VHDL State Machine

Booker A Robinson

January 2025

Contents

1 Introduction

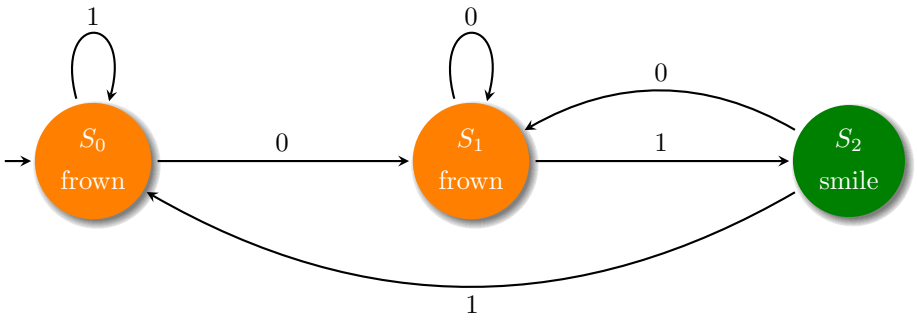
"VHSIC Hardware Description Language (VHDL) is defined. VHDL is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware."[1]

We will be using VHDL in this practice to implement the "snail brain" example with a finite state automata.

2 Snail Brain Model

A snail with 1 bit of memory crawls along a piece of paper with 0's and 1's on it. The snail smiles when it sees a 0 followed by a 1, and frowns otherwise.

This definition describes the following finite state automata.



Transition Function

State	input	Next State
S_0	0	S_1
S_0	1	S_0
S_1	0	S_1
S_1	1	S_2
S_2	0	S_1
S_2	1	S_0

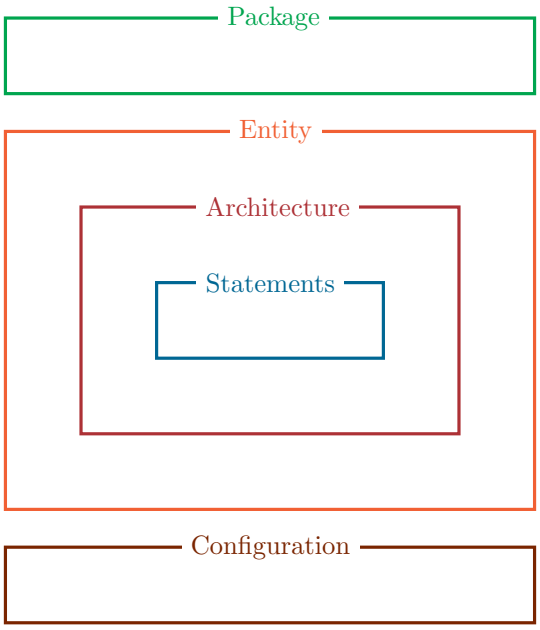
Output Function

State	output
S_0	frown
S_1	frown
S_2	smile

3 Anatomy of VHDL

3.1 Structure

The primary components of a VHDL description are called Library/Design units. They consist of Package, Entity, Architecture and Configuration declarations. Every design requires at least one Entity and Architecture declaration. Package and Configuration declarations are optional. A design may have any number of declarations.
The following is the structure of a Design unit.



3.2 Package

This optional design unit is used for making shared definitions. To use anything defined in a Package declaration, the **library** and **use** statements are needed. Something that may be in a Package declaration could be a type definition.

```
package example_package is
    type nibble is range 0 downto 3;
end example_package;
```

3.3 Entity

Entities define the I/O of a design. The entity behaves like a block symbol in a schematic. An entity has what are known as Ports, these are analogous to the

pins on a schematic symbol. All I/O for an entity flows through the ports.
Each Port must have one of 4 direction modes; **in**, **out**, **inout** or **buffer**. Mode **buffer** is equivalent to mode **out** except that a **buffer** port may be read within the Entity.
An Entity may also contain Generics. Unlike Ports, Generics are used to pass static information. They are often used to configure the behaviour of different instances of an Entity.

```
library example_lib;
use example_lib.example_package.all;

entity example_entity is
    generic(logic_size: nibble);
    port(
       a: in nibble;
       b: out nibble;
       c: inout nibble;
       d: buffer nibble
    );
end example_entity;
```

3.4 Architecture

The architecture is the actual description of the design. The architecture describes what is inside the functional block. An architecture can contain both concurrent and sequential statements. VHDL allows you to have more than one architecture for the same entity.
An architecture consists of two pieces: the architecture declaration section and the architecture body. The declaration section is where you declare objects that are local to your architecture, and the body section is where you specify the behavior of the architecture.

```
architecture behavioral of example_entity is
-- Declaration
begin
-- Body
end architecture;
```

3.5 Statements

3.5.1 Concurrent Statements

Concurrent statements are placed in the body of the architecture. Such statements include: signal connections, combinational logic, and process statements.

3.5.2 Sequential Statements

Despite the process statement itself being a concurrent statement, the body of a process contains sequential statements. Sequential statements also appear in

procedure and function bodies.

3.6 Configuration

Configuration declarations may be used to associate particular design entities to component instances (unique references to lower-level components) in a hierarchical design, or associate a particular architecture entity. As their name implies, configuration declarations are used to provide configuration management and project organization for a large design.

4 Describing the Snail

First we will look at the overall structure of the synchronous logic.

```
-- Useful type definitions using enums to reprasent the states
-- and the outputs.
package States is
    type MachineState is (S0, S1, S2); -- Same states as in
    diagrams above.
    type Emotion is (Frown, Smile); -- Output types.
end package;


use work.States.all; -- use States types in this file.
library ieee;
use ieee.std_logic_1164.all; -- use std_logic type in this file.
entity StateMachine is
    port(
        Clk: in std_logic;
        Rst: in std_logic; -- Positive reset
        input: in bit;
        output: out Emotion
    );
end StateMachine;

architecture rtl of StateMachine is
    signal currentState: MachineState;
    -- transitionFunction procedure definition will go here.
begin
    -- (Clk) sensitivity list makes the process only run on clock
    rising and falling edges.
    process(Clk) begin
        -- For synchronous logic we will only perform
        -- state transitions on the clock rising edge.
        if(rising_edge(Clk)) then
            if Rst = '1' then -- Positive reset
                currentState <= S0; -- S0 is the start state.
            else
                -- TODO (Implement transitionFunction procedure)
                transitionFunction(currentState, input);
            end if;
        end if;
    end process;
```

```
    -- (currentState) sensitivity list because since this is a
    -- Moore machine we must only change the output on
    -- state transitions.
    process(currentState) begin
        if currentState = S2 then
            output <= Smile;
        else
            output <= Frown;
        end if;
    end process;

end rtl;
```

The transition function in this case will actually be implemented as a procedure. The currentState signal is inout because we both read and write to it.

```
    procedure transitionFunction(
        signal currentState: inout MachineState;
        signal input: in bit
    ) is
    begin
              case currentState is
                        when S0 =>
                            case input is
                                when '0' => currentState <= S1;
                                when '1' => currentState <= S0;
                            end case;
                        when S1 =>
                            case input is
                                when '0' => currentState <= S1;
                                when '1' => currentState <= S2;
                            end case;
                        when S2 =>
                            case input is
                                when '0' => currentState <= S1;
                                when '1' => currentState <= S0;
                            end case;
                    end case;
    end procedure;
```

5 Simulating the Snail

5.1 Test Bench Code

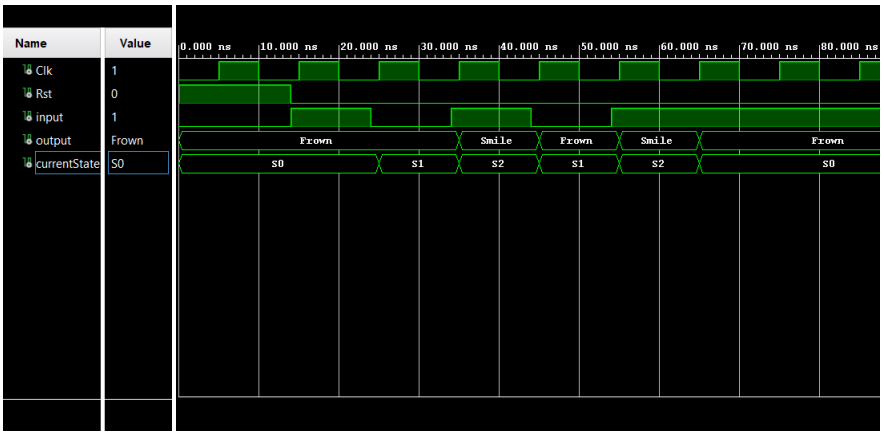
This code allows us to simulate inputs to our state machine so that we can evaluate it waveforms.

```
entity TB is
end TB;

use work.States.all;
```

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

architecture Behavioral of TB is
    signal Clk: std_logic := '0';
    signal Rst: std_logic;
    signal input: bit;
    signal output: Emotion;
begin

    -- Create an instance of the state machine and map ports.
    i_StateMachine: entity work.StateMachine(rtl)
    port map(
        Clk => Clk,
        Rst => Rst,
        input => input,
        output => output
    );

    Clk <= not Clk after 5ns; -- Simulate clock with 10ns period.

    process begin
        Rst <= '1'; -- Reset to initial state (S0)
        wait for 14ns; -- Unset reset 1ns before clock edge to
    start machine.
        Rst <= '0';
        input <= '1';
        wait for 10 ns;
        input <= '0';
        wait for 10ns;
        input <= '1'; -- rising edge
        wait for 10 ns;
        input <= '0';
        wait for 10ns;
        input <= '1'; -- rising edge
        wait;
    end process;

end Behavioral;
```

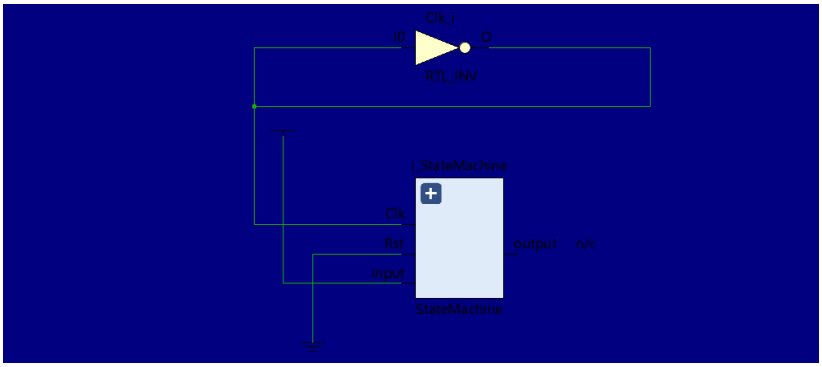
5.2 Waveforms

Here we can see the logic at work. Evaluating the wave form with the test bench code we can see that this machine is actually a **positive edge detector**.

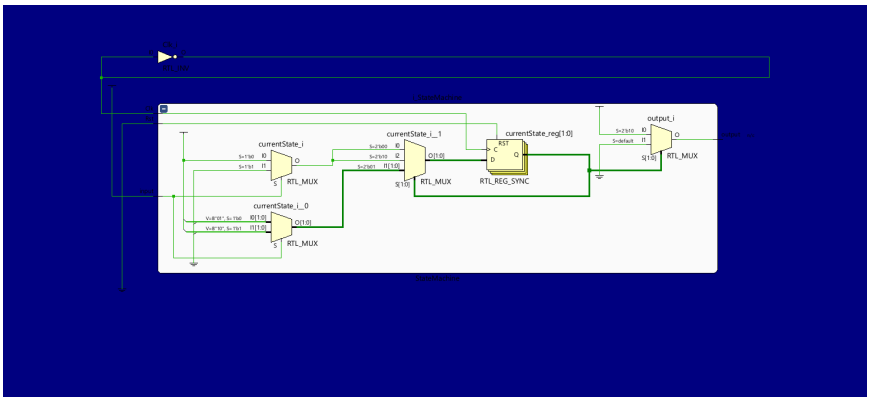


6 Understanding the Schematic

High level we see the test bench clock implementation with an or gate feedback, then we see the state machine block.

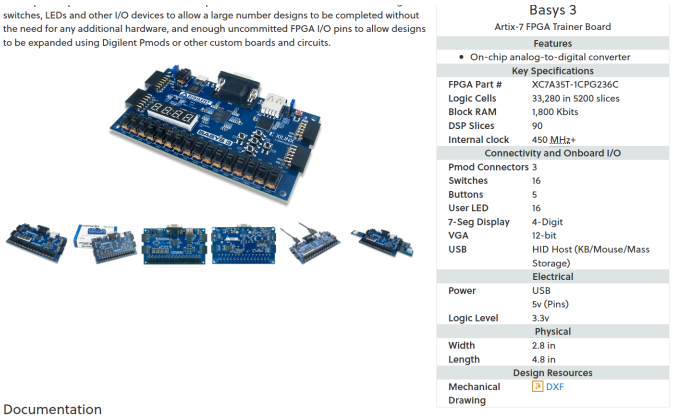


In the more detailed overview we can see the collection of Mux's that implement the case statement in the transition function (**left of the registers**). We can also see the Mux implementing the output logic (**right of the registers**). The registers in the center hold the current state of the machine.



7 Bringing it all together

Here we finish by running this all on real hardware. For this part we will be using an Artix 7 FPGA (basys 3).
`https://digilent.com/reference/programmable-logic/basys-3/start`



switches, LEDs and other I/O devices to allow a large number designs to be completed without the need for any additional hardware, and enough uncommitted FPGA I/O pins to allow designs to be expanded using Digilent Pmods or other custom boards and circuits.

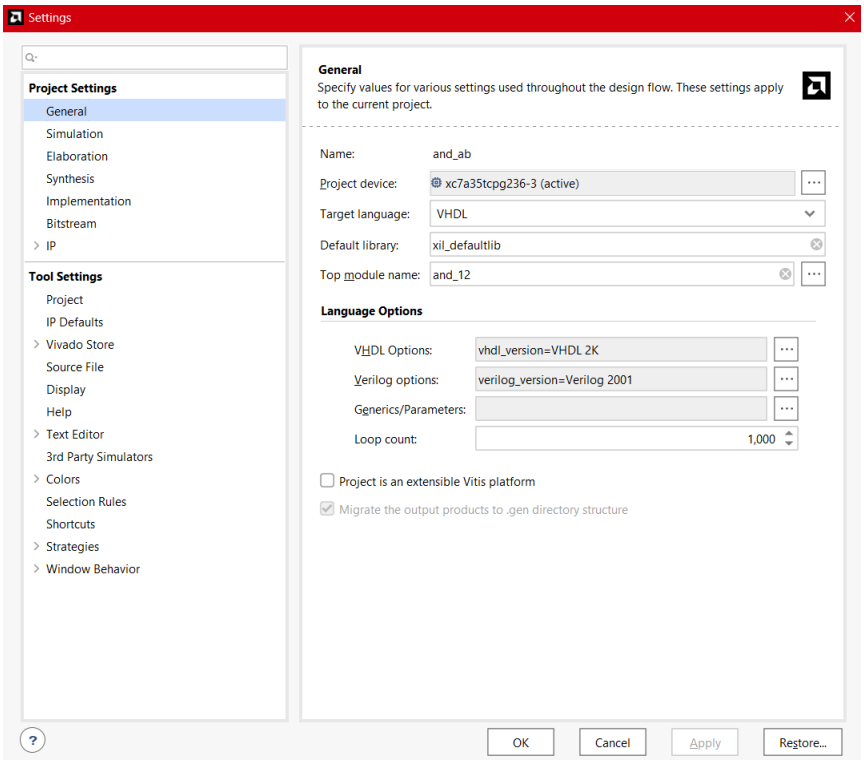
Documentation

Basys 3

Artix-7 FPGA Trainer Board

Features	
• On-chip analog-to-digital converter	
Key Specifications	
FPGA Part #	XC7A35T-1CPG236C
Logic Cells	33,280 in 5200 slices
Block RAM	1,800 Kbits
DSP Slices	90
Internal clock	450 MHz+
Connectivity and Onboard I/O	
Pmod Connectors	3
Switches	16
Buttons	5
User LED	16
7-Seg Display	4-Digit
VGA	12-bit
USB	HID Host (KB/Mouse/Mass Storage)
Electrical	
Power	USB 5v (Pins)
Logic Level	3.3v
Physical	
Width	2.8 in
Length	4.8 in
Design Resources	
Mechanical Drawing	DXF

The Part used for this board is the one shown for Project Device in the following.
xc7a35tcpg236-3



7.1 Bringing the snail to life

Here we are doing three things; defining the interface, re-mapping hardware incompatible types, and most importantly implementing a clock divider.

```
library ieee;
use ieee.numeric_std.all;
package num is
    -- The purpose of this subtype is to count 100e6 clockcycles.
    This is useful because the overflow logic for the clock divider
     is built into the type.
    subtype short is integer range 0 to 100e6; -- Defines ~0.5Hz
    clock divider overflow.
end package;
use work.num.all;

library ieee;
use ieee.std_logic_1164.all;
entity Snail is
    port(
        Clk: in std_logic;
        Rst: in std_logic;
        input: in std_logic; -- State Machine input led
        output: out std_logic; -- State Machine output led
        led: out std_logic; -- Testing led
        led2: out std_logic; -- Clock output led
```

```
        btn: in std_logic -- Testing button
    );
end Snail;

use work.States.all;


architecture Behavioral of Snail is
      signal em: Emotion;
      signal inp: bit;
      signal divider: short;
      signal dividedClock: std_logic;
begin

    i_StateMachine: entity work.StateMachine(rtl)
    port map(
        Clk => dividedClock,
        Rst => Rst,
        input => inp,
        output => em
    );

    led <= dividedClock; -- 0.5Hz led blink.
    led2 <= btn -- Button to led signal assignment for testing.

    -- Manual cast from std_logic to bit.
    process(input) begin
        if input = '1' then
            inp <= '1';
        else
            inp <= '0';
        end if;
    end process;

    -- Manual cast from Emotion to std_logic.
    process(em) begin
        if em = Smile then
            output <= '1';
        else
            output <= '0';
        end if;
    end process;

    -- Clock divider with 100e6 overflow divider.
    process(Clk) begin
        if rising_edge(Clk) then
            divider <= divider + 1; -- Adding 1 until 100e6
            if divider = 0 then -- Overflows every 1 second.
                dividedClock <= not dividedClock;
            end if;
        end if;
    end process;

end Behavioral;
```

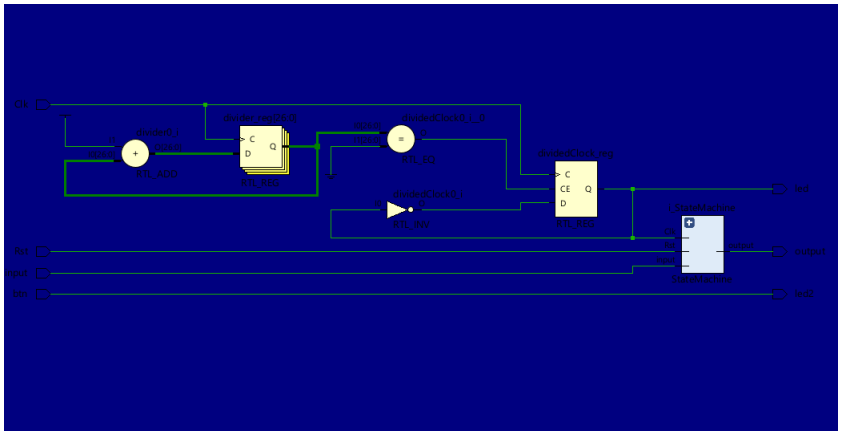
7.2 Hardware Configuration

The hardware configuration is stored in a **.xcd** file. Path: **_.src/constrs_1/new/xdc.xdc**
This is used to set up the clock at 100MHz and the package IO of the board.
The general XDC for the Basys3 rev B board can be found here:
`https://github.com/Digilent/digilent-xdc/blob/master/Basys-3-Master.xdc`

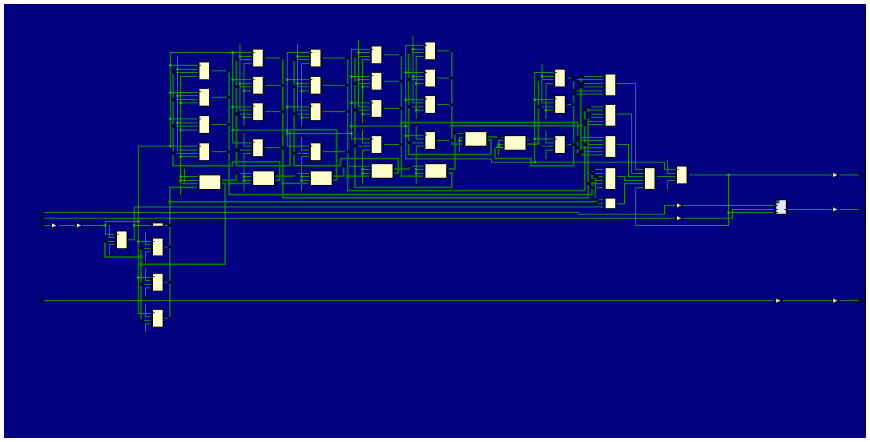
```
set_property PACKAGE_PIN U18 [get_ports input]
set_property PACKAGE_PIN U16 [get_ports output]
set_property PACKAGE_PIN U17 [get_ports Rst]
set_property PACKAGE_PIN W5 [get_ports Clk]
set_property IOSTANDARD LVCMOS33 [get_ports Clk]
create_clock -period 10.000 -name sys_clk_pin -waveform {0.000
    5.000} -add [get_ports Clk]
set_property IOSTANDARD LVCMOS33 [get_ports input]
set_property IOSTANDARD LVCMOS33 [get_ports output]
set_property IOSTANDARD LVCMOS33 [get_ports Rst]

set_property PACKAGE_PIN L1 [get_ports led]
set_property IOSTANDARD LVCMOS33 [get_ports led]

set_property PACKAGE_PIN W19 [get_ports btn]
set_property IOSTANDARD LVCMOS33 [get_ports btn]
set_property PACKAGE_PIN U3 [get_ports led2]
set_property IOSTANDARD LVCMOS33 [get_ports led2]
```

7.3 Stages

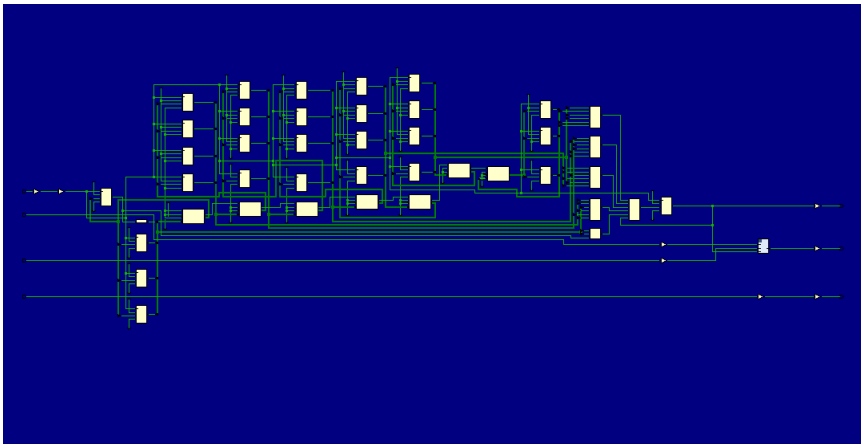
7.3.1 RTL Analysis



7.3.2 Synthesis



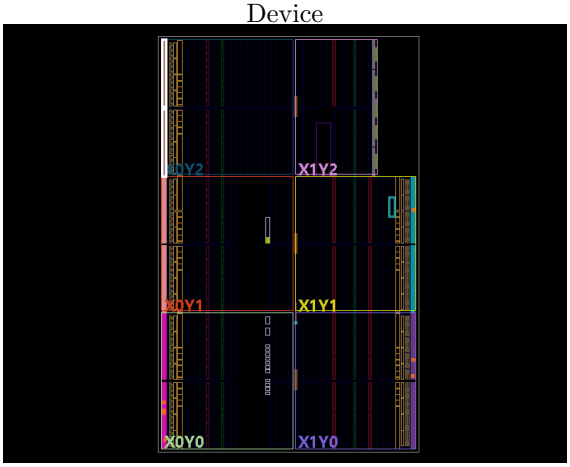
7.3.3 Implementation



7.4 Package and Device

Package

Device



8 Conclusion

In summary we have described and implemented the Snail state machine in an FPGA. Important things we have learned include; the basics of VHDL, Writing a State Machine, creating a test bench to program simulations, creating a clock divider for real hardware, and adding a hardware configuration to a project. The last step would be to upload this and run it on the Basys 3. Tutorial: `https://digilent.com/reference/learn/programmable-logic/tutorials/basys-3-programming-guide/start`

9 References

[1] `https://edg.uchicago.edu/~tang/VHDLref.pdf`
[2] `https://staff.fysik.su.se/~silver/digsyst/vhdl_ref.pdf`
[3] `https://digilent.com/reference/programmable-logic/basys-3/start`