FPGA Design
Part IV - State Machines & Sequential VHDL

Thomas Lenzi
Objective

• We learned how to use an FPGA to code combinatorial logic in VHDL.

• We will now focus on sequential logic and learn how to implement complexer functions in VHDL.

• But first we will have a look at state machines which are useful to control systems.
State Machines
State Machines

• A state machine is a construction in which a system is described by states in which it can be.

• In every state, the actions taken are clearly defined and solely depend on the state of the machine.

• The machine can go from one state to another using clearly defined conditions.

• The transition between state cannot influence the actions of the machine.
Diagrams

- State machines can be illustrated using diagrams.

State 1

- When button A is pushed
- When buttons A and B are pushed
- Always

State 2

- Otherwise
- When button B is pushed

State 3

- Otherwise
- Always

State 4
Exercise

• Draw the diagram of a fruit vending machine that only accepts 1€ and 0.50€ and returns money if the user gave too much.
Solution

Waiting for money

Inserted 1€ → Gave 1€

Inserted 0.50€ → Gave 0.50€

Less than 2€ → Nothing happens

More than 2€ → Give fruit

Exactly 2€ → Give money back

Always
Processes
Processes

• In order to code sequential statements in VHDL, you need to use *processes*.

• Statements in a process will not be executed sequentially, they will simply be interpreted that way and translated into appropriate logic by the synthesiser (do not compare this to C code).

• Processes take a *sensitivity list* as parameter. Every time one of the signals in the list changes, the process is fired.
Variables & Signals I

- When using processes, you can create *variables* at the beginning of the process.

```vhdl
process(a, b)
  variable tmp : std_logic := '0';
begin
  tmp := a and b;
  c <= tmp;
end process;
```

- The difference between *signals* and *variables* is that *signals* are concurrently assigned, while *variables* are sequentially assigned.

- The value of the *variable* is saved between the various times the process is fired.
In this example, the process will fire every time the clock changes, but the IF statement requires it to be high.

- The variable is first set to ‘0’ and then inverted. Therefore, the output will always be ‘1’.

- The signal is first set to ‘0’, but this instruction is overwritten by the following one. The signal will thus oscillate.
library ieee;
use ieee.std_logic_1164.all;

entity TripleAND is
  port(
    a : in std_logic_vector(2 downto 0);
    b : out std_logic;
    c : out std_logic
  );
end TripleAND;

architecture Behavioral of TripleAND is
  signal sig : std_logic := '1';
begin
  process(a)
  begin
    variable var : std_logic := '1';
    begin
      for I in 0 to 2 loop
        sig <= sig and a(I);
        var := var and a(I);
      end loop;
      b <= sig;
      c <= var;
    end process;
  end Behavioral;
Propagation delays and variables

• You must use variables with care as they will produce cascading logic.

• When running at low speed, the delay induced by this logic is not significant compared to the clock speed. But when using higher clock speed, you must ensure that the delay is less than the clock period.
Sequential Statements
If / elsif / else

15

process(a, b)
begin
    if (a = '1' and b = '1') then
        c <= '1';
    elsif ((a = '1' and b = '0') or (a = '0' and b = '1')) then
        c <= '0';
    else
        c <= '0';
    end if;
end process;

- The if / elsif / else statements have a similar use as in C.
- When encoding conditions, be sure to cover all possible cases!
Case

- The case statement has a similar use as in C.
- Each possibility can hold multiple sequential statements.
- When encoding conditions, be sure to cover all possible cases!
For & While

- Processes can use For and While loops to duplicate logic, similarly to the For Generate statements.

- Not every piece of code using For and While statements can be translated to logic. ISE will issue an error when this happens.

```vhdl
process(sw_i)
    variable J : integer := 0;
begin
    for I in 0 to 5 loop
        leds_o(I) <= sw_i(I);
    end loop;
    J := 0;
    while (J < 8) loop
        leds_o(J) <= sw_i(J);
        J := J + 1;
    end loop;
end process;
```
Using the Clock

• To use the clock as timer, you have to make use of the \textit{rising\_edge} function. It will ensure that the process is fired when the clock changes from low to high.

• Using this function in designs will result in the use of DFFs to buffer values.

• This is the proper way to code state machines.

```process(sw_i)
    variable led : std_logic := '0';
begin
    if (rising_edge(clk)) then
        led := not led;
        leds_o <= (others => led);
    end if;
end process;
```
Exercise

• Design a fruit vending machine that only accepts 1€ and 0.50€ (represented by two buttons) and returns money if the user gave too much. Fruits cost 2€.
Solution

process(clk)
  variable state : integer range 0 to 4 := 0;
  variable money : unsigned(7 downto 0) := (others => '0');
begin
  if rising_edge(clk) then
    -- Waiting
    if (state = 0) then
      fruit <= '0';
      toomuch <= '0';
      if (add_1 = '1') then
        state := 1;
      elsif (add_05 = '1') then
        state := 2;
      else
        state := 0;
      end if;
    end if;
    -- +1
    elsif (state = 1) then
      money := money + 100;
      if (money >= 200) then
        state := 3;
      else
        state := 0;
      end if;
    end if;
    -- +0.5
    elsif (state = 2) then
      money := money + 50;
      if (money >= 200) then
        state := 3;
      else
        state := 0;
      end if;
    end if;
    -- Give fruit
    elsif (state = 3) then
      fruit <= '1';
      if (money = 200) then
        money := 0;
        state := 0;
      elsif (state = 4) then
        state := 0;
      else
        money := 0;
        state := 0;
      end if;
    end if;
    -- Return money
  end if;
end process;
Packages
Packages

• Next to regular VHDL entities, you can also create Packages.

• Packages hold functions, procedures, custom data types, constants, etc. They are toolboxes that contain helping tools.

• To create a VHDL package, add a new file and select “VHDL Package”.
Package Structure

- Packages are decomposed in two parts: header and body.
- The header contains the custom data types, constants, and the functions and procedures headers.
- The body contains the logic of the functions and procedures.
Types and Constants

In VHDL, you can create new types:

- **subtype** is simply an alias to an existing type;
- **record** is similar to a struct in C and holds multiple signals;
- **array** defines an array of elements of a given type;
- **enumerate** defines a list of values a signal can hold.

You can also define constants that can be used in your design.
Functions

- All the parameters of a function are inputs.
- A function can only return one value.
- Statements in a function are sequential.
Procedures

library IEEE;
use IEEE.STD_LOGIC_1164.all;

package my_package is

    procedure my_procedure(
        constant var1 : in std_logic;
        signal var2 : in std_logic;
        signal var3 : out std_logic
    );

end my_package;

package body my_package is

    procedure my_procedure(
        constant var1 : in std_logic;
        signal var2 : in std_logic;
        signal var3 : out std_logic
    ) is
        variable tmp : std_logic;
    begin
        tmp := var1 and var2;
        var3 <= tmp;
    end procedure;

end my_package;

• Procedures apply logic to signals.

• The parameters of a procedure can be in/out signal/constants/variables.
Using Packages

- To use your package you need to include if using the `use` statement.
- Types can then be used for signals.
- Functions and procedures can then be called.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.my_package.all;

entity toplevel is
end toplevel;

architecture Behavioral of toplevel is

signal vec : vector_8 := (others => '0');
signal rec : my_type;
signal table : array4x8 := (others => (others => '0'));
signal enum : enumerate := VALUE_0;
signal s1, s2, s3, s4 : std_logic;
begin
  vec(1 downto 0) <= "11";
  rec.sig1 <= '0';
  rec.sig2 <= x"AB";
  table(0)(3 downto 0) <= x"F";
  enum <= VALUE_1;
  s3 <= my_function(s1, s2);
  my_procedure(s1, s2, s4);
end Behavioral;
```
Back to State Machines
A Better State Machine

- To allow for better optimisation and clarity, the state machine should respect the following rules:
  
  - The code that dictates the transitions between states and the code that acts depending on the state the machine is in should be separate.
  
  - The state variable should be an enumerate type.
architecture Behavioral of ex4 is

type states is (WAITING, ADD_1, ADD_05, CHECK, GIVE, TOO MUCH);
signal state : states := WAITING;

begin

process(clk)
begin
if (rising_edge(clk)) then
  case state is
    when WAITING =>
      if (add_1 = '1') then
        state := ADD_1;
      elsif (add_05 = '1') then
        state := ADD_05;
      else
        state := WAITING;
      end if;
    when ADD_1 | ADD_05 =>
      state := CHECK;
    when CHECK =>
      if (money = 200) then
        state := GIVE;
      elsif (money > 200) then
        state := TOO MUCH;
      else
        state := WAITING;
      end if;
    when GIVE =>
      state := WAITING;
    when TOO MUCH =>
      state := GIVE;
    when others =>
      state := WAITING;
  end case;
end if;
end process;

process(clk)
begin
if (rising_edge(clk)) then
  case state is
    when WAITING =>
      fruit <= '0';
      too much <= '0';
    when ADD_1 =>
      money := money + 100;
    when ADD_05 =>
      money := money + 50;
    when CHECK =>
      null;
    when GIVE =>
      fruit <= '1';
      money := 0;
    when TOO MUCH =>
      too much <= '1';
      money := 0;
    when others =>
      money := 0;
      fruit <= '0';
      too much <= '0';
  end case;
end if;
end process;
Simulations
Test Benches

- Test Benches are VHDL entities that drive signals into a Unit Under Test (UUT).

- They simulate components that are not in the FPGA or that are not yet implemented (clocks, communication protocols, …).

- Test benches can only be used in simulation and cannot be synthesised.
Create a Test Bench

- Go to “Project > New Source…”
- Select “VHDL Test Bench” and give it a name with a .vhd extension, then click “Next”
- On the next window, select the file for which you want to create a test bench and click “Next” and then “Finish”
ISE Simulation Environment

• To use the test bench, you need to switch to the simulation environment by selecting the Simulation view.

• In this environment, you will see that a new file is present and that it includes the entity you previously selected.

• The tools on the bottom of the screen also changed.
Structure of a Test Bench

• The logic inside a test bench uses the statements we previously reviewed, but it adds the possibility to use the *wait for* statement to wait for a given amount of time.

• In this example, the clock will be generated by the `clk_i_process` process which changes the value of the clock every 5 ns.

• Test benches recognise clock signals when their name start with *clk* and automatically create a process to generate them.

```vhdl
-- clock period definitions
castant clk_i_period : time := 10 ns;

-- instantiate the unit under test (uut)
uut: entity work.toplevel
port map (  
  clk_i => clk_i,
  reset_i => reset_i,
  a_i => a_i,
  b_i => b_i,
  c_o => c_o
);

-- clock process definitions
clk_i_process : process
begin
  clk_i <= '0';
  wait for clk_i_period/2;
  clk_i <= '1';
  wait for clk_i_period/2;
end process;

-- stimulus process
stim_proc: process
begin
  -- hold reset state for 100 ns.
  wait for 100 ns;
  wait for clk_i_period*10;
  -- insert stimulus here
  wait;
end process;
```
Code to Analyse

- The previous test bench will be used to analyse the following code.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity toplevel is
port(
    clk_i : in std_logic;
    reset_i : in std_logic;
    a_i : in std_logic;
    b_i : in std_logic;
    c_o : out std_logic
);
end toplevel;

architecture Behavioral of toplevel is
begin
    process(clk_i)
    begin
        if (rising_edge(clk_i)) then
            if (reset_i = '1') then
                c_o <= '0';
            else
                c_o <= a_i and b_i;
            end if;
        end if;
    end process;
end Behavioral;
```
Generating Signals

- To analyse our logic, we will use the following code:

```vhdl
-- stimulus process
stim_proc: process
begin
  reset_i <= '1';
  a_i <= '0';
  b_i <= '0';
  wait for 100 ns;
  reset_i <= '0';
  wait for clk_i_period * 10;
  a_i <= '1';
  wait for clk_i_period * 3;
  b_i <= '1';
  wait for clk_i_period * 3;
  a_i <= '0';
  wait for clk_i_period * 3;
  b_i <= '0';
  wait;
end process;
```

- Reset_i will be held high during 100 ns and then put low;
- a_i will then go high for 6 ns;
- b_i will be high for 6 ns;
- both a_i and b_i are put back low.

- The final wait statement without a specific duration insures that the process will never be ran again.
Running the Simulation

- To run the simulation, select the test bench in the top left menu, and double-click on “Simulate Behavioural Model”.

- A new window appears from the ISim program.

- ISim plots the input and output signals of the top level.
Simulation Result

- Using the controls to zoom out, you can observe the results of the simulation.

- As expected, \( c_o \) is high only when both \( a_i \) and \( b_i \) are high at the rising edge of \( clk_i \).
Exercise

• Re-write the code of a vending machine using types and create a test bench for it.
Best Practice VHDL
Common

• The file’s name is the entity’s name
• Use test benches whenever possible
• Use named signal mapping for entities
• When using case, when, … cover all the possible conditions
• Use constants when possible
• Avoid variables and use signals
IOs

- Only use IN and OUT modes
- Add suffixes _i or _o to inputs and outputs
- Only use std_logic, std_logic_vector, or (un)signed signals or records using them as IOs
Vectors

- Vectors always start at 0
- Vectors are always going downwards
Clocks and Resets

• Use synchronous resets

• Resets are active high

• Initialise all the signals at reset

• Use the rising edge of the clock

• Synchronous processes only have the clock in their sensitivity list
Communication Protocols
UART

• The Universal Asynchronous Receiver/Transmitter (UART) is a protocol that uses two wires: one to receive and one to transmit data.

• The clock is not transmitted thus both parties must agree on the sampling frequency of the signal (9600 Hz is the most common).
In I2C, a master controls multiple slaves by addressing them. The master and slaves use the same data line (sda) to transmit information that is synchronised to a clock line (sck). When a module is not using the lines to transmit data, they should pull it to high impedance in order to avoid shorts.
SPI

- SPI uses a Chip Select SS# bus to allow the master to select the slave it wishes to address.

- The master sends the clock SCLK and data MOSI (Master Out Slave In) to the slaves which respond using a separate wire MISO (Master In Slave Out).
Exercise
Using the 7-segments display
Exercise

• Develop a VHDL entity that will control the 7-segment display.

• It should take four 4-bit busses as input and represent their hexadecimal value on the four digits of the display.
7-segments Display

- We want to control the 7-segments display and be able to display hexadecimal characters on it.

- The only thing the user has to do is set 4 buses (one per digit) which hold the values he wants to display.
Questions

What IOs do we need to add to our design?
Is this a sequential or combinatorial design?
What inputs drive the design?
Do we need a state machine? If so, what are the states?
Step 1: IOs

- clk_50MHz_i : input clock
- reset_i : input reset
- x0_i, x1_i, x2_i, x3_i : values to print on the segments
- seg_o : output data that will be sent to the display
- dp_o : status of the point LED
- an_o : select the active segment
Step 2: Clock

- Divide the input clock from 50 MHz to 1 kHz.
- Using a counter, we generate a 1 kHz “clock”.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

architecture Behavioral of seg is

    signal counter : unsigned(15 downto 0) := (others => '0');

begin

process(clk_50MHz_i)
begin
    if (rising_edge(clk_50MHz_i)) then
        if (reset_i = '1') then
            counter <= (others => '0');
        else
            if (counter = 50_000) then
                counter <= (others => '0');
                -- Event at 1 kHz
                counter <= counter + 1;
            end if;
        end if;
    end if;
end process;

end Behavioral;
```
Step 3: States

- We will use 4 states, one for each active anode.
- The transition between states will be done every 1 ms.

```vhdl
type states is (AN0, AN1, AN2, AN3);
signal state : states := AN0;

-- Change states
process(clk_50MHz_i)
begin
  if (rising_edge(clk_50MHz_i)) then
    if (reset_i = '1') then
      counter <= (others => '0');
    else
      if (counter = 50_000) then
        counter <= (others => '0');
        case state is
          when AN0 => state <= AN1;
          when AN1 => state <= AN2;
          when AN2 => state <= AN3;
          when AN3 => state <= AN0;
          when others => state <= AN0;
        end case;
      else
        counter <= counter + 1;
      end if;
    end if;
  end if;
end process;
```
Step 4 : Data Translation

- The 7-segments display uses a data bus of 7 bits while the user uses 4 bits representing a number.

- We have to convert the number to the display.
Step 5 : Data Signalling

• Finally we need to transfer the data to the display.