Behavioural Modelling in VHDL

The learning objectives for this section are:

- To become familiar with syntax of the PROCESS construct;
- To understand the behaviour of variables in processes;
- To become familiar with behavioural modelling using conditional statements in processes.
- To become familiar with different data types and use of package constructs for numeric operations.
- To become familiar with the syntax and limitations of LOOP constructs.

The Process Body

We defined processes in Section 3.2 as either a signal assignment or a collection of statements inside an explicitly declared process body. In this section we will look at how combinational and sequential logic can be modelled using process bodies. A PROCESS has a collection of sequential statements, i.e., each statement is executed in order from top to bottom, similar to a conventional programming language. Because of the sequential flow of statements, a process body allows us to use high-level behavioural constructs with conditional statements.

Processes Using Sensitivity Lists

In Section 3.3 we implemented a MUX in two ways; in one we modelled the functionality of the logic gates directly and 'wired' them up, while in the other we used a one-line explicit MUX construct. Consider yet another implementation of a multiplexer:

```
ENTITY mux IS
  PORT (
    a:IN BIT;
    b:IN BIT;
    address:IN BIT;
    q:OUT BIT
 );
END mux;

ARCHITECTURE sequential OF mux IS
BEGIN
  select_proc : PROCESS (a,b,address)
  BEGIN
```

```
        IF (address = '0') THEN
          q <= a;
        ELSIF (address = '1') THEN
          q <= b;
        END IF;
      END PROCESS select_proc;
    END sequential;
```

As per our convention all keywords are shown in upper-case. This process has a label select_proc which is optional. The statements inside the process body execute one after the other. The process gets activated whenever an event occurs on one of the signals in the sensitivity list, specified in parenthesis after the keyword PROCESS at the beginning, which are a and b. It then runs through all the statements and suspends, to be awakened again upon an event on one of the signals in the sensitivity list.

- If you don't add all signals that are read to the process sensitivity list, the process won't be triggered even if there are events on the inputs. This is a common source of error in simulation. If a signal is read inside a process, it should always be in the sensitivity list of the process.
- In this example, signals that are read comprise all signals on the right-hand side of the assignment operator (i.e., a and b), and all signals read inside of the conditional operators IF and ELSIF (i.e., address).

The IF-THEN-ELSE form of conditional checking can only be used inside a PROCESS body. The dataflow architecture in the "mux.vhd" file shows how a conditional statement can be implemented outside a process body:

```
ARCHITECTURE dataflow OF mux IS
BEGIN
  q <= a WHEN address = '0' ELSE b;
END dataflow;
```

In both approaches it is important that the value of the output is specified in all possible selection branches. As we are implementing combinational logic in both instances, by definition, the output always has to be driven by the inputs. As address is a single bit, it can take the values of '0' and '1'. Thus the value of the output for both possibilities has to be specified. If instead, address was a 2-bit vector (for example), then there are four branches in the conditional tree, and the output has to be specificied for all four possibilities.

An example of a concurrent MUX with a 2-bit selection input is given below, for an entity named mux4, with data inputs a, b, c and d, 2-bit wide input select vector select, and output q.

```
ARCHITECTURE example1 OF mux4 IS
BEGIN
  q <= a WHEN select = "00" ELSE
     b WHEN select = "01" ELSE
     c WHEN select = "10" ELSE
     d;
END mux4;
```

Here too, we have specified the behaviour for all possible selection outcomes, using the final ELSE clause to choose the selection branch corresponding to select being "11". The following code would not only be functionally wrong, it would violate the semantical requirement that the output in combinational logic should always be driven by the inputs, as we don't explicitly specify what should happen to the output when select is "11".

```
ARCHITECTURE bad_example OF mux4 IS
BEGIN
  q <= a WHEN select = "00" ELSE
     b WHEN select = "01" ELSE
     c WHEN select = "10";
END mux4;
```

It would however (perhaps unfortunately) compile, as it is syntactically correct. Even if we were uninterested in the value of y when select is "11", this is not the correct way to describe the functionality. Unspecified conditional branches imply sequential logic, i.e., logic that holds the output at some value when the inputs do not explicitly drive it. Hence this code would synthesise to some hardware that is completely different from what you intended.

We could however use a final ELSE to cover many outcomes should the functionality dictate it. For example, imagine that in the above case, we had only three different assignment requirements, that y should be assigned a when select is "00", b when "11" and c when "01" or "10". The following code implements this functionality, while ensuring that all selection branches are covered:

```
ARCHITECTURE example2 OF mux4 IS
BEGIN
  q <= a WHEN select = "00" ELSE
     b WHEN select = "11" ELSE
     c;
END mux4;
```

Here the final ELSE clause covers both the "10" and "01" branches.

- A MUX can be implemented using sequential, conditional branching logic steps inside a process body, or using a concurrent construct outside a process body. Both methods are equivalent, and you can use whatever makes the code more readable.
- Whatever method you use, be sure to specify a value for the output in every conditional branch, as otherwise some unknown sequential logic is implied.

Processes Using wait Statements

Instead of a sensitivity list, it is also possible to have a WAIT statement. The following sequential2 implementation uses a WAIT statement and is functionally completely equivalent to the sequential implementation that had a sensitivity list, and used sequential conditional statements (using IF-THEN-ELSE clauses) to implement a MUX:

```
ARCHITECTURE sequential2 OF mux IS
BEGIN
  select_proc : PROCESS
  BEGIN
    IF (address = '0') THEN
```

```
      q <= a;
    ELSIF (address = '1') THEN
      q <= b;
    END IF;
    WAIT ON a,b,adress;
  END PROCESS select_proc;
END sequential2;
```

How this works is that the process will run once after initialisation and then suspend at the WAIT statement. The signals specified in the WAIT ON clause are then monitored for activity; anytime there is an event on a or b, the process awakens and executes the code between BEGIN and WAIT.

- A PROCESS cannot have both a sensitivity list and WAIT statements, it has to be one or the other.

It is possible to use more than one WAIT statement in the same process. In this case, the process will pause at WAIT statements in the order in which they are encountereed, and monitor signals specified in the WAIT ON clause to reawaken the process. Once awakened, the statements up to the next WAIT process will be executed, before the process suspends again.

Although more than one WAIT statement is allowed syntactically, you should not use it in general to describe synthesisable RTL, as quite often it does not result in synthesisable logic. Most synthesis programmes do not allow more than one WAIT statement. The use of multiple WAIT statements is useful in domain specific high-level synthesis (HLS) programmes which are not the focus of this unit.

You can avoid WAIT statements altogether, as everything you need to do when writing RTL for synthesis is possible with sensitivity lists, and it ensures you don't pick up bad habits, such as placing a WAIT statement in the middle of a code block rather than at the beginning or end, which can result in unreadable code.

- In this Unit, it is recommended you use sensitivity lists rather than WAIT statements when describing logic. It is of course perfectly fine to use WAIT statements for synthesis if you know how to use them correctly.

You should not hesitate to use multiple WAIT statements inside testbenches though, as testbenches are not intended to be synthesised or even model logic. Sometimes it is very useful to be able to suspend a process for a specific amount of time, as for example in the following statement:

```
wait for 30 ns;
```

It goes without saying that the above statement can never be synthesised!

Finally, it is worth emphasising again that a single line signal assignment is equivalent to a process with all the signals on the right-hand side of the assignment operator being in the sensitivity list. For example, the following signal assignment:

```
Y <= (A NAND B) OR C;
```

is identical in terms of functionality to the following process.

```
PROCESS(A, B, C)
BEGIN
  Y <= (A NAND B) OR C;
END PROCESS;
```

Also as mentioned previously, collections of processes describe the functionality of the component, whether the processes are implicit or explicit.

Simulating the MUX Implemented Using a Process

1. Copy the code of the sequential architecture body given at the beginning of Section 4, and paste it into the "mux.vhd" file to provide a third implementation of the MUX.
2. Add a fourth implementation in an architecture body entitled bool, based on the logic equations you derived in Exercise 3.3A 1.
3. Modify the testbench by adding two extra signals seqResult and boolResult, and instantiating two new components bound to the sequential and bool architectures, with the same inputs as the others but with seqResult and boolResult as the outputs respectively.
   - It is important that you stick to the specified naming convention as your programme will be checked using an automated testbench that will assume those signal and architecture names. The testbench supplied for Task3.4A uses the specified names so that you can check for any typos. This testbench should compile and run with your "mux.vhd" file with no modifications.
   - Throught these exercises a compilation error as a result of failing to conform to the specified naming convention will result in an automatic penalty up to a maximum of 30% deduction for that question, even if the syntax is correct. Discipline in conforming to specifications is very important in writing code, and this policy will apply to all questions / assignments where you have to submit source code as the entirety or part of the answer.
4. Observe the outputs of the components you added, and check if the outputs are identical to the outputs from the other components (disregarding the glitches). If the outputs are not identical (disregarding the glitches) make any necessary corrections to the code of the components you added.

Submit your MUX implementation with four implementations

1. Submit your "mux.vhd" source as the answer to Quiz 1, Question 5. Make sure you implement four implementations with the same outputs (including the provided sequential architecture). And mention which implementation do you prefer and the possible reason?

Variables in a Process

A VARIABLE is a value holder in VHDL, that can only be declared and used within a PROCESS; i.e., its scope is limited to the process in which it was declared. The syntax is illustrated in the following code:

```
ENTITY decoder IS
  PORT (
    a:IN BIT_VECTOR(2 DOWNTO 0); -- BIT_VECTOR is a 1D array of BIT
    q:OUT BIT_VECTOR(7 DOWNTO 0)
);
END decoder;

ARCHITECTURE behav OF decoder IS
BEGIN
```

```
    PROCESS(a)
      -- variable scope is internal to process
      variable var_out: BIT_VECTOR(7 DOWNTO 0);
    BEGIN
      CASE a IS
        WHEN "000" => var_out := "00000000"; -- zero
        WHEN "001" => var_out := "00000010"; -- one
        WHEN "010" => var_out := "00000100"; -- two
        WHEN "011" => var_out := "00001000"; -- three
        WHEN "100" => var_out := "00010000"; -- four
        WHEN "101" => var_out := "00100000"; -- five
        WHEN "110" => var_out := "01000000"; -- six
        WHEN "111" => var_out := "10000000"; -- seven
        WHEN OTHERS => var_out :="11111111"; -- catch all else branch
      END CASE;
      q <= var_out;
    END PROCESS;
  END behav;
```

There are a few things going on here; firstly, note how the variable declaration takes place after the process keyword and before begin, how the scope is local to the process in which the variable is declared, and the different symbol for assignment (i.e., := compared to <= for signal asignments).

There is also a new construct, CASE. The CASE construct offers a compact way of implementing a flat conditional selection tree. When there are multiple choices as in this case, a CASE structure is more readable than multiple nested IF-THEN-ELSE clauses. Note the trailing WHEN OTHERS statement which acts as a catch-all branch when none of the conditions to that point are met. In this particular case having this line is superfluous as all possible conditions are enumerated. Nevertheless, always having this clause is a good idea to ensure that if a possible branch or branches are missed out, perhaps accidentally, the logic functionality is specified. As we learnt earlier in this section, omitting a conditional branch implies sequential rather than combinational logic.

In terms of timing, variables and signals are fundamentally different, in that variable assignments incur no assignment delay, and their values are updated immediately; i.e., simulation time does not need to advance for VARIABLE values to be updated. Does this mean that you can use variables and "bypass" the signal assignment delay and its semantic implications?

Not at all. Variables are internal in scope to a process. To communicate outside a process you need to use signals that are declared within the architecture body and are external to the process. Thus, even if you use variables within a process, updating of a signal or signals need to take place for the values stored in the variables to be read outside the process.

Variables may be useful in enforcing encapsulation and hiding the internal details of a module. Variables are certainly useful in test benches.

Data Types

To this point we have used the native types BIT and BIT_VECTOR which is just a 1-D array of BIT scalars. While a binary type such as BIT which can take the value '0' or '1' is sufficient to describe binary logic, in hardware, at least one other value is essential, high impedance or 'Z'. A node is considered to be high impedance when it is not explicitly driven, but left floating, as would happen for example with a tri-state driver that can disconnect the output.

In practice, several other values are useful, and the STD_ULOGIC (note the 'U' before LOGIC) type defines nine values in total:

- 'U': Uninitialized. The value has not been set to anything. Often seen at the beginning of a simulation before a global or system reset and signals have not been given an initial value.

- 'X': Unknown with a strong signal drive. The signal is driven to a value that cannot be determined. This is often the result of some error, for example a signal being driven simultaneously to '0' and '1'.
- '0': logic zero with a strong signal drive.
- '1': logic one with a strong signal drive.
- 'Z': High Impedance.
- 'W': Weak unknown with a weak signal drive. If a signal has two drivers, one weak, and the other strong, the strong driver will win out. For example, an old fashioned bus interface with pull-up resistors would have several weak highs or 'H' (see below) and a strong '0', when the strong zero would win.
- 'L': logic 0 with a weak signal drive.
- 'H': logic '1' with a weak signal drive.
- '-': Don't care. Used in synthesis tools to denote don't cares for logic optimisation.

The STD_ULOGIC type is an example of an enumerated type. It is defined in an IEEE package called STD_LOGIC_1164 as:

```
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

The STD_LOGIC type is a resolved version of the STD_ULOGIC type. The type definition includes a resolution function, providing a means of resolving the value of a signal with multiple drivers based on rules such as the one above, where a strong driver wins out. What this means is that if you drive a signal simultaneously with two drivers, it won't result in a compilation error. However, this still has serious implications for synthesis. If you do need to drive the same signal (wire) with two drivers (for example in modelling a bus), you have to pay special consideration to the implementation. Generally, we won't require the resolution capability of the STD_LOGIC type in this course. Therefore, it's better to always use the **STD_ULOGIC** type, as that way, if you mistakenly assign the same signal in two places, the compiler will tell you immediately and you can fix the error.

Both STD_ULOGIC and STD_LOGIC types can be declared as scalars or vectors (1-D arrays). i.e., STD_ULOGIC_VECTOR and STD_LOGIC_VECTOR. similar to BIT and BIT_VECTOR. However, you cannot perform arithmetic operations on these data types directly, you have to use derived types that interpret the bit vectors as a binary number, either in unsigned or signed representation.

There are generally two widely used packages, NUMERIC_STD and STD_LOGIC_ARITH, which are more or less equivalent. The latter has origins in a proprietary library from Synopsys and is a de facto standard, whereas the former was designed from scratch for consistency and is supported by an IEEE standard ensuring that different vendors implement it in exactly the same way. Therefore you should always use **NUMERIC_STD** for new designs unless you're dealing with legacy code which uses types defined in STD_LOGIC_ARITH.

The NUMERIC_STD package implements two types, UNSIGNED and SIGNED. Physically, these types are identical to the STD_LOGIC_VECTOR type in that they are 1-D arrays (i.e., vectors) of STD_LOGIC. The type declaration for these two types are:

```
--------
type UNSIGNED is array ( NATURAL range <> ) of STD_LOGIC;
type SIGNED is array ( NATURAL range <> ) of STD_LOGIC;
--------
```

Declaring a signal as UNSIGNED for a width of N (eg: UNSIGNED(0 to N-1), UNSIGNED(N-1 DOWNTO 0) etc) causes the bit vector to be interpreted as an unsigned integer with values ranging from 0 to 2^{N-1}. Declaring a signal as SIGNED for a width of N (eg: SIGNED(0 to N-1), SIGNED(N-1 DOWNTO 0) etc) causes the bit vector to be interpreted as a signed integer with values ranging from -2^{N-1} to 2^{N-1}-1). The following example (after Lewis [4]) illustrates the difference in the decimal interpretation of these types.

```
SIGNAL A_unsigned   : UNSIGNED 3 downto 0;
SIGNAL B_signed     : SIGNED 3 downto 0;
SIGNAL C_stdLogVect : STD_LOGIC_VECTOR (3 downto 0);

A_unsigned   <= "1111";  -- decimal 15
B_signed     <= "1111";  -- decimal -1
C_stdLogVect <= "1111";  -- open to any interpretation
```

Not only does the NUMERIC_STD package provide the capability to perform comparison and arithmetic operations on UNSIGNED and SIGNED types, it allows UNSIGNED / SIGNED operands to be combined with INTEGER operands by providing overloaded functions for the relevant numeric and comparison operators. The package also includes conversion functions. These are summarised below. All of these operators and functions are supported for synthesis.

- **Arithmetic operators (for UNSIGNED / SIGNED and INTEGER operands).**
  - ABS (Return absolute value; Usage: ABS(a);)
  - + (Add; Usage: a+b; a+10;)
  - - (Subtract; Usage: a-b; 10-a; a-10;)
  - * (Multiply; Usage: a*b; 10*a; a*10;)
  - / (Divide; Usage: a/b; 10/a; a/10;)
  - REM (Remainder, defined as A REM B = A-(A/B)*B; result has sign of 1st operand; Usage: a REM b)
  - MOD (Modulus, defined as A mod B = A B * N for integer N; result has sign of 2nd operand; Usage: a MOD b)

  Comparison operators (for UNSIGNED / SIGNED and INTEGER operands).
  - > (Greater than)
  - < (Less than)
  - <= (Less than or equal to)
  - >= (Greater than or equal to)
  - = (Equal to)
  - /= (Not equal to)
- **Shift functions (for UNSIGNED / SIGNED operands).**
  - SHIFT_LEFT (Usage: SHIFT_LEFT(a,n) where a is vector and n is number of times to left shift. The vacated positions are filled with '0'. If the vector is a SIGNED type, the leftmost sign bit is untouched.)
  - SHIFT_RIGHT (Usage: SHIFT_RIGHT(a,n) where a is vector and n is number of times to right shift. The vacated positions are filled with Bit '0'. If the vector is a SIGNED type, the leftmost sign bit is untouched.)
  - ROTATE_LEFT (Usage: ROTATE_LEFT(a,n) where a is vector and n is number of times to rotate left. If the vector is SIGNED type, sign bit is not treated any differently.)
  - ROTATE_RIGHT(Usage: ROTATE_RIGHT(a,n) where a is vector and n is number of times to rotate right. If the vector is SIGNED type, sign bit is not treated any differently.)

    - RESIZE(Usage: RESIZE(a,n) where a is vector and n is the new size. To create a larger vector, the new [leftmost] bit positions are filled with '0' for UNSIGNED operands and with the sign bit for SIGNED operands. When truncating the leftmost bits are dropped for UNSIGNED operands, while the sign bit is retained along with the rightmost part for SIGNED operands.
- **Conversion functions**
    - Converting betwen STD_ULOGIC_VECTOR and UNSIGNED / SIGNED only requires type casting.
    - Converting to and from integer requires a conversion function
        - TO_INTEGER (Usage: TO_INTEGER(a) where a is vector.)
        - TO_UNSIGNED (Usage: TO_UNSIGNED(a,n) where a is vector and n is width of array.)
        - TO_SIGNED (Usage: TO_SIGNED(a,n) where a is vector and n is width of array.)

We can use these data types and associated utility functions by including the IEEE library and related packages by including the following lines at the top of the VHDL file:

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

The first line above simply declares that we will use packages contained in this library. With the next two lines we declare that we will use all of the constructs in two packages called std_logic_1164 (defining the std_ulogic and derivative types such as std_logic and related utilities) and numeric_std (implementing numeric operations).

Loop Constructs

VHDL offers two loop constructs, the FOR loop and the WHILE loop which need to be used with extreme care when writing code for synthesis. Loop constructs in HDLs are synthesised by unrolliing the loop and essentially making copies of the function being iterated. The number of copies is the number of times the loop is iterated. Thus, only loops where the limits of the loop index are constant can be synthesised. This makes complete sense if you think about it: how can you make copies of a function in hardware if the number of copies is variable? Synthesising loops where the loop range is not fixed is impossible.

FOR Loop

The FOR LOOP repeatedly executes a set of statements contained within the FOR LOOP body. The loop index and loop boundaries are part of the construct. Given below is an example which will synthesise.

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

ENTITY add_bcd IS
  PORT (
    a:IN STD_ULOGIC_VECTOR(0 TO 8); -- 3 BCD digits
    q:OUT STD_ULOGIC_VECTOR(0 TO 4)
  );
END add_bcd;

ARCHITECTURE for_loop OF add_bcd IS
  SIGNAL sig1: UNSIGNED(0 to 8);
BEGIN
  PROCESS(sig1)
```

```
      -- variable of type integer with constrained range and initial value
      VARIABLE tmp:INTEGER RANGE 0 TO 27:=0;  -- store the sum
    BEGIN
        FOR i IN 0 TO 2 LOOP
          tmp:=tmp+TO_INTEGER(sig1(0+i*3 to 2+i*3));
        END LOOP;
        q <= STD_ULOGIC_VECTOR(TO_UNSIGNED(tmp,5));
    END PROCESS;
    sig1 <= UNSIGNED(a);
  END for_loop;
```

This example brings together several of the constructs introduced in the previous few sections. First of all, the functionality of the add_bcd component is to add three binary-coded decimal (BCD) numbers together. In BCD format decimal digits are represented by four bits which can take values between '0' (i.e., "0000") and '9' (i.e., "1001"). In the BCD format the values from decomal '10' to decimal '15' i.e., bit patterns "1010" through "1111" have no relevance. There is also no sign bit so all BCD digits are positive.

The BCD numbers are presented packed together on a single input line a. The first four bits represent the first BCD digit, the next four bits the second digit and so on. The input and output vectors are declared as STD_ULOGIC_VECTOR.

The straightforward method to perform the addition is to convert each BCD number to integer form, and add the three integers. It is of course possible to convert from BCD to integer form using the definition of BCD, i.e., $int=b_0*2^0+b_1*2^1+b_2*2^2$ where b_0 is the least signficant (right-most) bit in the bits making up a BCD digit. It is, however, much simpler to use the built-in conversion functions available in the NUMERIC_STD package. A signal of type UNSIGNED called sig1 is declared to which the input lines are connected. As the input is of type STD_ULOGIC_VECTOR and the signal is of type UNSIGNED, the strong typing rules of VHDL prevent a direct assignment; i.e., the line sig1 <= a; will result in a syntax error being thrown by the compiler, which will report a type mismatch. A simple type casting sig1 <= UNSIGNED(a); will suffice to meet the strong typing rules. In this case a conversion function is not needed because both STD_ULOGIC_VECTOR and UNSIGNED share the same subtype.

The incoming BCD digits are available on the sig1 signal as a SIGNED vector because of the assignment sig1 <= UNSIGNED(a);. Thus, it is possible to use the built-in conversion function TO_INTEGER to convert the buts contained in this vector to integer format. It is possible to access a bit slice of any vector by using the appropriate index range. In this case, as there are three BCD digits, it is convenient to use a loop. A variable of type INTEGER is declared to store the sum which is iteratively calculated within the body of the loop. Note that the maximum sum of any three BCD digits is 27. Thus, the declaration restricts the range of the integer to be between 0 and 27. Declaring a range for integers genarlly results in more efficient hardware, as the synthesiser can use the minimum number of bits required to implement the range, in this case 5, rather than the standard width, which is usually 32. A 5-bit adder consumes far less power than a 32-bit adder. As statements inside a PROCESS body are executed sequentially, the sum is calculated and then assigned to the output. As the output is of type STD_ULOGIC_VECTOR, this assignment also requires type casting. Note that the process body itself and the signal assignment outside the process body are concurrent. This means, for example, the order in which the process body and the assignment to sig1 occur are not important.

[A]This comes with a caveat. If the targeted hardware platform only has adders of a given width, restricting the range may not reduce the bit width. nevertheless it is good practice to always restrict the range, to provide the synthesiser with the maximum amount of information. Also, implementation errors may be caught in a simulation if the range exceeds the declared range, which may be missed if the bit width is not restricted. Of course this also applies for other supported arithmetic operators, eg: multiplication.

The variable is initialied to zero, and having an initial value of zero is clearly essential for realising the correct functionality. This works in simulation, but does this work in synthesis? i.e., can initial values for variables or signals be synthesised? That depends on the target hardware platform. If the target is a custom IC, initial values for synthesis make no sense. However, if the target is an FPGA, at time of configuration, the initial values can be loaded into the confirguration bitstream. It is always good practice though to have a hardware reset, which when asserted explicty performs initialisation. You will see how to do that when we look at implementing sequential logic in the next section.

For the record, the loop parameter does not need to be an integer. It simply takes on the type required by the specified range. In the above example the range 0 to 2 clearly implies an integer. However, it could also be an enumerated type. For example, consider the following code which essentially populates an array with the same information in all locations:

```
PACKAGE filter IS
  TYPE my_filter_type is (LP, HP, BP); -- custom enumerated type (low-, high- and
  SUBTYPE bandwidth is INTEGER RANGE 20 to 200; -- declare a subtype of range rest
  TYPE bandwidth_array is ARRAY (my_filter_type) OF bandwidth; -- array of range r
END filter;

USE WORK.filter.ALL;

ENTITY writeArray IS
  PORT (
    bw:IN bandwidth ;
    d:OUT bandwidth_array
  );
END writeArray;

ARCHITECTURE example OF writeArray IS
BEGIN
  PROCESS(bw)
  BEGIN
    FOR filter IN my_filter_type LOOP
      d(filter) <= bw;
    END LOOP;
  END PROCESS;
END example;

USE WORK.filter.ALL;

ENTITY tb_writeArray IS END tb_writeArray;

ARCHITECTURE eg OF tb_writeArray IS

  COMPONENT writeArray IS
    PORT (
      bw:IN bandwidth ;
      d:OUT bandwidth_array
    );
  END COMPONENT;

  SIGNAL bw_in:bandwidth;
  SIGNAL bw_out:bandwidth_array;

  FOR A1:writeArray USE ENTITY work.writeArray(example);

BEGIN
  A1:writeArray PORT MAP(bw=>bw_in, d=>bw_out);

  bw_in<=
   20,
   40 AFTER 10 ns,
   60 AFTER 15 ns,
   40 AFTER 20 ns,
   100 AFTER 25 ns;
END eg;
```

Here too, the loop parameter filter is not declared, and the indices are asumed in order left to right of the enumerated type, i.e., LP, HP and BP. The array elements are addressed by the loop parameter filter.

One final thing to note is that the loop parameter cannot be written to, it can only be read.

- The loop parameter in the FOR loop does not need to be declared.
- The loop parameter has a constant range with static bounds, i.e., 0 to 2, and thus this construct is synthesisable.
- The arithmetic operations on the loop index in the add_bcd example are part of the indexing and don't result in adders or multipliers. The loop is unrolled, so that hardware copies for each iteration are synthesised.
- The loop parameter can be an integer or an enumerated type, which is inferred by the prescribed range,
- As the loop parameter is not declared, it is not a true value holder, and thus can only be read, not written to.

WHILE Loop

The WHILE LOOP repeately executes the set of statements contained within the WHILE LOOP body while a specified condition is true. Given below is the same example using a WHILE LOOP construct instead of a FOR LOOP.

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

ENTITY add_bcd IS
  PORT (
    a:IN STD_ULOGIC_VECTOR(0 TO 8); -- 3 BCD digits
    q:OUT STD_ULOGIC_VECTOR(0 TO 4)
  );
END add_bcd;

ARCHITECTURE while_loop OF add_bcd IS
  SIGNAL sig1: UNSIGNED(0 to 8);
BEGIN
  PROCESS(sig1)
    -- variables of type integer with constrained range and initial value
    VARIABLE i:INTEGER RANGE -1 TO 2;  -- loop index
    VARIABLE tmp:INTEGER RANGE 0 TO 27:=0;  -- store the sum
  BEGIN
      i:=-1;
      WHILE (i<2) LOOP
        i:=i+1; -- increment loop index;
        tmp:=tmp+TO_INTEGER(sig1(0+i*3 to 2+i*3));
      END LOOP;
      q <= STD_ULOGIC_VECTOR(TO_UNSIGNED(tmp,5));
  END PROCESS;
  sig1 <= UNSIGNED(a);
END while_loop;
```

There is an important difference here. The first is that the index used to control the loop iterations is not part of the construct itself and thus needs to be explicitly declared and initialised. Also the loop index needs to be initialised inside the process body rather than at the time of variable declaration as it needs to be reset everytime the process body is run. By contrast, in the FOR LOOP this was automatic, as it was part of the construct itself. This code will also synthesise with a similar schematic to the FOR LOOP as the loop bounds are static. Clearly, though, the FOR LOOP was a bit more convenient to use for this example. Xilinx Vivado is able to analyse the loop and understand that the loop bounds are static and this code does synthesise. You have to bear in mind that other synthesis tools may not be able to figure this out though.

LOOP on its own

Finally,the LOOP construct can be used on its own. In order to avoid an infinite loop an EXIT or WAIT statement should be included. If the loop is infinite, not only will it not synthesise, it will cause the simulation to hang. Here is another way of implementing the BCD adder which will also synthesise in Xilinx Vivado:

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

ENTITY add_bcd IS
  PORT (
    a:IN STD_ULOGIC_VECTOR(0 TO 8); -- 3 BCD digits
    q:OUT STD_ULOGIC_VECTOR(0 TO 4)
  );
END add_bcd;

ARCHITECTURE just_loop OF add_bcd IS
  SIGNAL sig1: UNSIGNED(0 to 8);
BEGIN
  PROCESS(sig1)
    -- variables of type integer with constrained range and initial value
    VARIABLE i:INTEGER RANGE -1 TO 2;  -- loop index
    VARIABLE tmp:INTEGER RANGE 0 TO 27:=0;  -- store the sum
  BEGIN
      i:=-1;
      LOOP
        i:=i+1; -- increment loop index;
        tmp:=tmp+TO_INTEGER(sig1(0+i*3 to 2+i*3));
        EXIT when i = 2; -- This is the exit clause
      END LOOP;
      q <= STD_ULOGIC_VECTOR(TO_UNSIGNED(tmp,5));
  END PROCESS;
  sig1 <= UNSIGNED(a);
END just_loop;
```

The key to all these implementations inferring legitimate hardware is that the loop bounds are static. Here is an example of a classic mistake where the bounds are not static.

```
----
ARCHITECTURE wont_synthesise OF comp IS
  SIGNAL sig1: INTEGER;
BEGIN
  sig1 <= in1; -- read in some value
  PROCESS(sig1)
    -- variables of type integer with constrained range and initial value
    VARIABLE i:INTEGER RANGE -1 TO 100;  -- loop index
  BEGIN
      i:=-1; -- increment loop index
      WHILE (i < sig1) LOOP -- sig1 is not static so this won't synthesise
        i := i+1;
        -- do something
      END LOOP;
  END PROCESS;
END wont_synthesise;
```

Unbounded loops are supported for simulation, where they may be useful for example in test benches.

- Any LOOP construct, whether on its own or with FOR or WHILE constructs, is unrolled to a parallel implementation of the functionality inside the loop; the number of hardware "copies" is equal to the number of iterations in the loop.
- Hence a LOOP is only synthesisable if the lower and upper bounds are static.

Summary

The following points were covered in this section.

- A process is a collection of statements executed sequentially. The statements comprise the process body. The process body as a whole runs concurrently with other process bodies and concurrent statements outside process bodies.
- A process has to have a sensitivity list or a WAIT statement, but not both. The sensitivity has to comprise all signals that are read inside the process. Any events on these signals cause the process to be activated, and run from beginning to end, once. It then suspends until the next event on a signal in the sensitivity list.
- Instead of a sensitvity list the process can have a WAIT ON <sig1>, <sig2> .. statement. The process runs and suspends at the WAIT statement, until the conditional clause (eg: an event on the named signals) is true. It is syntactically allowed to have multiple WAIT statements inside a process, but only processes with a single WAIT statement are synthesisable.
- Variables can be declared in processes and used as value holders. The scope of a variable is local to the process. Variable assignments incur no simulation delay (i.e., no delta delay as in signal assignments). In order for the value of the variable to be read outside the process, its value has to be assigned to a signal.
- The built-in types BIT and BIT_VECTOR don't allow the full range of values useful in modelling hardware. The STD_LOGIC_1164 package in the IEEE libray define an enumerated type STD_ULOGIC that, along with STD_ULOGIC_VECTOR, provide the full range of values needed to model and synthesise digital logic·
- The STD_LOGIC type is a resolved version of STD_ULOGIC. Clearly, it is not possible for two drivers to simultaneously drive the same signal. In hardware this could, for example, translate to a wire being driven simultaneously to '1' and '0', which would resolve in a short. The STD_LOGIC type gets around this issue for simulation by having a resolution which resolves between multiple drivers accoding to a priority scheme. For example, a strong '1' wins out over a weak '0', a strong '1' and a strong '0' result in an unkown 'X' and so on. However, this just ensures that there is no compilation error, it doesn't automatically synthesise some hardware capable of resolving between multiple drivers. It is useful in modelling hardware such as busses, but outside of a few very specific cases there is no good reason for having multiple drivers on the same wire. Thus, unless you are modelling one of these cases, using STD_LOGIC just masks errors. Hence always use STD_ULOGIC rather than STD_LOGIC as the default type.
- The NUMERIC_STD package from the IEEE library define two types called SIGNED and UNSIGNED, along with their vector versions, that are basically the same as STD_ULOGIC, but intepret the bit vectors as signed or unsigned numbers. Arithmetic operations can be directly performed on SIGNED and UNSIGNED types, and conversion functions are available for converting between SIGNED / UNSIGNED and INTEGER types. There is a different STD_LOGIC_ARITH package which allows you to perform arithemtic operations directly on the STD_LOGIC_VECTOR type. However, this is an old, proprietary package which is not standardised in the way that the IEEE packages are. Always use the NUMERIC_STD package rather than the STD_LOGIC_ARITH package. Never use both.
- LOOP constructs can be used inside a PROCESS body. The FOR LOOP construct defines a loop parameter and the bounds within the construct itself. The WHILE LOOP construct requires the loop index to be declared separately. A LOOP construct with an exit clause is also possible. All three types of loop constructs can be synthesised within Vivado, but other synthesis tools may be only be able to synthesise the FOR LOOP. In all cases, the bounds have to be static, as synthesis consists of unrolling the loop to realise a parallel implementation.