1164.ALL;
ENTITY and2 IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        y:OUT STD_LOGIC);
END and2;
ARCHITECTURE arch_and2 OF and2 IS
BEGIN
   y <= a AND b;
END arch_and2;
```

Notice that the entity is the same in the two cases

In this case the behavioral desciption is somewhat more complicated but this is no general rule

The behavioral desciption becomes more suited when we get to a bit more complex designs

Full adder cont.

```
ARCHITECTURE arch_full_adder OF
                         full_adder IS
BEGIN
   s<=(a XOR b) XOR cin;
   cout<=(a AND b) OR
         (a AND cin) OR
         (b AND cin);
END arch_full_adder;
```

In the simple cases we´ve seen so far we don´t need any internal signals we can use just the ports.

If we like we can add some internal signals anyway.
Let´s do it to the full adder just for illustration

Full adder cont.

```
ARCHITECTURE arch_full_adder OF
                         full_adder IS
BEGIN
   s<=(a XOR b) XOR cin;
   cout<=(a AND b) OR
         (a AND cin) OR
         (b AND cin);
END arch_full_adder;
```

```
ARCHITECTURE arch_full_adder OF
                         full_adder IS
   SIGNAL a_b1_signal:STD_LOGIC;
   SIGNAL a_b2_signal:STD_LOGIC;
   SIGNAL a_cin_signal:STD_LOGIC;
   SIGNAL b_cin_signal:STD_LOGIC;
BEGIN:
   a_b1_signal<=a XOR b;
   s<=a_b1_signal XOR cin;
   a_b2_signal<=a AND b;
   a_cin_signal<=a AND b;
   b_cin_signal<=b AND cin;
   cout<=a_b2_signal OR
         a_cin_signal OR
         b_cin_signal cin);
END arch_full_adder;
```

The signal names could be anything as long as they don´t start with a digit but I try to use descriptic names that include the type of signal holder (`_signal`)

If we synthezise the two designs the result will be exactly the same

Full adder cont.

A signal is just a connecting wire it doesn´t have any direction like ports have

```
ARCHITECTURE arch_full_adder OF
                         full_adder IS
   SIGNAL a_b1_signal:STD_LOGIC;
   SIGNAL a_b2_signal:STD_LOGIC;
   SIGNAL a_cin_signal:STD_LOGIC;
   SIGNAL b_cin_signal:STD_LOGIC;
BEGIN
   a_b1_signal<=a XOR b;
   s<=a_b1_signal XOR cin;
   a_b2_signal<=a AND b;
   a_cin_signal<=a AND b;
   b_cin_signal<=b AND cin;
   cout<=a_b2_signal OR
         a_cin_signal OR
         b_cin_signal;
END arch_full_adder;
```

In this design all the ports and signals are updated all the time in parallel. The description doesn´t indicate any sequential flow, it is concurrent

Full adder cont.

Process

We can do a design with a sequential flow but then we must introduce the PROCESS.

A process is a part of the architecture where the content is updated sequentially, statement by statement.

The whole process is updated in parallel withe the rest of the code though. The process is a concurrent block.

Full adder cont.

Process cont.

Let´s write a process version of our full adder.

```
ARCHITECTURE arch_full_adder OF
                         full_adder IS
BEGIN
   full_adder_process:
   PROCESS(a,b,cin)
   BEGIN
      s<=(a XOR b) XOR cin;
      cout<=(a AND b) OR
            (a AND cin) OR
            (b AND cin);
   END PROCESS full_adder_process;
END arch_full_adder;
```

Name of the process (not mandatory)

Sensitivity list

Notice

Notice

Name of the process again (not mandatory)

Full adder cont.

The sensitivety list

```
ARCHITECTURE arch_full_adder OF
                         full_adder IS
BEGIN
   full_adder_process:
   PROCESS(a,b,cin)
   BEGIN
      s<=(a XOR b) XOR cin;
      cout<=(a AND b) OR
            (a AND cin) OR
            (b AND cin);
   END PROCESS full_adder_process;
END arch_full_adder;
```

The sensitivity list is a list of the signals that should trigger the process, make it react and do it´s thing, when any of the items in the list change its value.

This can also be done using WAIT statements. Check this on your own. I find this a bit less intuitive.

This process will generate exactly the same hardware as the concurrent version.

There is a slight difference in simulation. We´ll get back to this

Full adder cont.

Sensitivety list mystery

The sensitivity list is a list of the signals that should trigger the process.

```
ARCHITECTURE arch_full_adder OF
                         full_adder IS
BEGIN
   full_adder_process:
   PROCESS(a,b,cin)
   BEGIN
      s<=(a XOR b) XOR cin;
      cout<=(a AND b) OR
            (a AND cin) OR
            (b AND cin);
   END PROCESS full_adder_process;
END arch_full_adder;
```

If we leave out one of the input signals, for example b then the process shouldn´t update when b changes it its value.

This is true in simulation but the synthesized design will not care about the absence of one of the signals from the sensitivity list.

The synthesized result will be the same with or without b in the sensitivity list and it will also be the same with or without the process.

In the version with signals in a process things may not work the way they should as we shall see now in another example.

Concurrent and sequential code

Let´s look some more at the differences between concurrent and sequential code.

We will not use the full adder but a simple design that is tailor made to illustrate this.

VHDL basics cont.

Example

Architecture

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

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

```
ARCHITECTURE arch_and_or OF and_or IS
   SIGNAL x_conc_signal:STD_LOGIC;
   SIGNAL x_seq_signal:STD_LOGIC;

BEGIN
   x_conc_signal <= a AND b;
   y_conc <= x_conc_signal OR c;

   seq_proc:
   PROCESS(a,b,c)
   BEGIN
      x_seq_signal <= a AND b;
      y_seq <= x_seq_signal OR c;
   END PROCESS seq_proc;
END arch_and_or;
```

Internal signals interconnections

Concurrent code

Process name

Sequential code

Sensitivety list

The signals that **trigger** (activate) the process

VHDL basics cont.

Example cont.

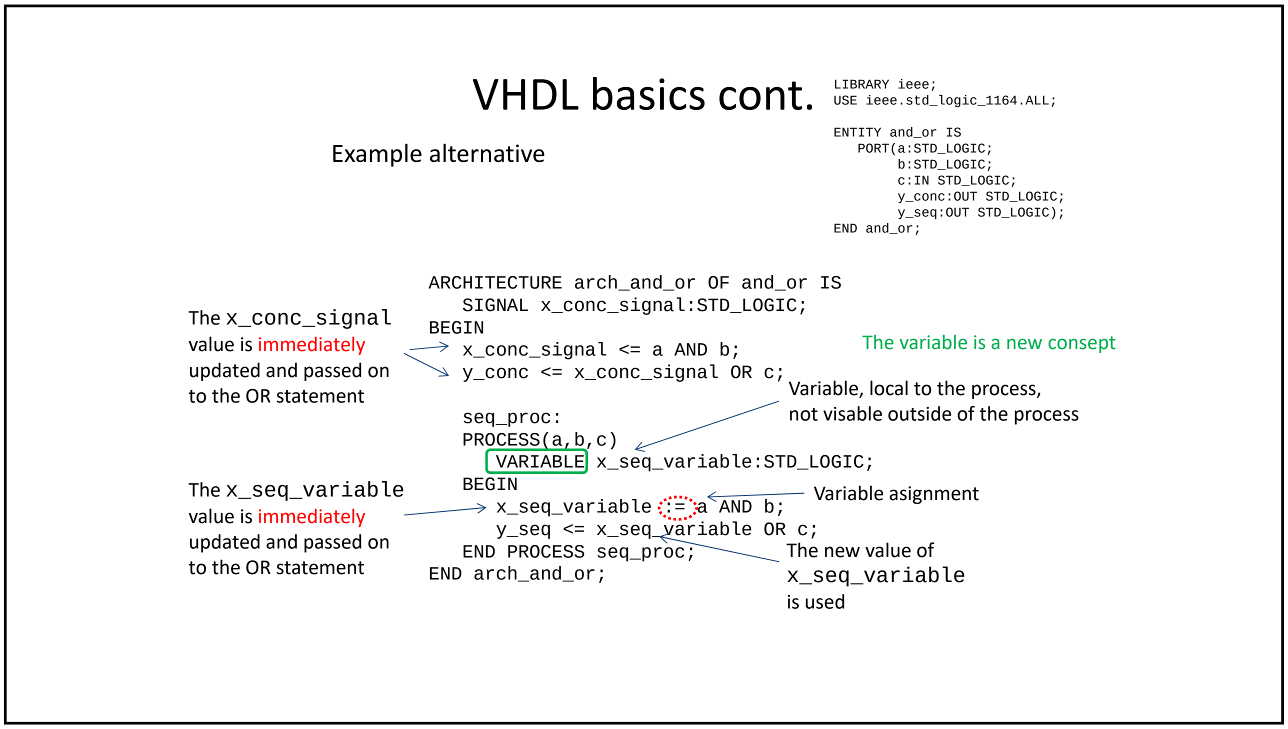
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

```
ARCHITECTURE arch_and_or OF and_or IS
   SIGNAL x_conc_signal:STD_LOGIC;
   SIGNAL x_seq_signal:STD_LOGIC;

BEGIN
   x_conc_signal <= a AND b;
   y_conc <= x_conc_signal OR c;

   seq_proc:
   PROCESS(a,b,c)
   BEGIN
      x_seq_signal <= a AND b;
      y_seq <= x_seq_signal OR c;
   END PROCESS seq_proc;
END arch_and_or;
```

The `x_conc_signal` value is immediately updated and passed on to the OR statement

The `x_seq_signal` value is updated when we leave the process

The value `x_seq_signal` had when we entered the process is used, the assignment on the line above isn´t effective yet



VHDL basics cont.

Example alternative

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

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

```
ARCHITECTURE arch_and_or OF and_or IS
   SIGNAL x_conc_signal:STD_LOGIC;
BEGIN
   x_conc_signal <= a AND b;
   y_conc <= x_conc_signal OR c;

   seq_proc:
   PROCESS(a,b,c)
      VARIABLE x_seq_variable:STD_LOGIC;
   BEGIN
      x_seq_variable := a AND b;
      y_seq <= x_seq_variable OR c;
   END PROCESS seq_proc;
END arch_and_or;
```

The `x_conc_signal` value is immediately updated and passed on to the OR statement

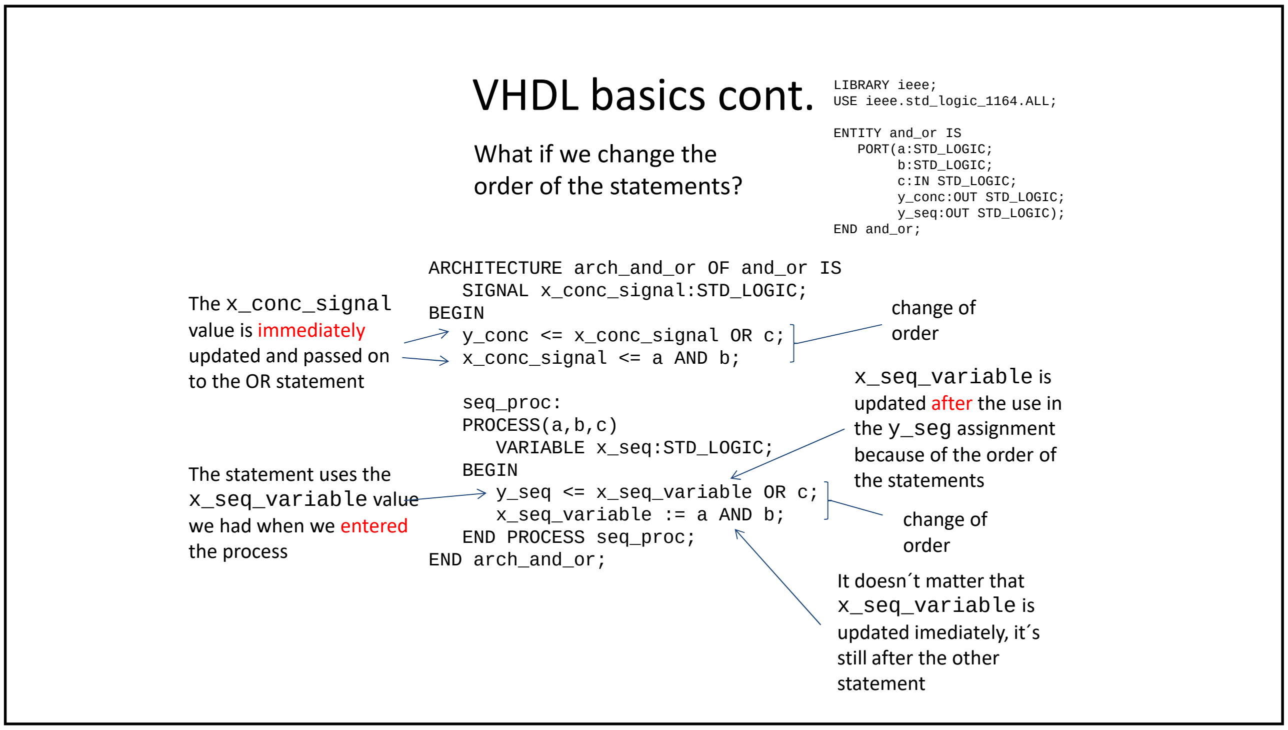
The variable is a new consept

Variable, local to the process, not visable outside of the process

The `x_seq_variable` value is immediately updated and passed on to the OR statement

Variable asignment

The new value of `x_seq_variable` is used



VHDL basics cont.

What if we change the order of the statements?

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

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

```
ARCHITECTURE arch_and_or OF and_or IS
   SIGNAL x_conc_signal:STD_LOGIC;
BEGIN
   y_conc <= x_conc_signal OR c;
   x_conc_signal <= a AND b;

   seq_proc:
   PROCESS(a,b,c)
      VARIABLE x_seq:STD_LOGIC;
   BEGIN
      y_seq <= x_seq_variable OR c;
      x_seq_variable := a AND b;
   END PROCESS seq_proc;
END arch_and_or;
```

The `x_conc_signal` value is immediately updated and passed on to the OR statement

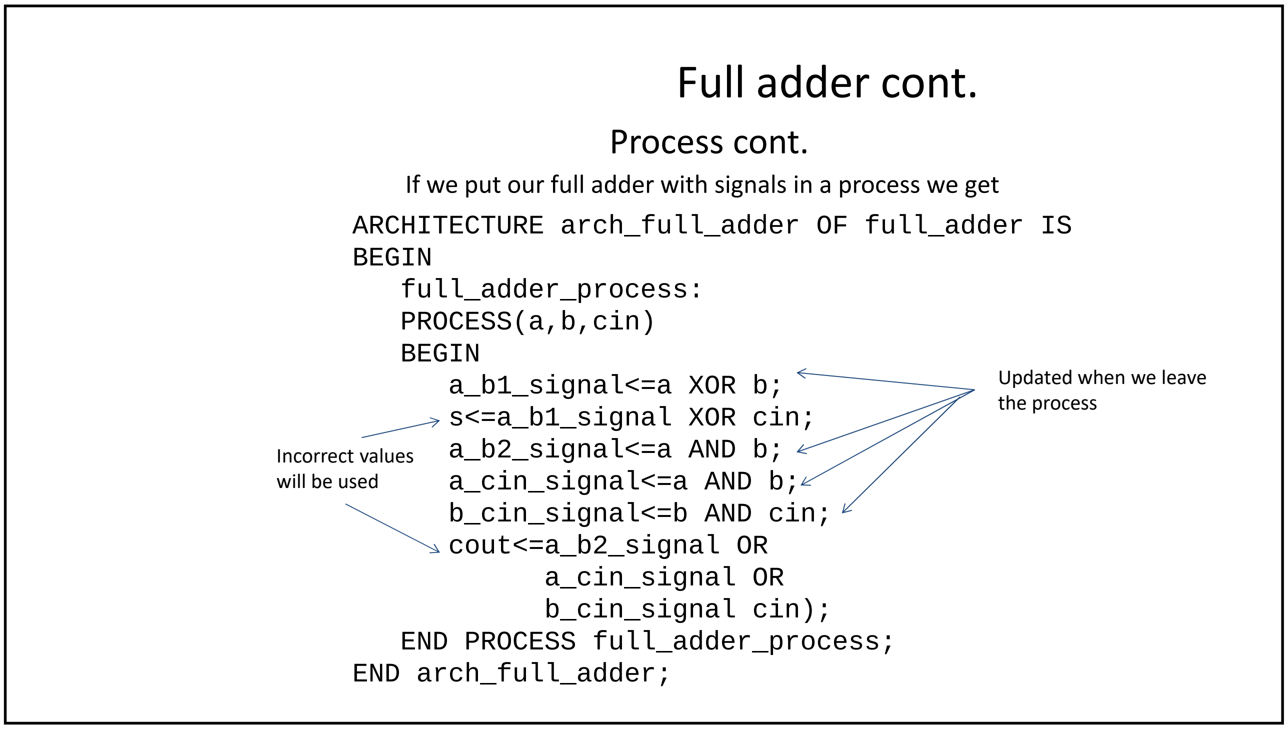
change of order

`x_seq_variable` is updated after the use in the `y_seq` assignment because of the order of the statements

The statement uses the `x_seq_variable` value we had when we entered the process

change of order

It doesn´t matter that `x_seq_variable` is updated imediately, it´s still after the other statement



Full adder cont.

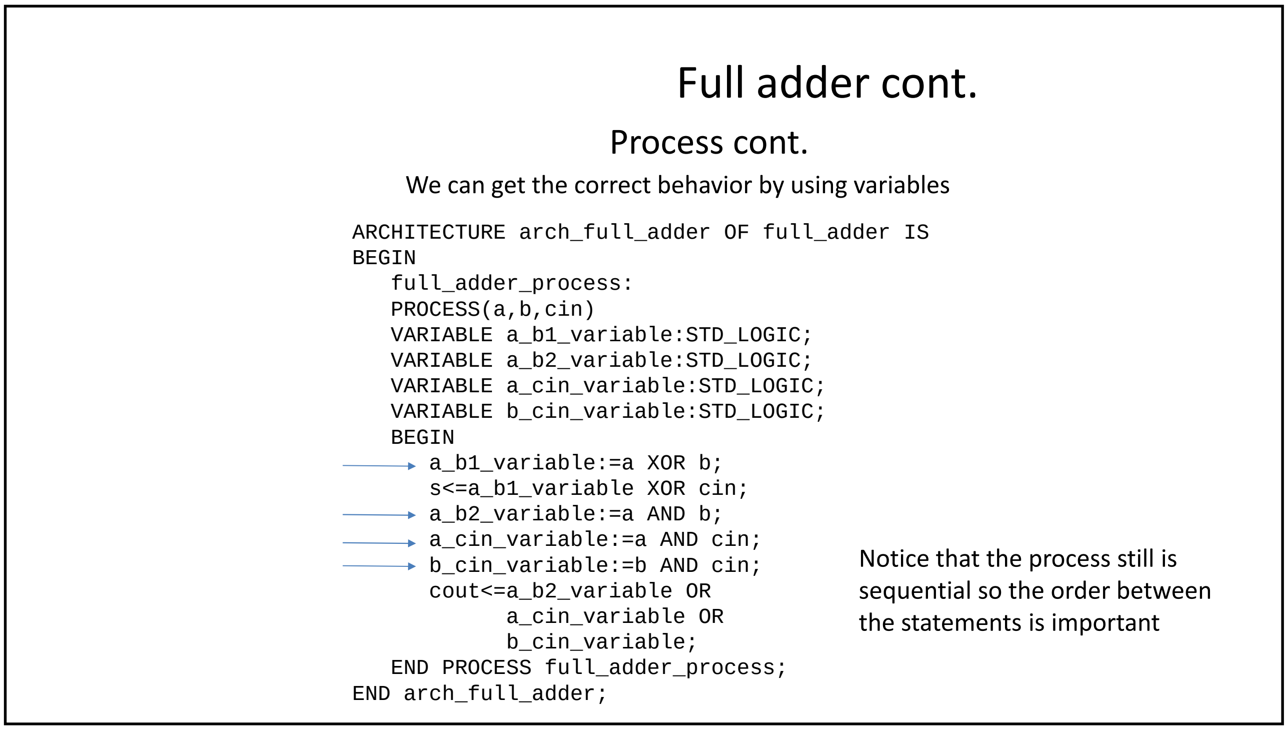
Process cont.

If we put our full adder with signals in a process we get

```
ARCHITECTURE arch_full_adder OF full_adder IS
BEGIN
   full_adder_process:
   PROCESS(a,b,cin)
   BEGIN
      a_b1_signal<=a XOR b;
      s<=a_b1_signal XOR cin;
      a_b2_signal<=a AND b;
      a_cin_signal<=a AND b;
      b_cin_signal<=b AND cin;
      cout<=a_b2_signal OR
            a_cin_signal OR
            b_cin_signal cin);
   END PROCESS full_adder_process;
END arch_full_adder;
```

Updated when we leave the process

Incorrect values will be used



Full adder cont.

Process cont.

We can get the correct behavior by using variables

```
ARCHITECTURE arch_full_adder OF full_adder IS
BEGIN
   full_adder_process:
   PROCESS(a,b,cin)
   VARIABLE a_b1_variable:STD_LOGIC;
   VARIABLE a_b2_variable:STD_LOGIC;
   VARIABLE a_cin_variable:STD_LOGIC;
   VARIABLE b_cin_variable:STD_LOGIC;
   BEGIN
      a_b1_variable:=a XOR b;
      s<=a_b1_variable XOR cin;
      a_b2_variable:=a AND b;
      a_cin_variable:=a AND cin;
      b_cin_variable:=b AND cin;
      cout<=a_b2_variable OR
            a_cin_variable OR
            b_cin_variable;
   END PROCESS full_adder_process;
END arch_full_adder;
```

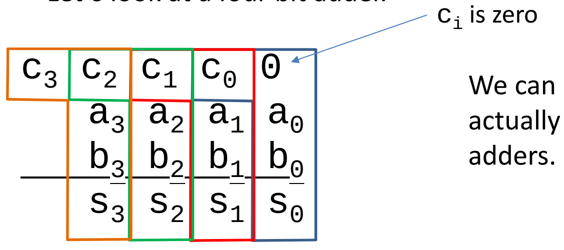
Notice that the process still is sequential so the order between the statements is important

Multi bit adder

A one bit adder is not that useful.

We must use more bits.

Let´s look at a four bit adder.



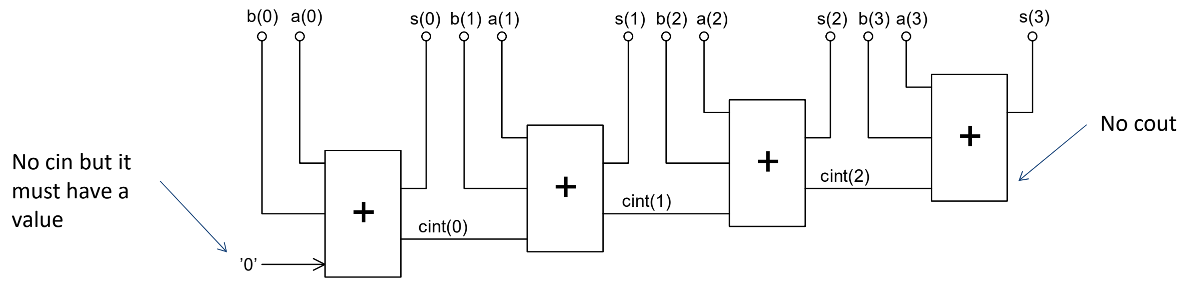
We can see that it´s actually four one bit adders.

Why not reuse our one bit adder?

Multi bit adder cont.

We use the one bit adder as a component in our four bit adder design and instantiate it four times.

Instantiate means to actually create the component so we need to do this four times.



We can of course keep `cin` and `cout` if we like.

Vectors

Before we move on we must look at how to represent signals with more than one bit.

VHDL basics cont.

```
TYPE ubyte IS RANGE 0 TO 255;
```

Composite data types

Arrays, vectors

Since we have to form our multi-value signals from binary values in our hardware implementation, binary vectors are our basic form for signal description besides single binary bits

8 bits

```
TYPE byte IS ARRAY (0 TO 7) OF std_logic;
```

Observe that this is not the same as the earlier type definition of `ubyte`. Both can take on 256 different values but the types are not interchangable.

Indexes can have any range, they don´t have to start with zero (0) or one (1)
Increasing indexes use TO, descending indexes use DOWNTO

We can address individual bits and vector ranges in the array using indexes

```
SIGNAL xbit_signal:std_logic;
SIGNAL xbyte_signal:byte;
SIGNAL xnibble_signal:std_logic(0 TO 3);
……
   xbit_signal <= xbyte_signal(3);
   xnibble_signal <= xbyte_signal(2 TO 5);
```

Single bit

Subarray

VHDL basics cont.

Arrays, vectors cont

In many cases our vector represents a binary value,
In these cases it is more natural to use descending indexes

```
TYPE byte IS ARRAY (7 DOWNTO 0) OF std_logic;
```

This type definition is in most cases not necessary since we have predefined vector types for bits and std_logic

VHDL basics cont.

Arrays, vectors cont

Predefined types

```
SIGNAL bit_word_signal:bit_vector(15 DOWNTO 0);
SIGNAL std_byte_signal:std_logic_vector(7 DOWNTO 0);
SIGNAL std_signal:std_logic_vector(1 TO 12);
```

For these pre-defined types the indexes must be natural numbers, that is positive or zero (0)

When we write a value to a std_logic_vector we treat the value as a string

```
std_logic_signal <= ”00110110”;
```

Double quotations indicate string

It is recommended to **avoid using bit_vector** and **always use std_logic_vectors**

Use descending indexes if there is no good reason to do otherwise

VHDL basics cont.

```
SIGNAL stdbyte_signal:std_logic_vector(7 DOWNTO 0);
```

Arrays, vectors cont

We can also read or write parts of a vector

```
 std_logic_signal(4 DOWNTO 2) <= ”110”;

four_bit_signal <=  std_logic_signal(4 DOWNTO 1);
```

or use concatination (&) to manipulate our vectors

A new line within the code line is fully acceptable

```
std_logice_signal <=
      std_logic_signal(4 DOWNTO 2) & ”00110”;
```

The number of bits on the two sides of the assingment sign must of course match

VHDL basics cont.

Multidimensional arrays

An array can have more than one dimension

```
TYPE multiarray IS ARRAY (0 TO 9,0 TO 4) OF STD_LOGIC;
```

We address the individual elements using two indexes

```
SIGNAL ma_signal:multiarray;
……
  ma_signal(5,3) <= ’1’;
```

VHDL basics cont.

Arrays of arrays

In some cases it is more practical to be able to address the rows of the multi dimensional array and not the individual elements.
This could be the case when we create a memory for byte sized data.
In these cases it is better to define a array of vectors

```
TYPE memory IS ARRAY (0 TO 9) OF
                 STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Here we address the rows of the array, that is the bytes and not the individual bits

```
SIGNAL mem_signal:memory;
……
  mem_signal(5) <= ”00110110”;
```

In this case we have no simple way of addressing the individual elements.

To do this we have to first read the row vector, address the individual bit in the row and then write the row vector back to its place

Multi bit adder cont.

```
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;
```

Back to the four bit adder

First the entity

```
ENTITY four_bit_ripple_adder IS
   PORT(a:IN STD_LOGIC_VECTOR(3 DOWNTO 0); ←
        b:IN STD_LOGIC_VECTOR(3 DOWNTO 0); ←
        y:OUT STD_LOGIC_VECTOR(3 DOWNTO 0)); ←
END four_bit_ripple_adder;
```

The entity is quite simular to the entity for our one bit adder but the ports have changed to vectors

Multi bit adder cont.

```
ENTITY four_bit_ripple_adder IS
   PORT(a:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
        cout:OUT STD_LOGIC);
END four_bit_ripple_adder;
```

Then the architecture.

We start by declaring the component we will use, the full adder

```
ARCHITECTURE arch_four_bit_ripple_adder OF
                              four_bit_ripple_adder 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;
```

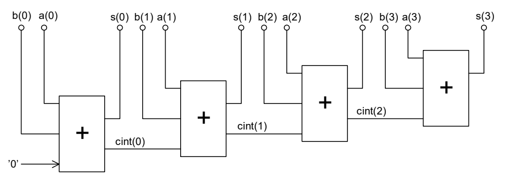
Multi bit adder cont.

```
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;
```

The component declaration looks exactly as the entity of the component with two differences.

```
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;
```

Multi bit adder cont.

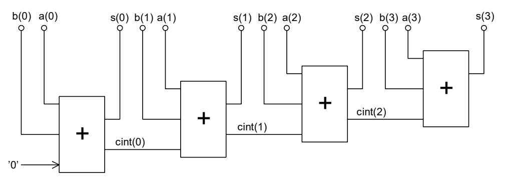


Then we need signals för our rippling carry signals

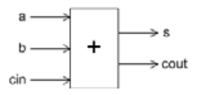
```
ARCHITECTURE arch_four_bit_ripple_adder OF
                              four_bit_ripple_adder IS
   COMPONENT full_adder IS
      PORT(a:IN STD_LOGIC;
           b:IN STD_LOGIC;
           cin:IN STD_LOGIC;
           s:OUT STD_LOGIC;
           cout:OUT STD_LOGIC);
   END COMPONENT full_adder;
   SIGNAL cint_signal:STD_LOGIC_VECTOR(2 DOWNTO 0);
```

We don´t need a vector here, we can use single bit signals but the vector will simplify things later on

Multi bit adder cont.



```
ENTITY four_bit_ripple_adder IS
   PORT(a:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END four_bit_ripple_adder;
```



Next we instantiate the actual components. First bit 0.

Notice

```
full_adder_comp_0:
   COMPONENT full_adder
      PORT MAP(a=>a(0),
               b=>b(0),
               cin=>'0',
               s=>y(0),
               cout=>cint_signal(0));
```

The instantiated component must have a unique name

Name of the declared component

Signal name on top level

Port name within the component

Zero, no carry in

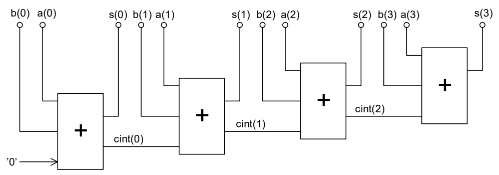
Multi bit adder cont.

There are two ways to assign signals to the ports of the component

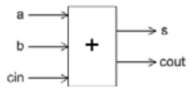
```
full_adder_comp_0:                       full_adder_comp_0:
   COMPONENT full_adder                     COMPONENT full_adder
      PORT MAP(a=>a(0),                        PORT MAP(a(0),
               b=>b(0),                                 b(0),
               cin=>'0',                                '0',
               s=>y(0),                                 y(0),
               cout=>cint_signal(0));                 cint_signal(0));
```

Use the left one, it´s much more readable

Multi bit adder cont.



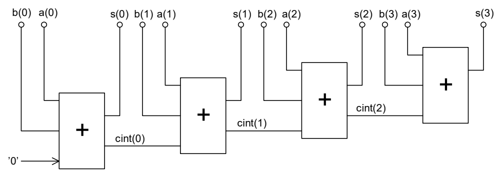
Bit one is very simular.



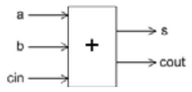
```
full_adder_comp_1:
   COMPONENT full_adder
      PORT MAP(a=>a(1),
               b=>b(1),
               cin=>cint_signal(0),
               s=>y(1),
               cout=>cint_signal(1));
```

Carry from bit zero

Multi bit adder cont.



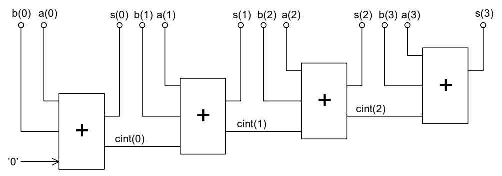
And bit two



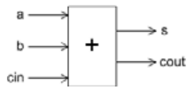
```
full_adder_comp_2:
   COMPONENT full_adder
      PORT MAP(a=>a(2),
               b=>b(2),
               cin=>cint_signal(1),
               s=>y(2),
               cout=>cint_signal(2));
```

Carry from bit one

Multi bit adder cont.



Finally bit three

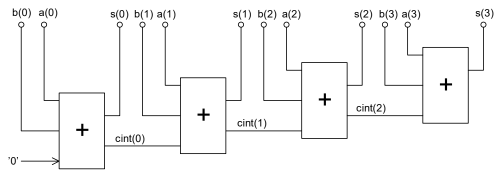


```
full_adder_comp_3:
   COMPONENT full_adder
      PORT MAP(a=>a(3),
               b=>b(3),
               cin=>cint_signal(2),
               s=>y(2));
```

Carry from bit two

Since we don´t use `cout` from bit three we can just leave it out

Multi bit adder cont.

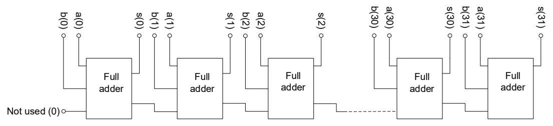


The use of components means that we can reuse code but the shown method will get quit tiresome if we have many bits.

Let´s look at a 32 bit adder

Multi bit adder cont.

32 bit adder



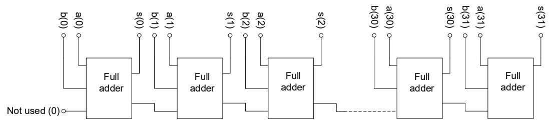
The entity is the same as for the four bit adder exept for the number of bits

The name can´t start with a digit

```
ENTITY bit_32_ripple_adder IS
   PORT(a:IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END bit_32_ripple_adder ;
```

Multi bit adder cont.

32 bit adder cont.



If we look at the 32 bit adder we can see that the 30 adders in the middle have identical behavior while the first and the last adder have other sets of signals.

The declaration of the one bit adder component is the same as before.

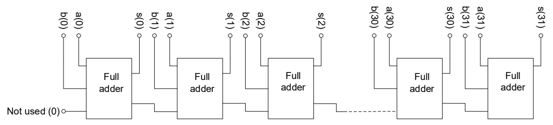
The signal for the ripple values are the same but with another number of values.

```
SIGNAL cint_signal:STD_LOGIC_VECTOR(30 DOWNTO 0);
```

We instantiate the LSB and MSB adders one their own since they are different.

Multi bit adder cont.

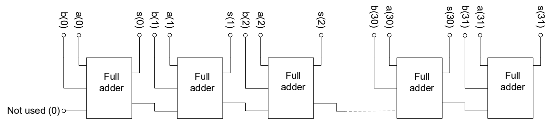
32 bit adder cont.



```
full_adder_comp0:COMPONENT full_adder
         PORT MAP(a=>a(0),b=>b(0),cin=>'0',
                  s=>s(0),cout=>cint_signal(0));

full_adder_comp31:COMPONENT full_adder
      PORT MAP(a=>a(31),b=>b(31),
               cin=>cint_signal(30),s=>s(31));
```

Multi bit adder cont.

32 bit adder cont.



Name of the GENERATE statement

We use a LOOP GENERATE statment for the 30 one bit adders in between.

```
G:FOR i IN 1 TO 30 GENERATE
      full_adder_comp_i:COMPONENT full_adder
         PORT MAP(a=>a(i),b=>b(i), cin=>cint_signal(i-1),
                  s=>s(i),cout=>cint_signal(i));
   END GENERATE;
```

Component name, the same name for all 30 components

Note that this is not the same as a loop in software. The declaration will not loop but actually instantiate 30 one bit adders in hardware

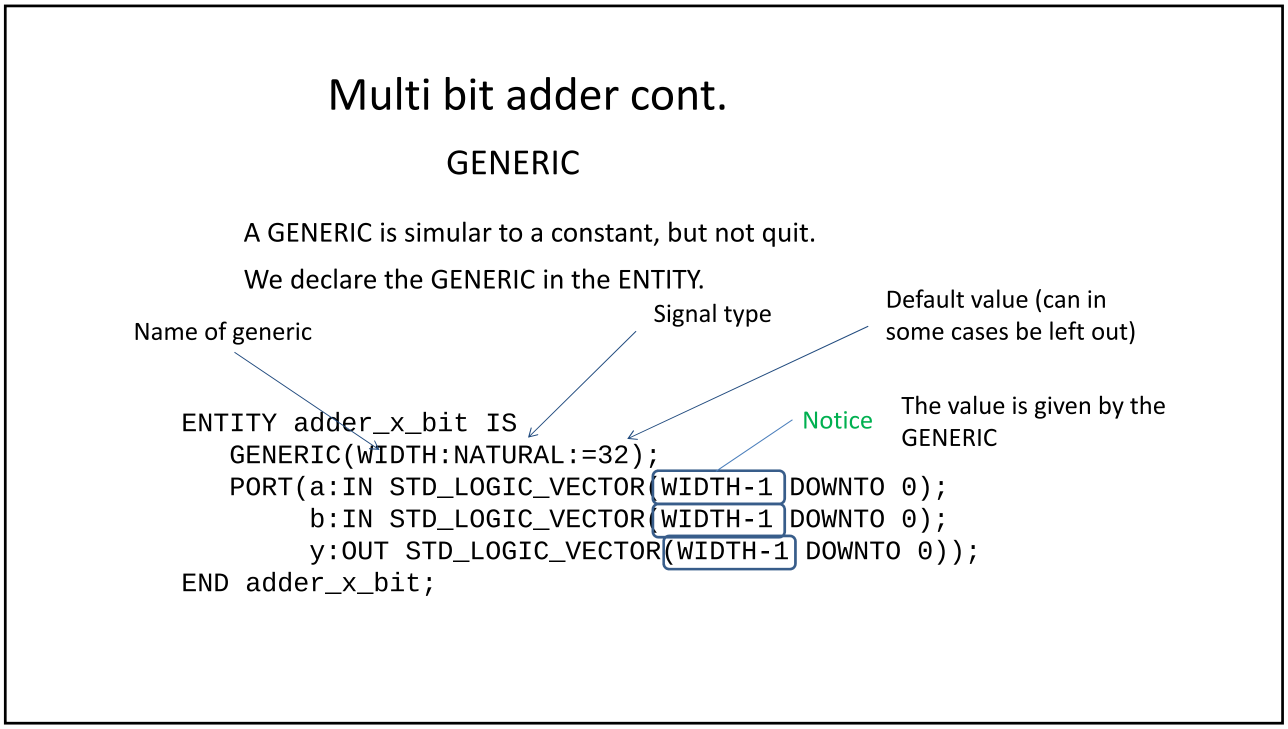
Multi bit adder cont.

N bit adder

Now what if we want to use our multi-bit adder as a component in another top level design.

Is there a simple way to implement an adder with the required number of bits for that specific design?

Yes there is! We can use a GENERIC.



Multi bit adder cont.

GENERIC

A GENERIC is simular to a constant, but not quit.

We declare the GENERIC in the ENTITY.

Name of generic

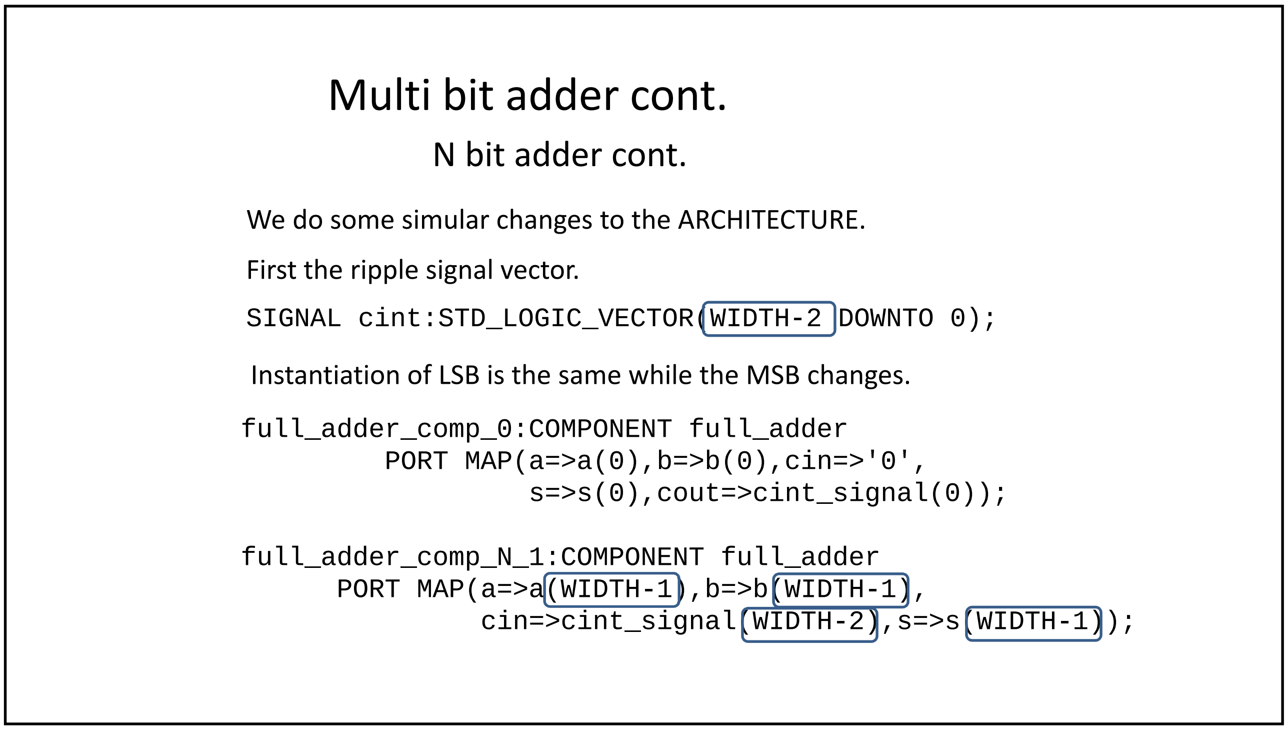
Signal type

Default value (can in some cases be left out)

Notice

The value is given by the GENERIC

```
ENTITY adder_x_bit IS
   GENERIC(WIDTH:NATURAL:=32);
   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 adder_x_bit;
```



Multi bit adder cont.

N bit adder cont.

We do some simular changes to the ARCHITECTURE.

First the ripple signal vector.

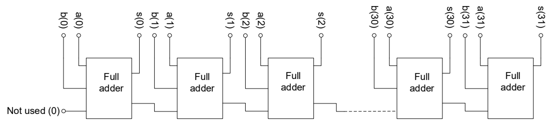
```
SIGNAL cint:STD_LOGIC_VECTOR(WIDTH-2 DOWNTO 0);
```

Instantiation of LSB is the same while the MSB changes.

```
full_adder_comp_0:COMPONENT full_adder
         PORT MAP(a=>a(0),b=>b(0),cin=>'0',
                  s=>s(0),cout=>cint_signal(0));

full_adder_comp_N_1:COMPONENT full_adder
      PORT MAP(a=>a(WIDTH-1),b=>b(WIDTH-1),
               cin=>cint_signal(WIDTH-2),s=>s(WIDTH-1));
```

Multi bit adder cont.

N bit adder cont.



Finally the LOOP GENERATE statement.

```
G:FOR i IN 1 TO WIDTH-2 GENERATE
      full_adder_comp_i:COMPONENT full_adder
         PORT MAP(a=>a(i),b=>b(i), cin=>cint_signal(i-1),
                  s=>s(i),cout=>cint(i));
   END GENERATE;
```

Multi bit adder cont.

To use our generic multi bit adder

Let´s say that we want a design with two adders, one with 8 bits and one with 16 bits.

First the entity.

```
ENTITY dual_adder IS
   GENERIC(WIDTH8:NATURAL:=8,
           WIDTH16:NATURAL:16)
   PORT(cin:IN STD_LGIC;
        a8:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        b8:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        a16:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        b16:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        s8:OUT STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        s16:OUT STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END dual_adder ;
```

Ports for the 8 bit adder

Ports for the 16 bit adder

Multi bit adder cont.

To use our generic multi bit adder

```
ENTITY dual_adder IS
   GENERIC(WIDTH8:NATURAL:=8,
           WIDTH16:NATURAL:16)
   PORT(a8:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        b8:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        a16:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        b16:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        s8:OUT STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        s16:OUT STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END dual_adder ;
```

Then the architecture.

```
ARCHITECTURE arch_dual_adder OF dual_adder IS

  COMPONENT adder_x_bit IS
   GENERIC(WIDTH:NATURAL:=32);
   PORT(cin:IN STD_LOGIC;
        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);
        cout:OUT STD_LOGIC);
END COMPONENT adder_x_bit;
```

Note that this version have both cin and cout

We only need to declare the component once although it´s used, instantiated, twice with different number of bits.

Multi bit adder cont.

To use our generic multi bit adder

```
ENTITY dual_adder IS
   GENERIC(WIDTH8:NATURAL:=8,
           WIDTH16:NATURAL:16)
   PORT(a8:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        b8:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        a16:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        b16:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        s8:OUT STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        s16:OUT STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END dual_adder ;
```

Then we instantiate the adders.

```
adder_8_bit_comp:
COMPONENT adder_x_bit IS
   GENERIC MAP(WIDTH=>WIDTH8);
   PORT MAP(cin=>cin,a=>a8,b=>b8,s=>s8);

adder_16_bit_comp:
COMPONENT adder_x_bit IS
   GENERIC MAP(WIDTH=>WIDTH16);
   PORT MAP(cin=>’0’a=>a16,b=>b16,s=>s16,cout=>cout);
```

Notice

Notice

Note that the 8-bit adder has a cin but no cout

Note that the 16-bit adder has no cin but a cout