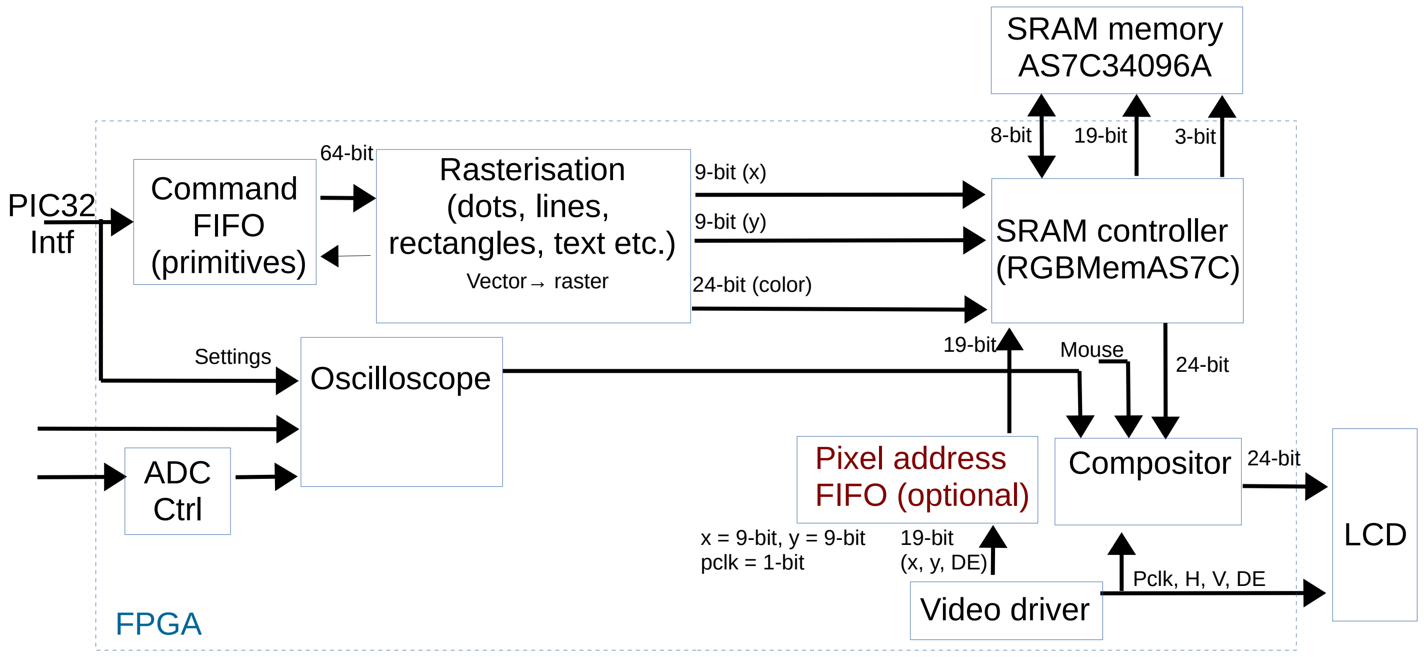
Translating VHDL To Pascal For High Level Simulation

Author VCC
First release date: 2025.05.30.

The idea of this simulation method and the simulator built around it, started as a solution to debug glitches in an FPGA design, which could not be visible by simulating the synthesized design, but were observed in hardware. It has to be mentioned, from the beginning, that the mentioned design and the presented simulator are rather old (from about 2013-2014), and although mostly functional, still require some debugging, at the time of writing this (May 2025). As the time allows, the design may be released for a detailed overview of the described simulation idea.

Board design background

The design involves a hobby development board, consisting of a Spartan-6 FPGA, connected to a TFT display, a PIC32 microconcontroller, for running basic firmware, a touch panel controller and a PIC18F microconcontroller as an interface between the touch panel controller and the PIC32 chip. The FPGA is used, in part, as a custom "video card", with an on-board video RAM, and also for custom user design. The PIC32 chip runs a firmware, which makes use of both the video part and the user part of the FPGA design. There is also a FT232H chip, for high-speed USB connectivity. This FPGA design is capable of rendering simple primitives like dots, lines, rectangles and text, as required by the DynTFT library (DynTFT repo on https://github.com/VCC02/DynTFT). For now, the board can be (partially) seen at https://www.youtube.com/watch?v=ZqpVpNuFb_s



As opposed to many microcontroller applications, which involve graphic displays, the TFT library, which usually runs on the microcontroller, is now replaced by a hardware implementation, inside the FPGA. This improves performance and response time and also saves some RAM as one of the most precious resources of a microcontroller. On the other hand, the DynTFT library, by design uses more RAM than other implementations, as a trade-off for allowing creation and destruction of graphic components (widgets) at run-time and also by having much of the GUI design encoded efficiently as binary.

The main design, intended for this board was a simple digital oscilloscope, implemented in FPGA, with a control GUI, running on the PIC32 microcontroller. This digital oscilloscope would be the user part of the FPGA design, the interchangeable part, while the graphic rendering part would be the fixed part.

The problem

Although not among the best design approaches, the development involved validating the behaviour of various (VHDL) components by simulating them, both individually and interconnected, using post-synthesis models (at that time, using Xilinx ISIM). The synthesis and implementation were intended to provide timing closure without too much manual intervention. Since these tools provided really good implementations results overall, validating the design at high-level, using post-synthesis simulation, proved to be buggy, because the simulator included timings, which did not match the implementation. This development approach was kind of tedious, also because of long simulation times and sometimes inaccurate results. Since the ISIM simulator wrongly displayed the expected behavior, but the hardware performed differently, another type of simulator was required, one which could also avoid simulating timings, and at the same time, allow a faster development approch, that would not require waiting for the simulation to run and manually check waveforms. Indeed, no test-like simulation was performed, using non-synthesizable code. Post-implementation simulation would have been more difficult, because more design components had to be connected and simulated at once.

The simulator as a solution

The VHDL simulators, although useful, were (at that time) limited to displaying waveforms of the simulated design. This is still slow from a development perspective, because it is difficult to validate the design under many different input combinations for every change of the VHDL code (regression testing) in a reasonable time. The existing simulated corner-cases remained the basis of validating the design by regression testing. To solve both the problems of finding the glitches, which were not visible in simulation and having a faster development, creating a simulator was needed, because for directly simulating VHDL code, it meant either finding an existing open-source VHDL interpreter, or writing one. None of these options were fit.

The approach was not to simulate at VHDL level or at RTL (or lower), but having a kind of direct code translation from VHDL to Pascal (as intended to be compiled with Delphi). To keep things simple, the simulator did not had to work with VHDL at all. It is the user who translates from VHDL to Pascal, in such a way, that the design still behaves as intended, by keeping the FPGA-like behavior of parallel "existence and functioning" of all parts of the design.

A code, intended to run on a (desktop) computer can fit many programming patterns, which are not typical for what was needed here. The code had to be written in such a way that, for example, no loops would be allowed (in the Pascal translation) and every simulated component would have to be run together with all the others, which is unusual for a desktop application.

Although not encountered during the modeling of this design, it is expected that not all synthesizable VHDL code would be able to be properly modeled for this approach. Since the central part of the whole simulated design, is the model itself, the simulator becomes nothing more than a wrapper over the described/implemented model of the design, together with some input and output.

Because the design is pretty complex (oscilloscope and GUI), much of it was not required to be simulated. This includes the oscilloscope part. Also, some of the remaining parts could be abstracted and implemented using available APIs for drawing on a window. This speeds up the simulation. Because all these, the simulator may have parts, which can be viewed as an emulator, so it sits somewhere betwen a simulator and an emulator.

More implementation details

One of the design decisions, involving the VHDL code, was to use both transitions of a clock signal in finite state machines. This approach has it downsides, like having two clock networks, for multiple components, or having the implementation tools work harder on timing closure. However, the reason was for easy understanding the code and following the waveforms, resulted from simulation. This may be a bad practice, but it was the desired approach.

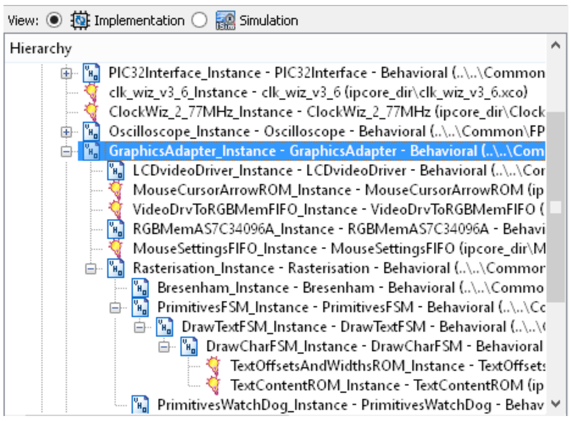
Among the multiple ways of writing a synthesizable state machine, the preferred one was to use three processes (see example in Xilinx XST user guide), one for updating the current state, one for deciding the next state and the last, for setting outputs. Updating the state is done on the rising edge of the clock, while setting outputs happens on the falling edge. Only the first and the third processes use registered signals. In this way, an output, which is updated on the falling edge of the clock, can be used to decide the next transition, on the rising edge. It may be counter-intuitive, but it avoids pipelining.

To simulate this kind of behavior, using Pascal, the model has to allow calling the same code twice, once when the clock is set to 0, then when the clock is set to 1. This has to be done for the entire design, no matter how its components are interconnected or how they work internally.

Although it might be useful in some cases to start by presenting how to model basic building blocks, this may not be a proper example of how data passes from one component to another. Because of this, parts of the mentioned design will be presented only, and how are they translated to Pascal, regardless of their complexity. The fact that the design was successfully simulated, proved that relativelty complex code can be also be simulated. As a limitation, the current implementation, never included displaying waveforms, although useful. At some point, the design complexity causes the simulation to run slower than what is comparable to the actual hardware implementation, but depending on this complexity, the simulation runs around pseudo real-time. Of course, this also depends on how fast the computer is, and if the design is / can be split into multiple threads. On the other hand, there are parts of the design, which do not need to be simulated, or are simply impractical (e.g. TFT signals).

The design

The design is a hierarchy of state machines, which receive commands for rendering primitives and move data to and from memories and to the TFT driver. The presented design includes one of the state machines, used for rendering and how it is modeled for simulation.



```
View: ● Implementation ○ Simulation
Hierarchy
  ⊞ PIC32Interface_Instance - PIC32Interface - Behavioral (..\..\Common
    clk_wiz_v3_6_Instance - clk_wiz_v3_6 (ipcore_dir\clk_wiz_v3_6.xco)
    ClockWiz_2_77MHz_Instance - ClockWiz_2_77MHz (ipcore_dir\Clock
  ⊞ Oscilloscope_Instance - Oscilloscope - Behavioral (..\..\Common\FP
  ⊟ GraphicsAdapter_Instance - GraphicsAdapter - Behavioral (..\..\Com
      LCDvideoDriver_Instance - LCDvideoDriver - Behavioral (..\..\Cor
      MouseCursorArrowROM_Instance - MouseCursorArrowROM (ip
      VideoDrvToRGBMemFIFO_Instance - VideoDrvToRGBMemFIFO (
      RGBMemAS7C34096A_Instance - RGBMemAS7C34096A - Behavi
      MouseSettingsFIFO_Instance - MouseSettingsFIFO (ipcore_dir\M
    ⊟ Rasterisation_Instance - Rasterisation - Behavioral (..\..\Commor
        Bresenham_Instance - Bresenham - Behavioral (..\..\Commo
      ⊟ PrimitivesFSM_Instance - PrimitivesFSM - Behavioral (..\..\Cc
        ⊟ DrawTextFSM_Instance - DrawTextFSM - Behavioral (..\..\
          ⊟ DrawCharFSM_Instance - DrawCharFSM - Behavioral
              TextOffsetsAndWidthsROM_Instance - TextOffsets
              TextContentROM_Instance - TextContentROM (ip
        PrimitivesWatchDog_Instance - PrimitivesWatchDog - Behav
```

Starting with one of the easiest components to understand, the *PrimitivesWatchDog* component is intended to reset the state machine from the *PrimitivesFSM* component, after a predefined time (hardcoded to 250ms), in case it doesn't stop on its own, after drawing a primitive.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity PrimitivesWatchDog is
    Port (Clk: in STD_LOGIC; --100MHz
          Reset: in STD_LOGIC;
          PrimitiveStartImpulse: in STD_LOGIC;  --connects to PrimitivesFSM.StartImpulse signal
          PrimitiveDone: in STD_LOGIC;          --connects to PrimitivesFSM.PrimitiveDone signal
          PrimitivesFSM_Reset: out STD_LOGIC;   --connects to PrimitivesFSM.Reset signal
          PrimitiveDoneFiltered: out STD_LOGIC  --connects to CommandFIFOFSM.PrimitiveDone
          );
end PrimitivesWatchDog;

architecture Behavioral of PrimitivesWatchDog is

  type TWatchDogFSM is (SInit, SWaitForPrimitiveStartImpulse, SWaitForPrimitiveDone,
                        SResetPrimitivesFSM, SWaitOneClk, SReleaseResetPrimitivesFSM,
                        SAssertPrimitiveDone, SReleasePrimitiveDone);

  signal MHzCounter: STD_LOGIC_VECTOR(6 downto 0) := CONV_STD_LOGIC_VECTOR(0, 7);
  signal MicroSecondCounter: STD_LOGIC_VECTOR(19 downto 0) := CONV_STD_LOGIC_VECTOR(0, 20);
  signal CE: STD_LOGIC := '0';
  signal CE_FromMHz: STD_LOGIC := '0'; --generated by MHzCounter
  signal CE_FromFSM: STD_LOGIC := '0'; --generated by watchdog
  signal CE_FromFSM_Reg: STD_LOGIC := '0'; --generated by watchdog - registered on rising edge
  signal ResetMicroSecondCounter: STD_LOGIC := '0';
  signal ResetMicroSecondCounterFromFSM: STD_LOGIC := '0';

  signal State: TWatchDogFSM := SInit;
  signal NextState: TWatchDogFSM := SWaitForPrimitiveStartImpulse;
begin

  CE <= CE_FromMHz and CE_FromFSM_Reg;
  ResetMicroSecondCounter <= Reset or ResetMicroSecondCounterFromFSM;

  process(Clk)
  begin
    if Clk'event and Clk = '1' then
      if Reset = '1' then
        MHzCounter <= CONV_STD_LOGIC_VECTOR(0, 7);
      else
        if MHzCounter = 99 then
          MHzCounter <= CONV_STD_LOGIC_VECTOR(0, 7);
          CE_FromMHz <= '1';
        else
          MHzCounter <= MHzCounter + 1;
          CE_FromMHz <= '0';
        end if;
      end if;

      CE_FromFSM_Reg <= CE_FromFSM;
    end if;
  end process;

  process(Clk)
  begin
    if Clk'event and Clk = '1' then
      if ResetMicroSecondCounter = '1' then
        MicroSecondCounter <= CONV_STD_LOGIC_VECTOR(0, 20);
      else
        if CE = '1' then
          MicroSecondCounter <= MicroSecondCounter + 1;
        end if;
      end if;
    end if;
  end process;

  --============================
  --FSM

  process(Clk)
  begin
    if Clk'event and Clk = '1' then
      if Reset = '1' then
        State <= SInit;
      else
        State <= NextState;
      end if;
    end if;
  end process;

  process(State, PrimitiveStartImpulse, PrimitiveDone, MicroSecondCounter)
  begin
    case State is
      when SInit =>
        NextState <= SWaitForPrimitiveStartImpulse;

      when SWaitForPrimitiveStartImpulse =>
        if PrimitiveStartImpulse = '0' then
          NextState <= SWaitForPrimitiveStartImpulse; --loop here
        else
          NextState <= SWaitForPrimitiveDone;
        end if;

      when SWaitForPrimitiveDone =>
        if PrimitiveDone = '0' then
          if MicroSecondCounter < 250000 then
            NextState <= SWaitForPrimitiveDone; --loop here
          else
            NextState <= SResetPrimitivesFSM; --force reset
          end if;
        else
          NextState <= SWaitForPrimitiveStartImpulse; --wait for another cycle
        end if;

      when SResetPrimitivesFSM =>
        NextState <= SWaitOneClk;

      when SWaitOneClk =>
        NextState <= SReleaseResetPrimitivesFSM;

      when SReleaseResetPrimitivesFSM =>
        NextState <= SAssertPrimitiveDone;

      when SAssertPrimitiveDone =>
        NextState <= SReleasePrimitiveDone;

      when SReleasePrimitiveDone =>
        NextState <= SWaitForPrimitiveStartImpulse; --new cycle, new command

      when others =>
    end case;
  end process;

  process(Clk)
  begin
    if Clk'event and Clk = '0' then
      case State is
        when SInit =>
          PrimitivesFSM_Reset <= '1';
          CE_FromFSM <= '0';
          ResetMicroSecondCounterFromFSM <= '1';
          PrimitiveDoneFiltered <= '0';

        when SWaitForPrimitiveStartImpulse =>
          PrimitivesFSM_Reset <= '0';
          ResetMicroSecondCounterFromFSM <= '1'; --keep the counter from counting
          PrimitiveDoneFiltered <= PrimitiveDone; --copy PrimitiveDone value from PrimitivesFSM

        when SWaitForPrimitiveDone =>
          CE_FromFSM <= '1';                     --enable counting
          ResetMicroSecondCounterFromFSM <= '0'; --enable counting
          PrimitiveDoneFiltered <= PrimitiveDone; --copy PrimitiveDone value from PrimitivesFSM

        when SResetPrimitivesFSM =>
```

```
          PrimitivesFSM_Reset <= '1';
          CE_FromFSM <= '0'; --stop counting

        when SWaitOneClk =>


        when SReleaseResetPrimitivesFSM =>
          PrimitivesFSM_Reset <= '0';

        when SAssertPrimitiveDone =>
          PrimitiveDoneFiltered <= '1';

        when SReleasePrimitiveDone =>
          PrimitiveDoneFiltered <= '0';

        when others =>
      end case;
    end if;
  end process;

end Behavioral;
```

The Pascal translation of above code follows the pattern of having a procedure, which initializes "signals" to their default values and a procedure which is exported. This one will be called from the *Rasterisation* unit. In the following translations, the `STD_LOGIC` type is defined as `0..1`.

```
unit EmuVHDL_PrimitivesWatchDog;

interface

uses
  Windows, SysUtils, EmuVHDL_Types;

type
  TWatchDogFSM_States = (SInit, SWaitForPrimitiveStartImpulse, SWaitForPrimitiveDone,
                         SResetPrimitivesFSM, SWaitOneClk, SReleaseResetPrimitivesFSM,
                         SAssertPrimitiveDone, SReleasePrimitiveDone);

procedure InitPrimitivesWatchDogSignals;
procedure PrimitivesWatchDog(
  Clk: STD_LOGIC; //--100MHz
  Reset: STD_LOGIC;
  PrimitiveStartImpulse: STD_LOGIC;  //--connects to PrimitivesFSM.StartImpulse signal
  PrimitiveDone: STD_LOGIC;          //--connects to PrimitivesFSM.PrimitiveDone signal
  out PrimitivesFSM_Reset: STD_LOGIC;   //--connects to PrimitivesFSM.Reset signal
  out PrimitiveDoneFiltered: STD_LOGIC  //--connects to CommandFIFOFSM.PrimitiveDone
);

implementation

type
  TVPrimitivesWatchDog = record
    MHzCounter: Byte;//STD_LOGIC_VECTOR(6 downto 0) := CONV_STD_LOGIC_VECTOR(0, 7);
    MicroSecondCounter: DWord; //STD_LOGIC_VECTOR(19 downto 0) := CONV_STD_LOGIC_VECTOR(0, 20);
    CE: STD_LOGIC;
    CE_FromMHz: STD_LOGIC; //--generated by MHzCounter
    CE_FromFSM: STD_LOGIC; //--generated by watchdog
    CE_FromFSM_Reg: STD_LOGIC; //--generated by watchdog - registered on rising edge
    ResetMicroSecondCounter: STD_LOGIC;
    ResetMicroSecondCounterFromFSM: STD_LOGIC;

    State: TWatchDogFSM_States;
    NextState: TWatchDogFSM_States;
  end;

var
  PrimitivesWatchDogSignals: TVPrimitivesWatchDog;

procedure InitPrimitivesWatchDogSignals;
begin
  PrimitivesWatchDogSignals.MHzCounter := 0;
  PrimitivesWatchDogSignals.MicroSecondCounter := 0;
  PrimitivesWatchDogSignals.CE := 0;
  PrimitivesWatchDogSignals.CE_FromMHz := 0;
  PrimitivesWatchDogSignals.CE_FromFSM := 0;
  PrimitivesWatchDogSignals.CE_FromFSM_Reg := 0;
  PrimitivesWatchDogSignals.ResetMicroSecondCounter := 0;
  PrimitivesWatchDogSignals.ResetMicroSecondCounterFromFSM := 0;
  PrimitivesWatchDogSignals.State := SInit;
  PrimitivesWatchDogSignals.NextState := SWaitForPrimitiveStartImpulse;
end;

procedure PrimitivesWatchDog(
  Clk: STD_LOGIC; //--100MHz
  Reset: STD_LOGIC;
  PrimitiveStartImpulse: STD_LOGIC;  //--connects to PrimitivesFSM.StartImpulse signal
  PrimitiveDone: STD_LOGIC;          //--connects to PrimitivesFSM.PrimitiveDone signal
  out PrimitivesFSM_Reset: STD_LOGIC;   //--connects to PrimitivesFSM.Reset signal
  out PrimitiveDoneFiltered: STD_LOGIC  //--connects to CommandFIFOFSM.PrimitiveDone
);
begin
  with PrimitivesWatchDogSignals do
  begin
    CE := CE_FromMHz and CE_FromFSM_Reg;
    ResetMicroSecondCounter := Reset or ResetMicroSecondCounterFromFSM;

    //process(Clk)
    if Clk = 0 then //if Clk'event and Clk = '1' then
    begin
      if Reset = 1 then
        MHzCounter := 0 //CONV_STD_LOGIC_VECTOR(0, 7);
      else
      begin
        if MHzCounter = 99 then
        begin
          MHzCounter := 0; //CONV_STD_LOGIC_VECTOR(0, 7);
          CE_FromMHz := 1;
        end
        else
        begin
          MHzCounter := MHzCounter + 1;
          CE_FromMHz := 0;
        end;
      end;

      CE_FromFSM_Reg := CE_FromFSM;
    end; //if Clk = 1


    //process(Clk)
    if Clk = 0 then //if Clk'event and Clk = '1' then
    begin
      if ResetMicroSecondCounter = 1 then
        MicroSecondCounter := 0 // CONV_STD_LOGIC_VECTOR(0, 20);
      else
      begin
        if CE = 1 then
          MicroSecondCounter := MicroSecondCounter + 1;
      end;
    end; //if Clk = 1


    //--===========================
    //--FSM

    //process(Clk)
    if Clk = 0 then //if Clk'event and Clk = '1' then
    begin
      if Reset = 1 then
        State := SInit
      else
        State := NextState;
    end;

    //process(State, PrimitiveStartImpulse, PrimitiveDone, MicroSecondCounter)
    case State of
      SInit:
        NextState := SWaitForPrimitiveStartImpulse;

      SWaitForPrimitiveStartImpulse:
      begin
        if PrimitiveStartImpulse = 0 then
          NextState := SWaitForPrimitiveStartImpulse //--loop here
        else
          NextState := SWaitForPrimitiveDone;
      end;

      SWaitForPrimitiveDone:
      begin
        if PrimitiveDone = 0 then
        begin
          if MicroSecondCounter < 250000 then
            NextState := SWaitForPrimitiveDone //--loop here
          else
            NextState := SResetPrimitivesFSM; //--force reset
        end
        else
          NextState := SWaitForPrimitiveStartImpulse; //--wait for another cycle
      end;

      SResetPrimitivesFSM:
        NextState := SWaitOneClk;

      SWaitOneClk:
        NextState := SReleaseResetPrimitivesFSM;

      SReleaseResetPrimitivesFSM:
        NextState := SAssertPrimitiveDone;

      SAssertPrimitiveDone:
        NextState := SReleasePrimitiveDone;

      SReleasePrimitiveDone:
        NextState := SWaitForPrimitiveStartImpulse; //--new cycle, new command
    end; //case

    //process(Clk)
    if Clk = 1 then //if Clk'event and Clk = '0' then
    begin
      case State of
        SInit:
        begin
          PrimitivesFSM_Reset := 1;
          CE_FromFSM := 0;
          ResetMicroSecondCounterFromFSM := 1;
          PrimitiveDoneFiltered := 0;
        end;

        SWaitForPrimitiveStartImpulse:
        begin
          PrimitivesFSM_Reset := 0;
          ResetMicroSecondCounterFromFSM := 1; //--keep the counter from counting
          PrimitiveDoneFiltered := PrimitiveDone; //--copy PrimitiveDone value from PrimitivesFSM
        end;

        SWaitForPrimitiveDone:
        begin
          CE_FromFSM := 1;                     //--enable counting
          ResetMicroSecondCounterFromFSM := 0; //--enable counting
          PrimitiveDoneFiltered := PrimitiveDone; //--copy PrimitiveDone value from PrimitivesFSM
        end;

        SResetPrimitivesFSM:
        begin
          PrimitivesFSM_Reset := 1;
          CE_FromFSM := 0; //--stop counting
        end;

        SWaitOneClk:
        begin
          //
        end;

        SReleaseResetPrimitivesFSM:
          PrimitivesFSM_Reset := 0;

        SAssertPrimitiveDone:
          PrimitiveDoneFiltered := 1;
          SReleasePrimitiveDone:
            PrimitiveDoneFiltered := 0;

        end; //case
      end;//if Clk = 1 then

    end; //with
end;

end.
```

Regarding the translation of how to use the clock transitions to register signals, it has to be mentioned, that the equivalent of a rising edge is represented by comparaing the Clk signal with a logic "0", and a falling edge, with a "1". This means that the value before the transition is used.
The `InitPrimitivesWatchDogSignals` procedure is called once, at the beginning of the simulation.
The `PrimitivesWatchDog` procedure is called by the `Rasterisation` procedure, from the wrapper "component" `EmuVHDL_Rasterisation`. It is called twice, once for Clk=0, then for Clk=1.

```
  PrimitivesWatchDog(
    Clk,
    Reset,
    PrimitivesFSMStartImpulse,
    RasterisationSignals_Old.PrimitiveDone,
    RasterisationSignals_New.PrimitivesFSM_Reset,
    PrimitiveDoneFiltered //out from Rasterisation
  );
```

It can be seen that two local "state" variables are used when calling `PrimitivesWatchDog` and other components. These are `RasterisationSignals_Old` and `RasterisationSignals_New`. The "`New`" one is used, where a component outputs its signals to locally declared signals. The "`Old`" is used for local inputs.
After calling all the components, the above mentioned variables are updated:

```
RasterisationSignals_Old := RasterisationSignals_New;
```

This prepares the local signal for the next iteration. The other "signals", are directly used from the list of component's parameters (e.g. `Clk`, `Reset`, `PrimitivesFSMStartImpulse` and `PrimitiveDoneFiltered`).

Possible limitations

What has not been tested, is the simulation of `inout` signals and/or open-drain busses. They would require additional variables, to carry the "drive" information of a signal, for deciding which signals drive the others.