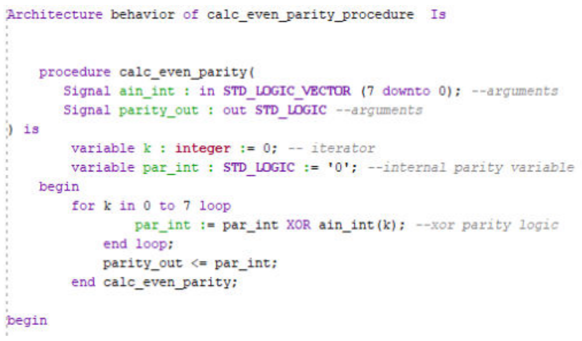
1. Introduction

As FPGAs are leveraged to solve more complex problems; sub-programs may be required. VHDL allows you to define sub-programs such as procedures and functions to improve the readability and exploit the re-usability of VHDL code. Once called, a procedure performs a task which can include timing control but does not directly return values, whereas a function returns values. Furthermore, functions are equivalent to combinatorial logic and can not be used to control events or delays. Simulating hardware models can be performed on testbenches using HDL programs. The purpose of this lab is to develop procedures and functions for modelling circuits as well as developing test benches for circuit design.

2. Procedures (Part 1)

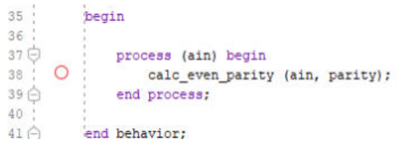
Procedures provide the ability to execute common pieces of code from different parts of a model. Moreover, they can have timing controls as well as calling upon other procedures and functions. The procedure must be defined between the **Architecture section** and the **Begin section** of the model. Additionally, procedures must contain its own begin/end section to process the provided arguments internally as well as variables to store data internally.



```
Architecture behavior of calc_even_parity_procedure  Is


    procedure calc_even_parity(
       Signal ain_int : in STD_LOGIC_VECTOR (7 downto 0); --arguments
       Signal parity_out : out STD_LOGIC --arguments
) is
        variable k : integer := 0; -- iterator
        variable par_int : STD_LOGIC := '0'; --internal parity variable
    begin
        for k in 0 to 7 loop
                par_int := par_int XOR ain_int(k); --xor parity logic
            end loop;
            parity_out <= par_int;
        end calc_even_parity;

begin
```

Figure 2.1: Procedure defined between **Architecture** and **begin section**.

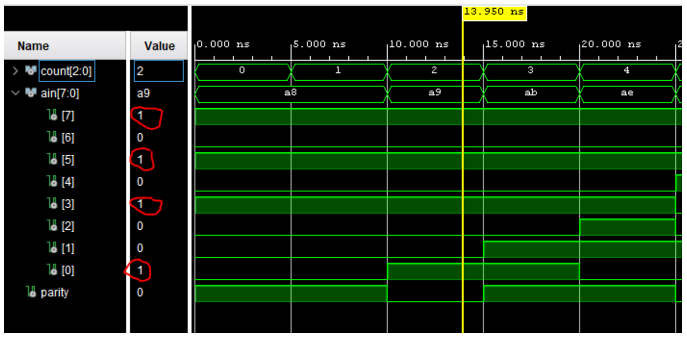
To demonstrate the efficacy using procedures, the "calc_even_pairty" procedure was created. As shown in Figure 2.1, the procedure "calc_even_pairty" accepts "ain_int" (input signal vector argument) and "parity_out" (output signal argument). Although the primary purpose of a procedure is not to output a value, when an output type signal is used as an argument, the procedure can produce an output by assigning the provided output argument(s) with internally processed data. As shown in Figure 2.2 "calc_even_parity" procedure was not directly assigned to an output signal. However, the procedure assigns "parity_out" with internally processed data in Figure 2.1.



```
begin

    process (ain) begin
        calc_even_parity (ain, parity);
    end process;

end behavior;
```

Figure 2.2: Procedure called, but instance cannot be directly assigned to a signal.

The “calc_even_pairty” procedure is a simple error detecting code which ensures that the total number of 1-bits in an input is even via XOR gates. As the primary purpose is to detect for even parities, it is considered good practice to implement this piece of code as a procedure for data integrity, as it can be frequently called upon in multiple parts of a program.



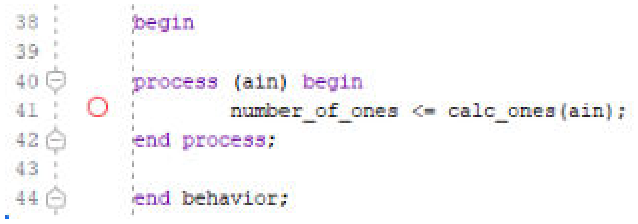
Name	Value
count[2:0]	2
ain[7:0]	a9
[7]	1
[6]	0
[5]	1
[4]	0
[3]	1
[2]	0
[1]	0
[0]	1
parity	0

Figure 2.3: “calc_even_parity” procedure simulated against a testbench.

To verify the feasibility of the “calc_even_parity” procedure, the procedure is simulated against a testbench. The “calc_even_parity” works as intended since it toggles the parity output when the number of ones in the “ains” input are even as shown in Figure 2.3.

3. Functions (part 2)

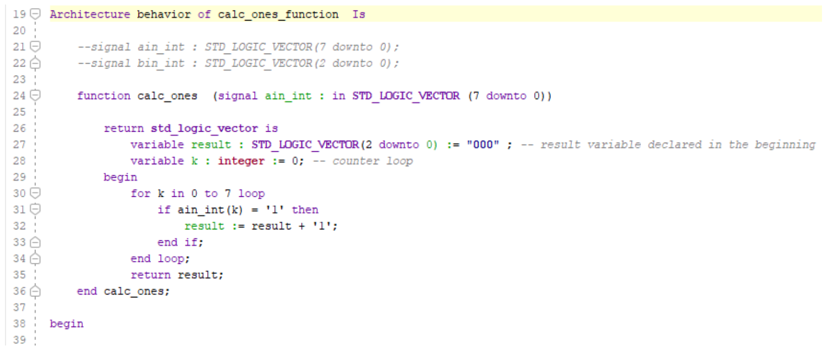
Similarly to procedures, functions are subprogrames defined between the **Architecture section** and **Begin section**, and come with their own begin/end section, process arguments internally, and use variables as shown in Figure 3.2. However, functions directly return a value. This is achieved by assigning its instance to an output signal as shown in Figure 3.1. (compared to a procedure where output signals are required to be passed as arguments). Additionally, functions may only implement combinatorial behaviour and functions cannot accept output or inout as arguments. Finally, functions cannot have delays, timing or event control constructs that are present.



```
begin

process (ain) begin
        number_of_ones <= calc_ones(ain);
end process;

end behavior;
```

Figure 3.1: Function called and instance is directly assigned to a signal.

To demonstrate the usage of functions, “calc_ones” function was generated as shown in Figure 3.2. The function accepts an 8 bit vector as an input to be processed internally, with the use of variables (similarly to Figure 2.1) to produce a 3-bit output by returning the number of ones in the 8 bit vector.



```vhdl
Architecture behavior of calc_ones_function  Is

    --signal ain_int : STD_LOGIC_VECTOR(7 downto 0);
    --signal bin_int : STD_LOGIC_VECTOR(2 downto 0);

    function calc_ones  (signal ain_int : in STD_LOGIC_VECTOR (7 downto 0))

        return std_logic_vector is
            variable result : STD_LOGIC_VECTOR(2 downto 0) := "000" ; -- result variable declared in the beginning
            variable k : integer := 0; -- counter loop
        begin
            for k in 0 to 7 loop
                if ain_int(k) = '1' then
                    result := result + '1';
                end if;
            end loop;
            return result;
    end calc_ones;

begin
```

Figure 3.2: “calc_ones” function; returns a vector that totals up the number of ones inside the 8 bit input vector.

As shown in Figure 3.2; the 8 bit input vector “ain_int” is processed internally with variables (counter loop “k”, and output vector “results”). As the primary purpose of a function is to return a value, “calc_ones” will internally sum up the number of ones in the 8 bit input vector of “ain_int” and present it as a 3 bit output port using a counter loop. Functions may prove to be versatile in a complex systems design where latency is a more significant factor as functions behave more combinatorially as opposed to procedures, which behave more sequentially.

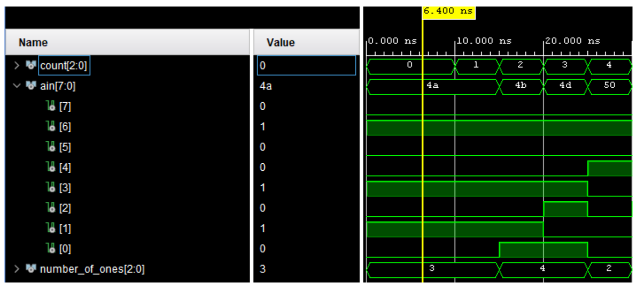


Figure 3.3: “calc_ones” function simulated against a testbench.

To verify the feasibility of the “calc_ones” function, the function is simulated against a testbench. The “calc_ones” works as intended since it manages to sum up the number of ones in the first sequence as indicated in Figure 3.3. Similarly to procedures, functions can be called upon in multiple parts of a model to complete repetitive tasks such as totalling up the number of ones in a vector.

4. Testbenches

Test benches act as test cases for firmware development. It is important to use as part of validating the functionality of a given VHDL design through waveform simulation; this method can drastically reduce time taken to test a circuit design that has a significant amount of input signals by automating the testing procedure. Furthermore, the processing time of testbench simulations is shorter than the time it takes for a circuit to synthesise, implement and generate bitstream. As shown in Figure 4.1 the arranged waveform timing can assist developers in assessing the correctness of circuit’s logic execution by running a testbench waveform in parallel with the circuit inputs/outputs.

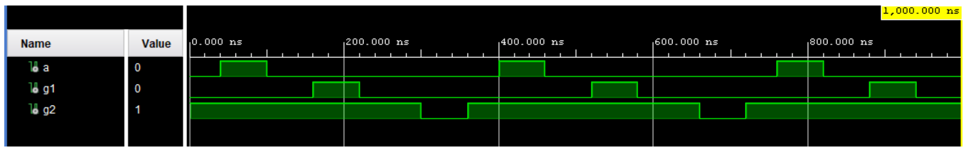
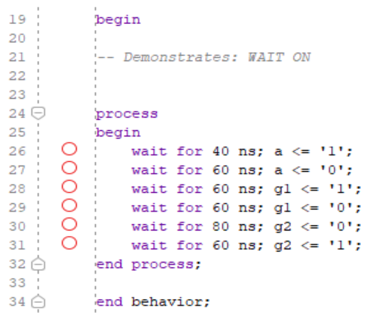


Figure 4.1: timing waveform testbench simulation

Testbenches for more complex circuit design should include inputs and output. However, if the inputs, outputs and timing controls are known by the circuit developer, testbench simulation may only require timing controls as shown in Figure 4.2, to verify the feasibility of a circuit design as illustrated in Figure 4.1.



```
begin

-- Demonstrates: WAIT ON


process
begin
    wait for 40 ns; a <= '1';
    wait for 60 ns; a <= '0';
    wait for 60 ns; g1 <= '1';
    wait for 60 ns; g1 <= '0';
    wait for 80 ns; g2 <= '0';
    wait for 60 ns; g2 <= '1';
end process;

end behavior;
```

Figure 4.2: Testbench simulation with only timing controls.

Conclusion

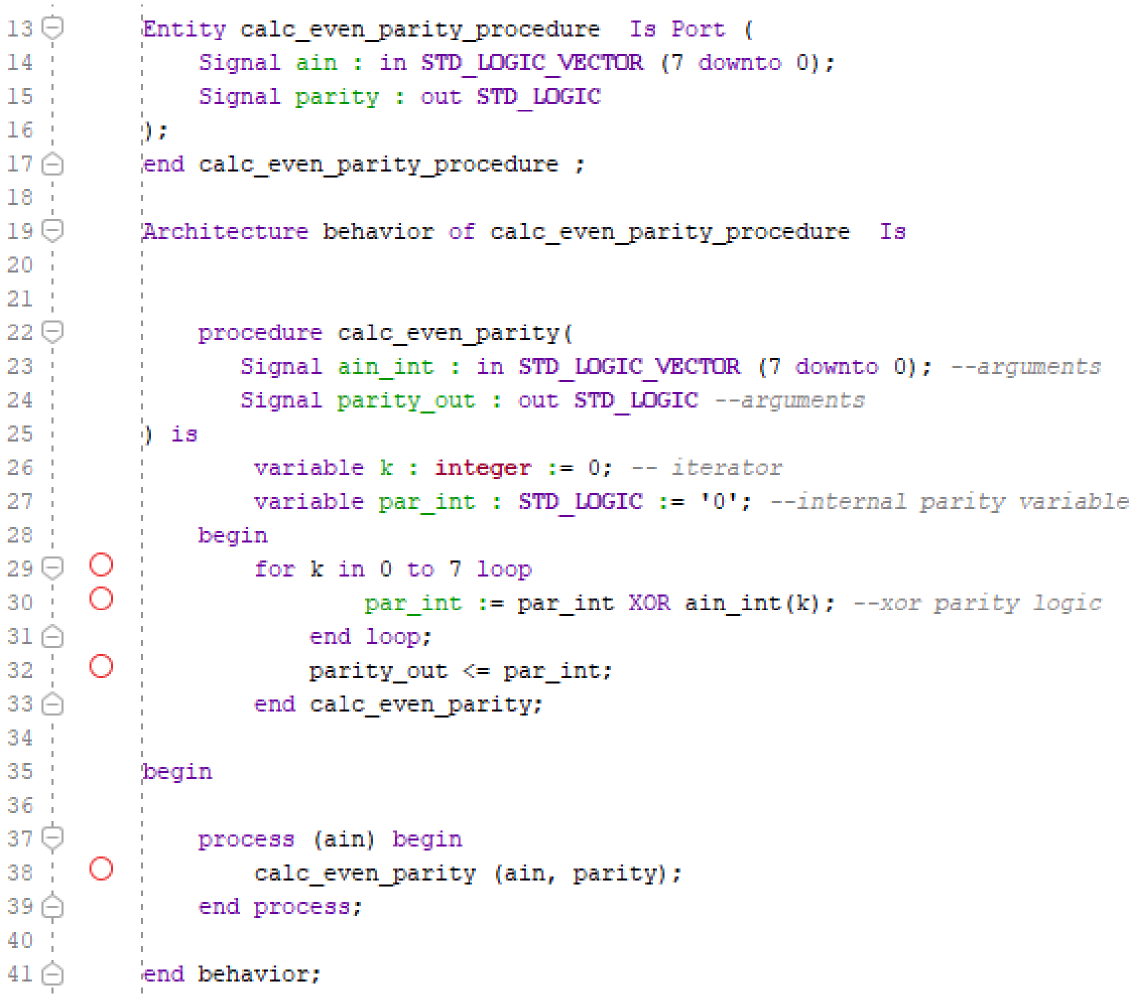
procedure	**both**	**functions**
More sequential	Process data internally; with variables and being/end section	More combinatorial
Does not return a value; but can change signals	Must be defined between **Architecture** and **begin section**	Directly return a value
Can have timing controls	Improves readability	Can not have timing controls
	Reusable code	

Table 1: difference between procedures and functions.

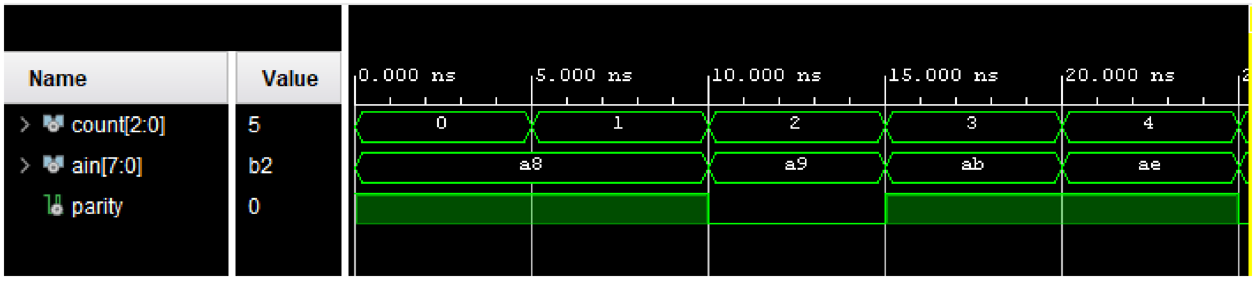
In conclusion, procedures and functions provide a high level approach to FPGA circuit designs by improving the readability of VHDL, as well as providing scalability towards more complex circuit designs due to their ability to provide reusable code in multiple parts of a model; difference in use case can be shown in Table 1. Complex circuit designs would greatly benefit from testbenches and may also mitigate production delays as testbench simulations can be quickly executed instead of waiting for the circuit to synthesise, implement and generate bitstream. Additionally, testbenches may assist in metastability as it can present a visual representation of timing anomalies for the firmware developer.

Appendix

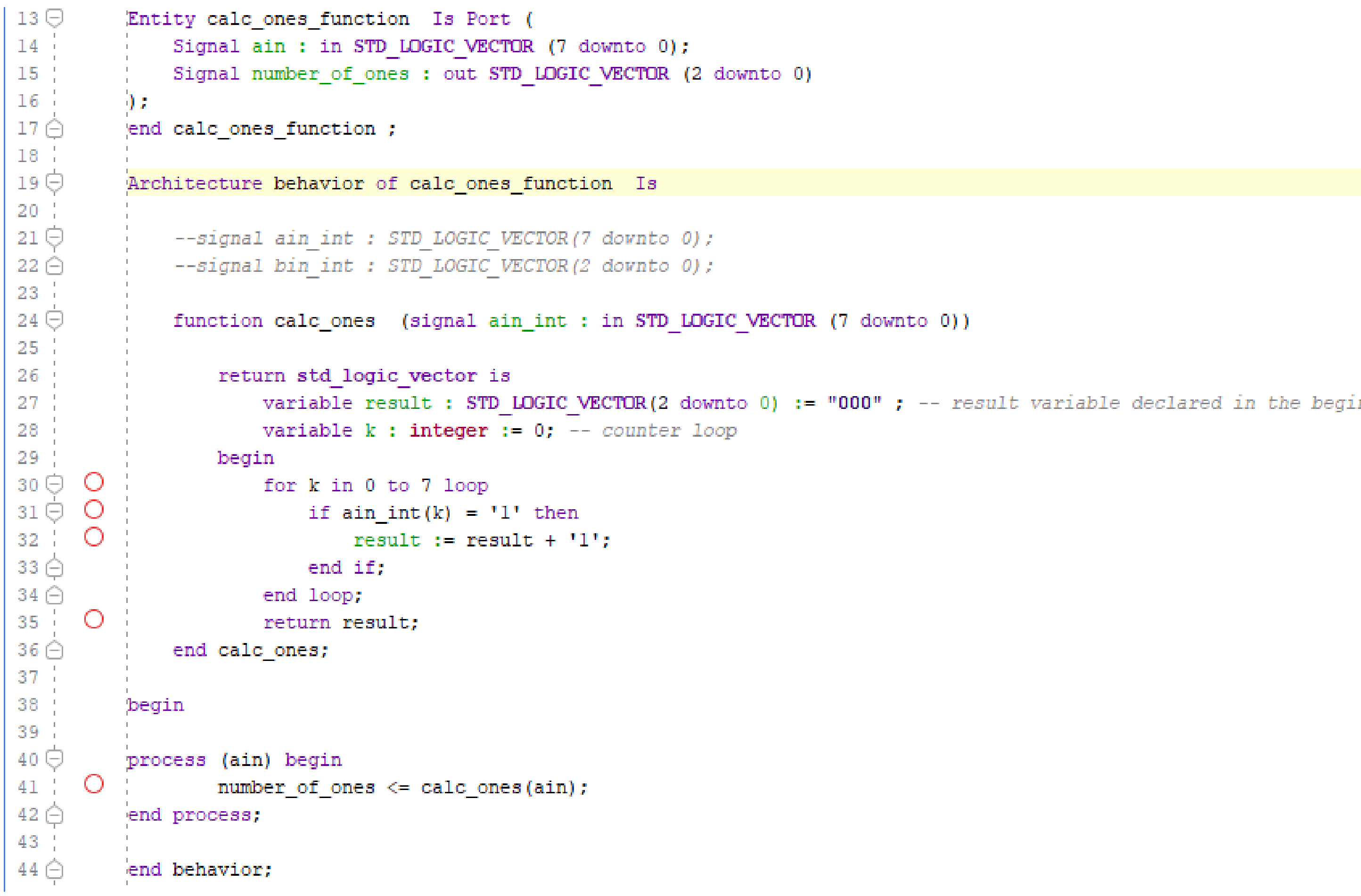
Part 1



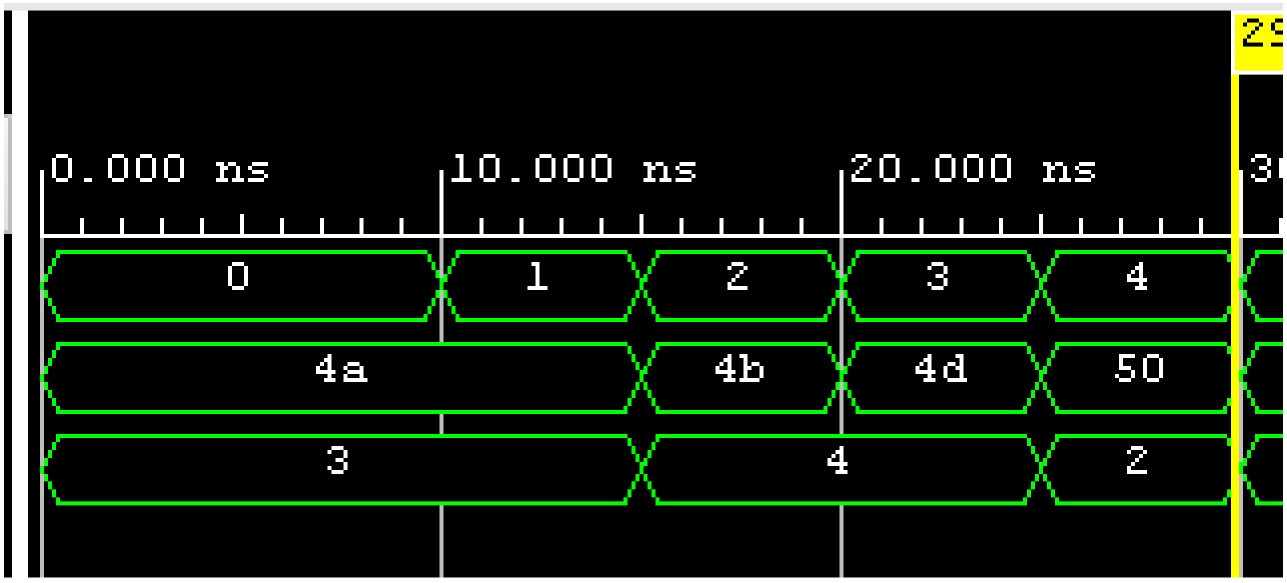
```vhdl
Entity calc_even_parity_procedure  Is Port (
    Signal ain : in STD_LOGIC_VECTOR (7 downto 0);
    Signal parity : out STD_LOGIC
);
end calc_even_parity_procedure ;

Architecture behavior of calc_even_parity_procedure  Is


    procedure calc_even_parity(
       Signal ain_int : in STD_LOGIC_VECTOR (7 downto 0); --arguments
       Signal parity_out : out STD_LOGIC --arguments
) is
        variable k : integer := 0; -- iterator
        variable par_int : STD_LOGIC := '0'; --internal parity variable
    begin
        for k in 0 to 7 loop
                par_int := par_int XOR ain_int(k); --xor parity logic
            end loop;
            parity_out <= par_int;
        end calc_even_parity;

begin

    process (ain) begin
        calc_even_parity (ain, parity);
    end process;

end behavior;
```



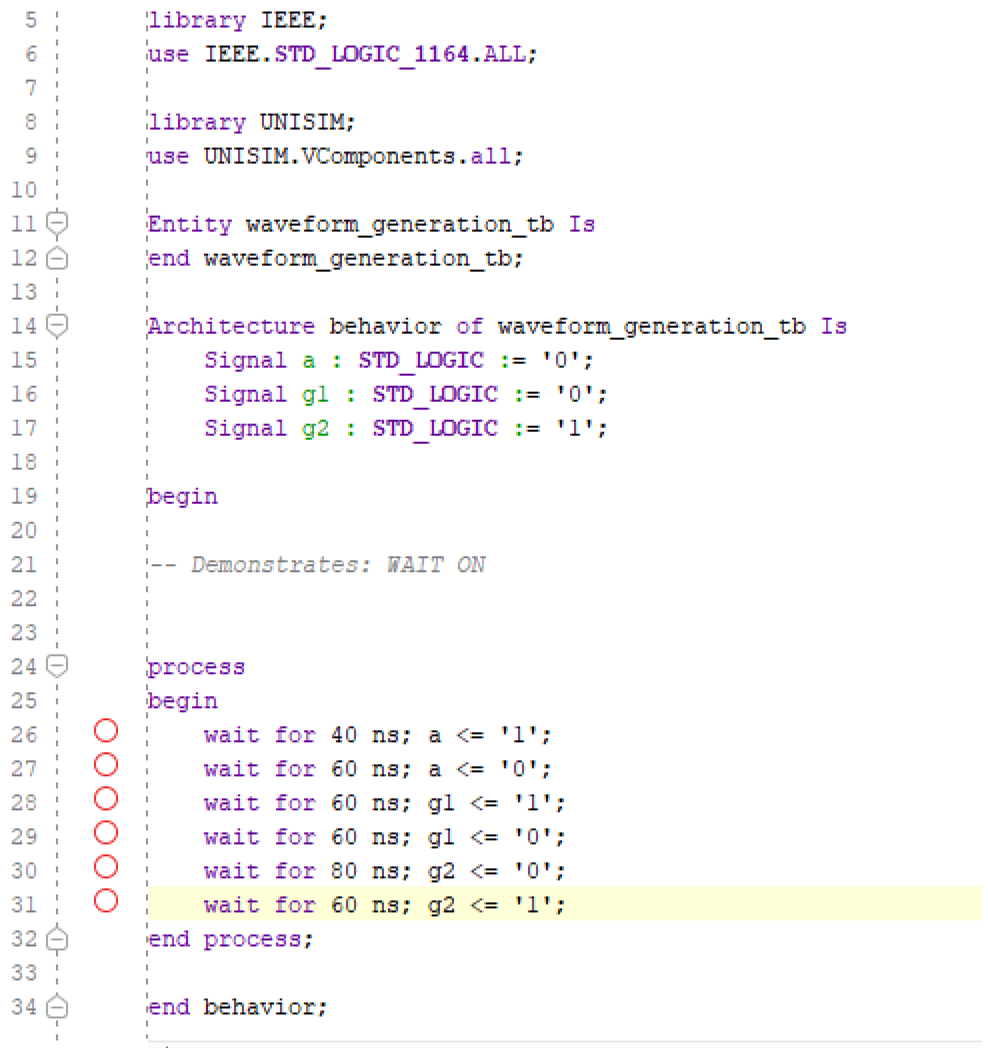
Part 2



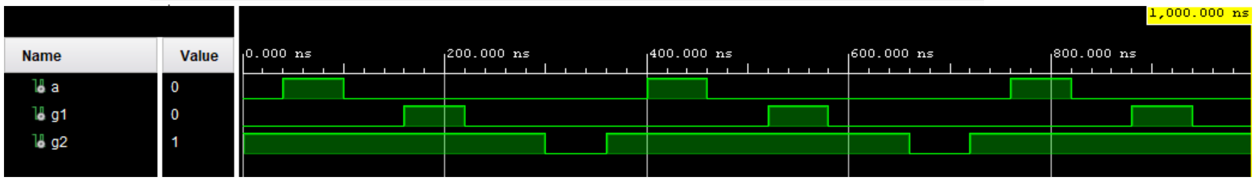
```
Entity calc_ones_function  Is Port (
    Signal ain : in STD_LOGIC_VECTOR (7 downto 0);
    Signal number_of_ones : out STD_LOGIC_VECTOR (2 downto 0)
);
end calc_ones_function ;

Architecture behavior of calc_ones_function  Is

    --signal ain_int : STD_LOGIC_VECTOR(7 downto 0);
    --signal bin_int : STD_LOGIC_VECTOR(2 downto 0);

    function calc_ones  (signal ain_int : in STD_LOGIC_VECTOR (7 downto 0))

        return std_logic_vector is
            variable result : STD_LOGIC_VECTOR(2 downto 0) := "000" ; -- result variable declared in the begi
            variable k : integer := 0; -- counter loop
        begin
            for k in 0 to 7 loop
                if ain_int(k) = '1' then
                    result := result + '1';
                end if;
            end loop;
            return result;
    end calc_ones;

begin

process (ain) begin
        number_of_ones <= calc_ones(ain);
end process;

end behavior;
```



Part 3



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VComponents.all;

Entity waveform_generation_tb Is
end waveform_generation_tb;

Architecture behavior of waveform_generation_tb Is
    Signal a : STD_LOGIC := '0';
    Signal g1 : STD_LOGIC := '0';
    Signal g2 : STD_LOGIC := '1';

begin

-- Demonstrates: WAIT ON


process
begin
    wait for 40 ns; a <= '1';
    wait for 60 ns; a <= '0';
    wait for 60 ns; g1 <= '1';
    wait for 60 ns; g1 <= '0';
    wait for 80 ns; g2 <= '0';
    wait for 60 ns; g2 <= '1';
end process;

end behavior;
```



1,000.000 ns
Name
Value
a
g1
g2
0
0
1
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns