

---

# **hVHDL**

***Release 0.1***

**johonkanen**

**Oct 02, 2023**



# ABOUT

<b>1</b>	<b>What is hVHDL</b>	<b>3</b>
<b>2</b>	<b>Usage</b>	<b>5</b>
2.1	Using the sources in your project . . . . .	5
<b>3</b>	<b>Module Repository Structure</b>	<b>7</b>
<b>4</b>	<b>example project of high level vhdl</b>	<b>9</b>
4.1	Example Project . . . . .	9
4.2	Noisy sine generation . . . . .	11
4.3	Filter interface . . . . .	12
4.4	Floating point filter implementation . . . . .	13
4.5	Fixed point filter implementation . . . . .	15
4.6	Filtering results . . . . .	16
4.7	Synthesis results . . . . .	20
<b>5</b>	<b>Fixed point math</b>	<b>25</b>
5.1	Multiplier . . . . .	25
5.2	Divider . . . . .	26
5.3	Sine and Cosine . . . . .	27
5.4	synchronous coordinate transforms . . . . .	28
<b>6</b>	<b>floating point math</b>	<b>29</b>
6.1	Adder/subtractor . . . . .	29
<b>7</b>	<b>FPGA Memory</b>	<b>31</b>
<b>8</b>	<b>FPGA interconnect</b>	<b>33</b>
<b>9</b>	<b>Dynamic verification module</b>	<b>35</b>
<b>10</b>	<b>Gigabit ethernet</b>	<b>37</b>
<b>11</b>	<b>Uart</b>	<b>39</b>
<b>12</b>	<b>Create High Level Interfaces in VHDL</b>	<b>41</b>
12.1	Records as abstract data types in VHDL . . . . .	41
12.2	Subroutines . . . . .	41
12.3	Using a simple interface . . . . .	42
<b>13</b>	<b>Using libraries and packages to abstract VHDL</b>	<b>43</b>
13.1	The specialty of use work.package . . . . .	43

<b>14 Automatic code timing with handshake interfaces</b>	<b>45</b>
14.1 Backpressure instead of timing . . . . .	45
14.2 Adding backpressure to sequential / blocking function . . . . .	45
14.3 Adding backpressure to pipelined function . . . . .	45
14.4 Adding ability to do backpressure to a IP module . . . . .	46
<b>15 Real numbers in synthesizable VHDL</b>	<b>47</b>
15.1 Real numbers as a form of abstract interfaces . . . . .	47
<b>16 High Level Coding Patterns in synthesizable VHDL</b>	<b>49</b>
16.1 Creating callable object with records and subroutines . . . . .	49
16.2 Creating objects from objects . . . . .	49
16.3 Simple state machines . . . . .	49
<b>17 Reusing VHDL source code</b>	<b>51</b>
17.1 Packages . . . . .	51
<b>18 Sharing hardware resources</b>	<b>53</b>
18.1 Sharing hw resouces inside a procedure . . . . .	53
18.2 Sharing hardware accross processes with a bus pattern . . . . .	53
<b>19 Solving differential equations on FPGA</b>	<b>55</b>
19.1 Numerical integration . . . . .	55

---

**Note:** This project is under active development.

---

Listing 1: High level code example

```
if ethernet_rx_is_active(ethernet_rx_ddio_data_out) then
    capture_ethernet_frame(ethernet_rx, ethernet_rx_ddio_data_out);
end if;
```

**hVHDL** is a set of coding patterns for standard VHDL that are designed manage complexity of digital system design in VHDL. Complexity is managed by dividing large systems into small individual pieces that can be designed, tested and updated in isolation from each other. The coding patterns allow us to support incremental design, testing and development of the VHDL source code and to increase the level of abstraction. All code has been tested with an FPGA using Xilinx Vivado, ISE, Intel Quartus or Efinix Efinity tools and simulated with GHDL.

<https://github.com/hVHDL>



## **WHAT IS HVHDL**

**hVHDL** is a set of coding patterns for standard VHDL that are designed manage complexity of digital system design in VHDL. Complexity is managed by dividing large systems into small individual pieces that can be designed, tested and updated in isolation from each other. The coding patterns allow us to support incremental design, testing and development of the VHDL source code and to increase the level of abstraction. All code has been tested with an FPGA using Xilinx Vivado, Lattice Diamond,, Intel Quartus or Efinix Efinity tools and simulated with GHDL.





## USAGE

The repositories are developed as independently as possible as long as there is no need to repeat code. To add a repository to your project just add them as a submodule. For example adding the fixed point math library as submodule into your project through console

```
git submodule add -b main https://github.com/hVHDL/hVHDL_math_library.git --init --  
↪remote --recursive
```

After this you can run the VUnit script to run all testbenches found in the module and save the simulations into gtkwave wave format using

```
python vunit_run.py -p 8 --gtkwave-fmt ghw
```

### 2.1 Using the sources in your project

After the module is added to your project, you need to add the sources you need into your project and optionally specify to which VHDL library the sources are added. All references to packages in the repositories are made without specifying the library using *work.package.all* so there is no need to have a special library for the code.



## MODULE REPOSITORY STRUCTURE

Repository structure with the modules follow the following pattern. The interfaces through which the modules are used will be in interface/ folder. This folder has test benches with functional tests for the interfaces. These interface tests are the specification for the behavior of the interfaces. The tests are very vague and lacking details by design as we want to have the possibility for changing any part of the code behind the interfaces. These tests are the only ones that are not ment to changed.

Since all modules are ment to be used as submodules, these interface tests allow us to use any version of the code as long as we use the code through these interfaces. These interfaces also allow us to develop the code in the submodules independently of the application code where they are used. This separation of module from the application is very important to allow code being reused and still retain the ability to modify it.

In VHDL abstract interfaces are created using functions and procedures as well as a record. In order to change the specifics of the code we can use records in records. The way we introduce modifiability to the insides of these records and subroutines we use packages and libraries. These are further explained in the page about *Create High Level Interfaces in VHDL*.



## EXAMPLE PROJECT OF HIGH LEVEL VHDL

There is an example project that includes build scripts for building the project using Efinix Efinity, Lattice Diamond, Intel Quartus and Xilinx Vivado. The example project is tested with FPGA. The build scripts allow a single command to build the project with any of the tools.

The example project uses uart, multiplier, floating point math and the internal bus. The design creates a noisy sine wave and a fixed and a floating point filters to clean the noise from the sine. The sine, noise, and fixed and floating point filtered versions of the noisy sine are then connected to an internal bus. The bus is connected to an uart which allows communication between the FPGA and a PC. The communication allows streaming the register pointed by a number that is obtained from the UART, thus any register connected to the bus is readable from the uart with a PC.

See the example project here : [https://github.com/hVHDL/hVHDL\\_example\\_project](https://github.com/hVHDL/hVHDL_example_project)

### 4.1 Example Project

The project top module is called 'top'. Intel, Lattice and Xilinx tools use similar top files that instantiate the main clocks and connects signals from project into the physical IO signals in the top file. With Efinix tools the PLL is managed by the build system, thus Efinix tools manage this PLL layer of the design. The efinix\_top.vhd file is common for all builds which reduces the need to manage multiple different versions of the project main IO routing layer.

Listing 1: Diamond, Quartus and Vivado top file

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

entity top is
    port (
        clk      : in std_logic ;
        uart_rx  : in std_logic ;
        uart_tx  : out std_logic
    );
end entity top;

architecture rtl of top is

-----

    component main_clock is
        port (
            CLKI : in std_logic;
```

(continues on next page)

(continued from previous page)

```

        CLKOP: out std_logic);
end component;

-----

signal clock_120mhz : std_logic := '0';

-----

begin

-----

    u_main_clocks : main_clock
    port map(clk, clock_120mhz);

-----

    u_hvhd_example : entity work.efinix_top
    port map(
        clock_120mhz => clock_120mhz,
        uart_rx      => uart_rx,
        uart_tx      => uart_tx);

-----

end rtl;

```

The efinix top module instantiates the main system interconnect module which instantiates the main design modules and connects them together. The first interesting design feature is visible in the IO names of the routed uart IO signal. The reason for the long names of the uart sources is that the IO are routed through the design using records. The use of records allows syntax checking to catch signal routing bugs in the design. If we change a module with IO, then the syntax checking will see that the name of the module is changed and flags an error if we did not remember to change the signals accordingly.

Listing 2: Efinix top

```

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

entity efinix_top is
    port (
        clock_120mhz : in std_logic;
        uart_rx      : in std_logic;
        uart_tx      : out std_logic
    );
end entity efinix_top;

architecture rtl of efinix_top is

begin

-----

    u_hvhd_example : entity work.hvhd_example_interconnect
    port map(
        system_clock => clock_120mhz,
        hvhd_example_interconnect_FPGA_in.communications_FPGA_in.uart_FPGA_in.uart_

```

(continues on next page)

(continued from previous page)

```

↪transreceiver_FPGA_in.uart_rx_fpga_in.uart_rx      => uart_rx,
    hvhdl_example_interconnect_FPGA_out.communications_FPGA_out.uart_FPGA_out.uart_
↪transreceiver_FPGA_out.uart_tx_fpga_out.uart_tx => uart_tx);

end rtl;

```

## 4.2 Noisy sine generation

The High level coding patterns are visible in the sine generation. The process that creates a sine, noise and 100kHz time level for the signal and filtering is shown below. In the process we create a multiplier, a sincos module that calculates both sine and cosine of an angle and initialize the internal bus node as well as the fixed and floating point filter modules. The full source code can be found here [https://github.com/hVHDL/hVHDL\\_example\\_project/blob/main/source/hvhdl\\_example\\_interconnect/hvhdl\\_example\\_interconnect\\_pkg.vhd](https://github.com/hVHDL/hVHDL_example_project/blob/main/source/hvhdl_example_interconnect/hvhdl_example_interconnect_pkg.vhd)

The process creates a down counter, that counts from 1199 to zero. This corresponds to 100kHz time level at 120MHz system clock. When the counter is zero, the sine calculation is requested and both noise and angle are updated. When sine calculation is ready as indicated by the sincos\_is\_ready function call returning true, the fixed and floating point filters are requested.

Listing 3: sine and timelevel generation

```

create_noisy_sine : process(system_clock)
begin
    if rising_edge(system_clock) then
        create_multiplier(multiplier);
        create_sincos(multiplier , sincos);

        init_example_filter(floating_point_filter_in);
        init_example_filter(fixed_point_filter_in);

        init_bus(bus_from_interconnect);
        connect_read_only_data_to_address(bus_from_master, bus_from_interconnect, 100,
↪get_sine(sincos)/2 + 32768);
        connect_read_only_data_to_address(bus_from_master, bus_from_interconnect, 101,
↪angle);
        connect_read_only_data_to_address(bus_from_master, bus_from_interconnect, 102,
↪to_integer(signed(prbs7))+32768);
        connect_read_only_data_to_address(bus_from_master, bus_from_interconnect, 103,
↪sine_with_noise/2 + 32768);

        if i > 0 then
            i <= (i - 1);
        else
            i <= 1199;
        end if;

        if i = 0 then
            request_sincos(sincos, angle);
            angle    <= (angle + 10) mod 2**16;
            prbs7    <= prbs7(5 downto 0) & prbs7(6);
            prbs7(6) <= prbs7(5) xor prbs7(0);

```

(continues on next page)

(continued from previous page)

```

    end if;

    if sincos_is_ready(sincos) then
        sine_with_noise <= get_sine(sincos) + to_integer(signed(prbs7)*64);
        request_example_filter(floating_point_filter_in, sine_with_noise);
        request_example_filter(fixed_point_filter_in, sine_with_noise);
    end if;

    end if; --rising_edge
end process testi;

-----

u_floating_point_filter : entity work.example_filter_entity(float)
    generic map(filter_time_constant => filter_time_constant)
    port map(system_clock, floating_point_filter_in, bus_from_master, bus_from_
↳floating_point_filter);

-----

u_fixed_point_filter : entity work.example_filter_entity(fixed_point)
    generic map(filter_time_constant => filter_time_constant)
    port map(system_clock, fixed_point_filter_in, bus_from_master, bus_from_fixed_
↳point_filter);

-----

```

Sine, noise, angle and noisy sine are all connected to the internal bus. This is done with a procedure call to connect\_read\_only\_data\_to\_address. The arguments are the in and out directional bus, address to which the data is readable from and then the data which is connected to the bus. The bus is made to be 16 bit wide, making that the numbers are between 0 and 65535, thus 32768 is added to the signals in order to fit negative numbers into the 16 bit number range.

### 4.3 Filter interface

The filters have a common entity. Since both the fixed and floating point filter take in the noisy sine and then connect the filtered sine into the internal bus, we use a common source file for both. The example\_filter\_entity.vhd has a package that defines a record and two interface subroutines. These subroutines are init\_filter\_example and request\_filter\_example, which allows an abstract interface to the module. With the use of this record, the module which uses the filter does not need to have accurate description on how the filterin is actually used. This allows also changing the interface if this is needed.

Listing 4: filter interface functions and entity description

```

package example_filter_entity_pkg is

    type example_filter_input_record is record
        filter_is_requested : boolean;
        filter_input        : integer;
    end record;

    constant init_example_filter_input : example_filter_input_record := (false, 0);

```

(continues on next page)



(continued from previous page)

```

-----
procedure init_example_filter (
    signal example_filter_input : out example_filter_input_record);
-----

procedure request_example_filter (
    signal example_filter_input : out example_filter_input_record;
    data : in integer);
-----

end package example_filter_entity_pkg;
-----

-----

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

    use work.fpga_interconnect_pkg.all;

    use work.example_filter_entity_pkg.all;

entity example_filter_entity is
    generic(filter_time_constant : real);
    port (
        clock : in std_logic;
        example_filter_input : in example_filter_input_record;
        bus_in          : in fpga_interconnect_record;
        bus_out         : out fpga_interconnect_record
    );
end entity example_filter_entity;

```

## 4.4 Floating point filter implementation

The actual filter implementation is in the two alternate architectures of the filter interface. Since the noisy sine is calculated using fixed point, the floating point filter first transforms the integer to floating point number, then filters the signal and then transforms the result back to fixed point and lastly connects the result to the internal bus.

The floating point module is accessed with call to the create\_float\_alu, create\_denormalizer and create normalizer procedures with associated signals as arguments. The float alu implements floating point add and subtract functions and a floating point multiplier. The created normalizer and denormalizer are used for transforming between integer and floating point numbers. The actual filter calculation calculates a first order filter using  $y \leq y + (u - y) * \text{filter\_gain}$ .

Listing 5: floating point filter implementation

```

floating_point_filter : process(clock)
begin
    if rising_edge(clock) then
        init_bus(bus_out);
        connect_read_only_data_to_address(bus_in, bus_out, 106, get_mantissa(get_
↪filter_output(float_filter)));

```

(continues on next page)

(continued from previous page)

```

        connect_read_only_data_to_address(bus_in, bus_out, 107, get_exponent(get_
↪filter_output(float_filter)));
        connect_read_only_data_to_address(bus_in, bus_out, 108, get_
↪integer(denormalizer) + 32768);

        create_float_alu(float_alu);
        create_denormalizer(denormalizer);
        create_normalizer(normalizer);

        if example_filter_input.filter_is_requested then
            to_float(normalizer, example_filter_input.filter_input, 15);
        end if;

        if normalizer_is_ready(normalizer) then
            request_float_filter(float_filter, get_normalizer_result(normalizer));
        end if;

        request_scaling(denormalizer, get_filter_output(float_filter), 14);

-----

        filter_is_ready <= false;
        CASE filter_counter is
            WHEN 0 =>
                subtract(float_alu, u, y);
                filter_counter <= filter_counter + 1;
            WHEN 1 =>
                if add_is_ready(float_alu) then
                    multiply(float_alu , get_add_result(float_alu) , filter_gain);
                    filter_counter <= filter_counter + 1;
                end if;

            WHEN 2 =>
                if multiplier_is_ready(float_alu) then
                    add(float_alu, get_multiplier_result(float_alu), y);
                    filter_counter <= filter_counter + 1;
                end if;
            WHEN 3 =>
                if add_is_ready(float_alu) then
                    y <= get_add_result(float_alu);
                    filter_counter <= filter_counter + 1;
                    filter_is_ready <= true;
                end if;
            WHEN others => -- wait for start
        end CASE;

-----

        end if; --rising_edge
    end process floating_point_filter;

```

## 4.5 Fixed point filter implementation

The fixed point filter is similarly implemented in its own architecture. The fixed point filter process similarly to the floating point implementation creates a fixed point filter. The filter needs a multiplier and the actual filter record type signal. The `create_first_order_filter` procedure call creates the state machines that are needed for the fixed point filter calculation. The implementation of the fixed point filter can be found in the math library [https://github.com/hVHDL/hVHDL\\_math\\_library/blob/main/first\\_order\\_filter/first\\_order\\_filter\\_pkg.vhd](https://github.com/hVHDL/hVHDL_math_library/blob/main/first_order_filter/first_order_filter_pkg.vhd)

The fixed point filter uses a slightly different implementation of the filter than the floating point one, but resulting response is the same.

Listing 6: fixed point filter implementation

```
fixed_point_filter : process(clock)
begin
    if rising_edge(clock) then
        init_bus(bus_out);
        connect_read_only_data_to_address(bus_in, bus_out, 104, get_filter_
        ↪output(filter)/2 + 32678);
        connect_read_only_data_to_address(bus_in, bus_out, 105, scaled_sine/2 + 32678);
        create_multiplier(multiplier2);
        create_first_order_filter(filter => filter , multiplier => multiplier2 , time_
        ↪constant => filter_time_constant);

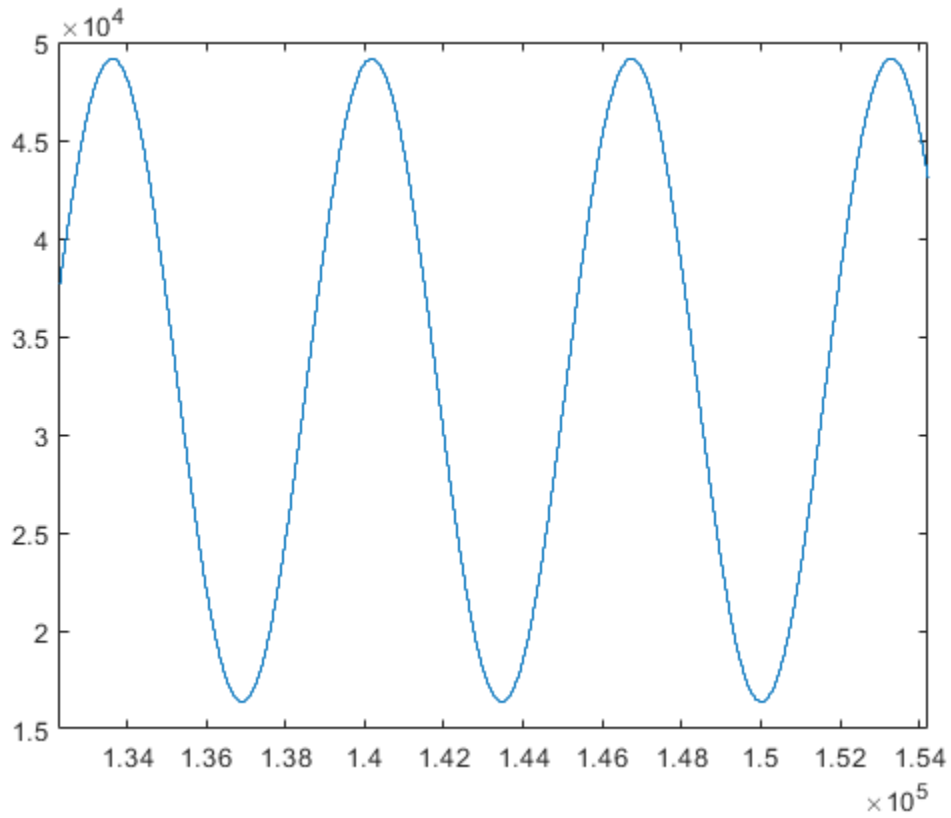
        if example_filter_input.filter_is_requested then
            filter_data(filter, example_filter_input.filter_input);
            process_counter <= 0;
        end if;

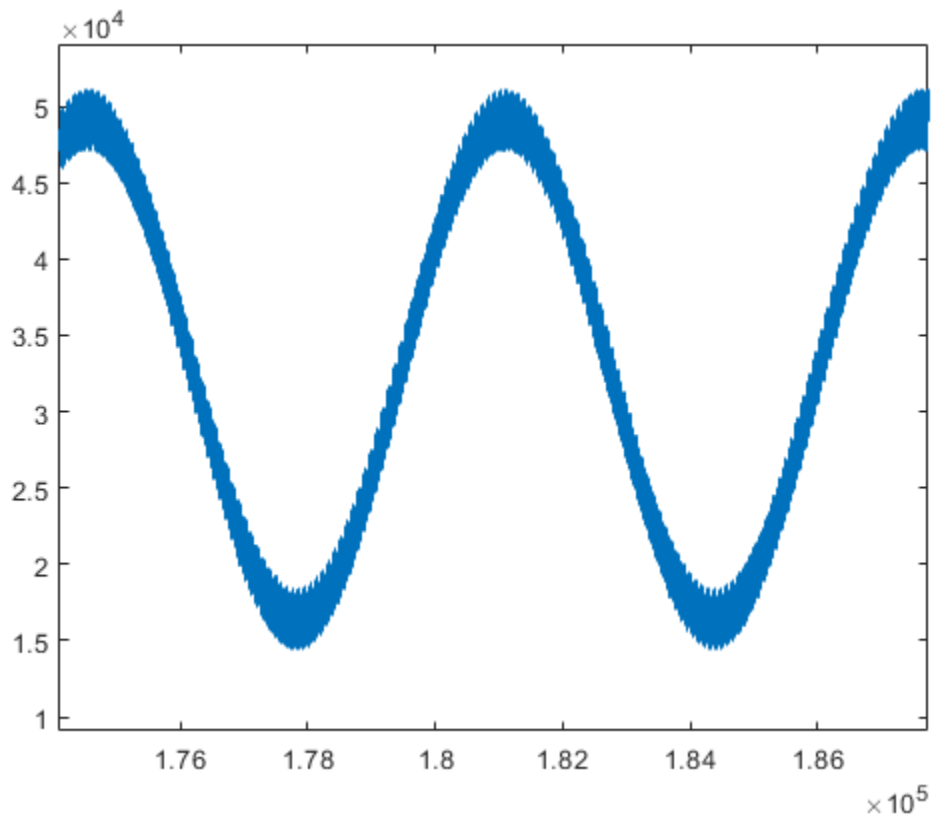
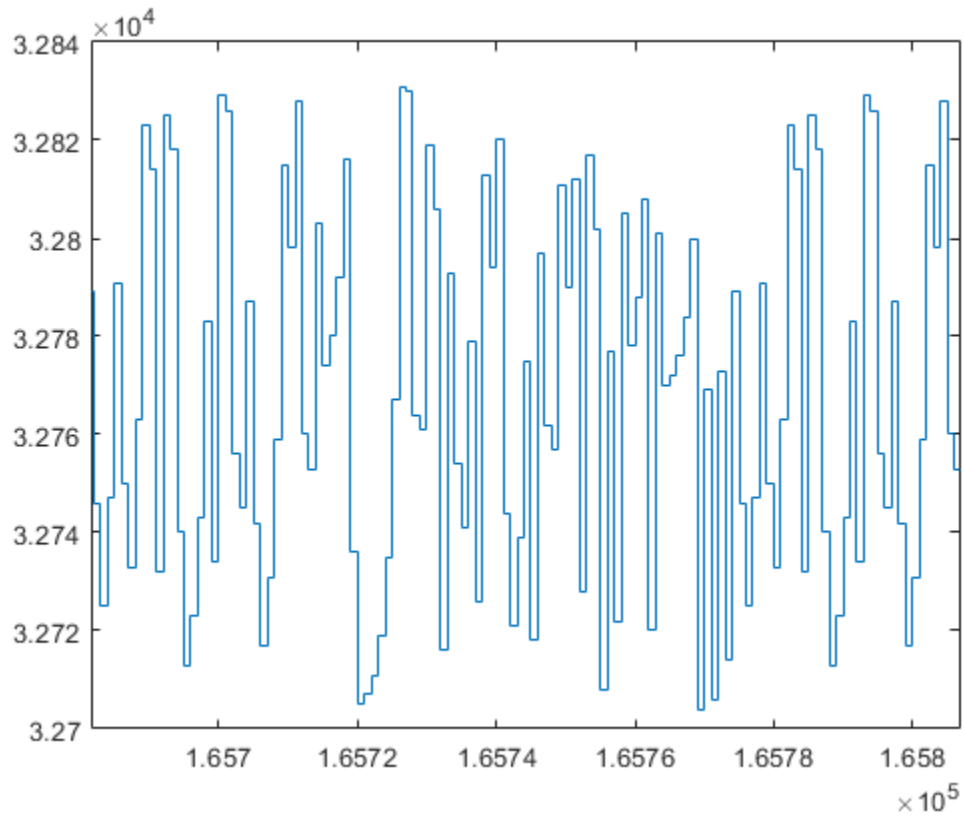
        CASE process_counter is
            WHEN 0 =>
                if filter_is_ready(filter) then
                    multiply(multiplier2, get_filter_output(filter), integer(32768.0*3.
                    ↪3942));
                    process_counter <= process_counter + 1;
                end if;
            WHEN 1 =>
                if multiplier_is_ready(multiplier2) then
                    scaled_sine <= get_multiplier_result(multiplier2, 15);
                    process_counter <= process_counter + 1;
                end if;
            WHEN others => -- wait for start
        end CASE;

        end if; --rising_edge
    end process;
```

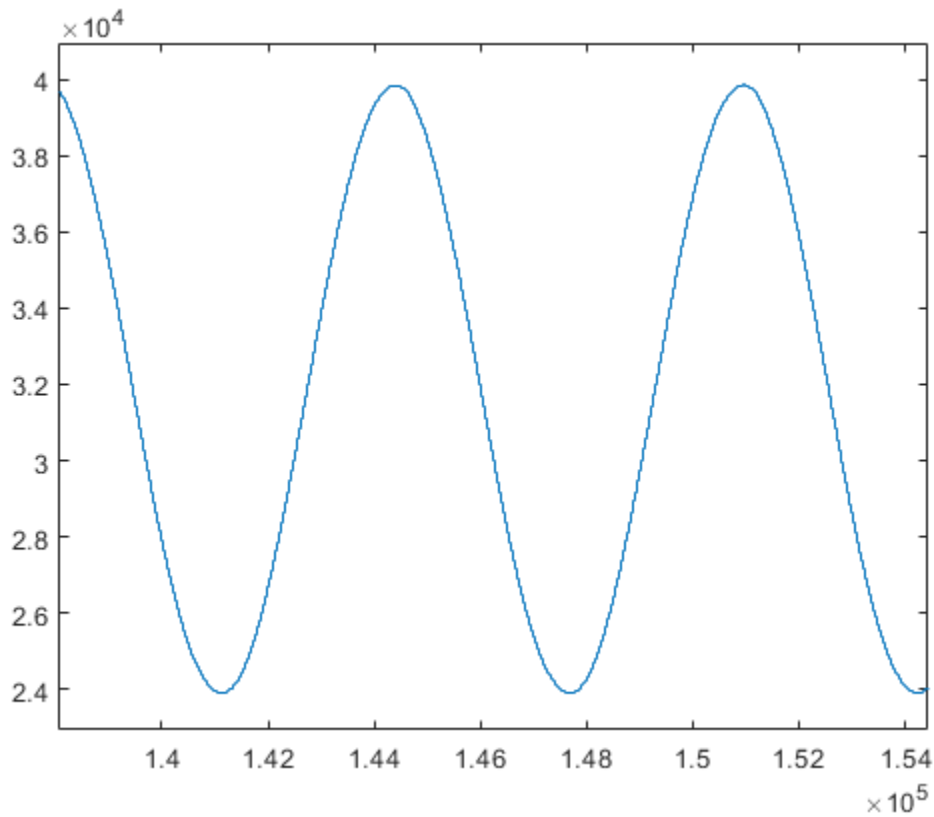
## 4.6 Filtering results

The signals are captured using uart and plotted into figures with matlab. The original sine, noise and combined noisy sine are shown below

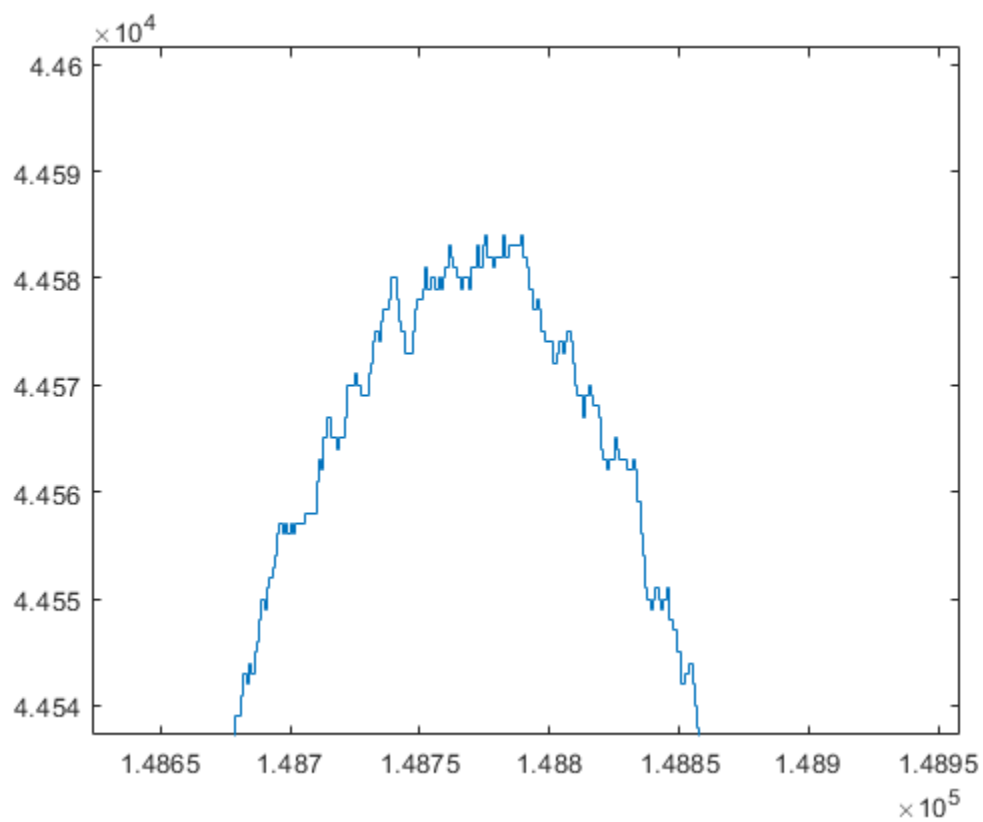
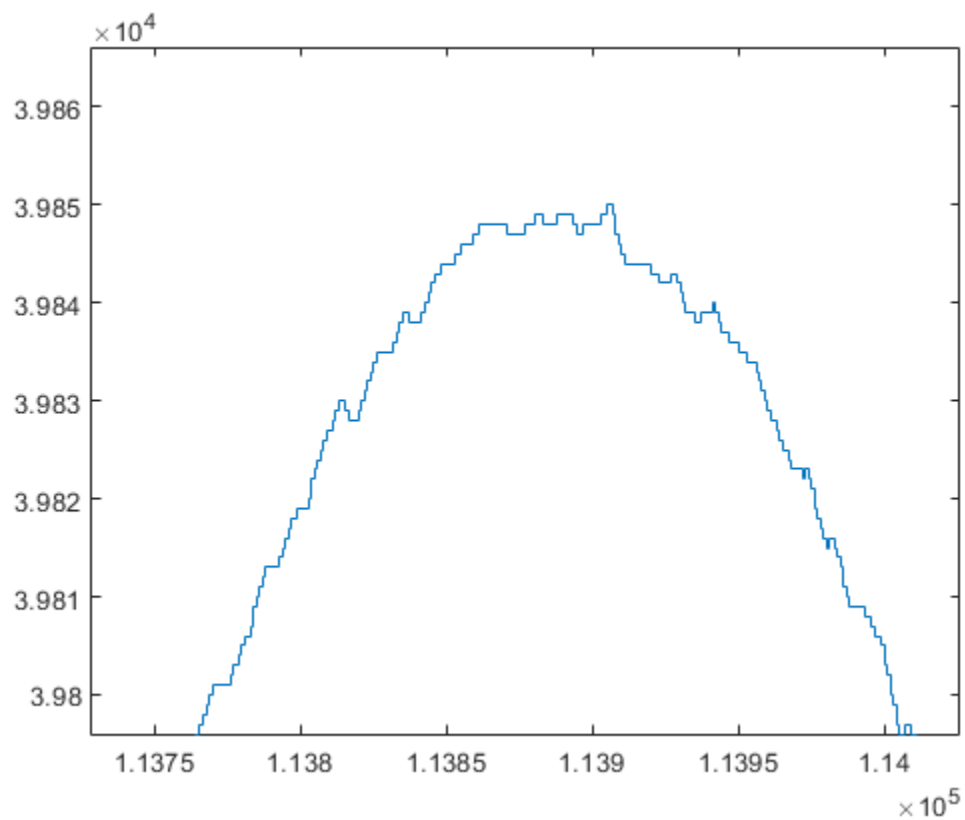




Since the fixed and floating point implementations of the filters have the same response, the filtering results are also almost identical.



The difference between fixed and floating point can be seen in the zoomed in figures shown below. The fixed point figure is first



## 4.7 Synthesis results

The synthesis results are shown in figure below the ecp, efinix and intel resource use including both luts and multipliers are quite similar due to 4 input lookup tables. Spartan 7 has six input luts and the compiler manages to use a lot of the 6 input luts, thus the resource use is somewhat smaller. All of the designs meet timing at 120MHz.

### Design Summary


```
Number of registers: 2136 out of 12687 (17%)
  PFU registers:      2134 out of 12096 (18%)
  PIO registers:      2 out of 591 (0%)
Number of SLICES:    2365 out of 6048 (39%)
  SLICES as Logic/ROM: 2341 out of 6048 (39%)
  SLICES as RAM:      24 out of 4536 (1%)
  SLICES as Carry:    399 out of 6048 (7%)
Number of LUT4s:     3353 out of 12096 (28%)
  Number used as logic LUTs: 2507
  Number used as distributed RAM: 48
  Number used as ripple logic: 798
  Number used as shift registers: 0
Number of PIO sites used: 3 out of 197 (2%)
Number of block RAMs: 0 out of 32 (0%)
Number of GSRs: 0 out of 1 (0%)
JTAG used : No
Readback used : No
Oscillator used : No
Startup used : No
DTR used : No
Number of Dynamic Bank Controller (BCINRD): 0 out of 4 (0%)
Number of Dynamic Bank Controller (BCLVDSOB): 0 out of 4 (0%)
Number of DCC: 0 out of 60 (0%)
Number of DCS: 0 out of 2 (0%)
Number of PLLs: 1 out of 2 (50%)
Number of DDRDLLs: 0 out of 4 (0%)
Number of CLKDIV: 0 out of 4 (0%)
Number of ECLKSYNC: 0 out of 10 (0%)
Number of ECLKBRIDGECS: 0 out of 2 (0%)
Notes:-
  1. Total number of LUT4s = (Number of logic LUT4s) + 2*(Number of
distributed RAMs) + 2*(Number of ripple logic)
  2. Number of logic LUT4s does not include count of distributed RAM and
ripple logic.

Number Of Mapped DSP Components:
-----

MULT18X18D      6
MULT9X9D        0
ALU54B          2
ALU24B          0
```



▸ Placement	
▸ Routing	
▸ Bitstream	
Debugger	
MIPI RX	0 / 2
MIPI TX	0 / 2
PLL	1 / 7
Core Resources	
Inputs	2 / 1108
Outputs	1 / 1263
Clocks	1 / 16
Logic Elements	3134 / 112128
Memory Blocks	0 / 1056
Multipliers	6 / 320
Interface	
Missing Interface Pins	0
Unassigned Core Pins	0

Flow Summary	
 <<Filter>>	
Flow Status	Successful - Mon Jul 18 10:47:56 2022
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	top
Top-level Entity Name	top
Family	Cyclone 10 LP
Device	10CL025YU256I7G
Timing Models	Final
Total logic elements	2,905 / 24,624 ( 12 % )
Total registers	1684
Total pins	3 / 151 ( 2 % )
Total virtual pins	0
Total memory bits	0 / 608,256 ( 0 % )
Embedded Multiplier 9-bit elements	11 / 132 ( 8 % )
Total PLLs	1 / 4 ( 25 % )

Site Type	Used	Fixed	Available	Util%
Slice	751	0	2000	37.55
SLICEL	510	0		
SLICEM	241	0		
LUT as Logic	2283	0	8000	28.54
using 05 output only	0			
using 06 output only	1992			
using 05 and 06	291			
LUT as Memory	40	0	2400	1.67
LUT as Distributed RAM	0	0		
LUT as Shift Register	40	0		
using 05 output only	14			
using 06 output only	5			
using 05 and 06	21			
Slice Registers	1438	0	16000	8.99
Register driven from within the Slice	1020			
Register driven from outside the Slice	418			
LUT in front of the register is unused	240			
LUT in front of the register is used	178			
Unique Control Sets	55		2000	2.75

\* \* Note: Available Control Sets calculated as Slice \* 1, Review the Control Sets Report for control sets.

9. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0	0	10	0.00
RAMB36/FIFO*	0	0	10	0.00
RAMB18	0	0	20	0.00

\* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate 018E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a

1. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	4	0	20	20.00
DSP48E1 only	4			



## FIXED POINT MATH

Fixed point math library allows high level access to basic mathematical functions. These include Multiplier, divider, sine and cosine, abc to dq and dq to abc transforms, PI controller and a first order filter. When we talk about fixed point we refer to arithmetic done using integers. Note that this applies to both fixed point and floating point arithmetic.

The modules are found at [https://github.com/hVHDL/hVHDL\\_math\\_library](https://github.com/hVHDL/hVHDL_math_library)

### 5.1 Multiplier

To add multiplier into your design you need to include the multiplier package and a word length configuration in the same library. The configuration package allows changing the required word lengths as well as number of pipeline cycles if needed. There are ready made configuration packages for 18x18, 22x22 and 26x26 bit multipliers.

The shift and rounding logic is in the `get_multiplier_result` function.

Listing 1: example of using a single multiplier for several multiplications

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

  use work.multiplier_pkg.all;

entity test is
  port ( clock : in std_logic);
end entity test;

architecture rtl of test is

  signal multiplier      : multiplier_record := init_multiplier;

  signal counter         : integer := 0;
  signal multiplier_result : integer := 0;

begin

  process(clock)
  begin
    if rising_edge(clock) then
      counter <= (counter + 1) mod 2**15;
      create_multiplier(multiplier);
    end if;
  end process;
end architecture;
```

(continues on next page)

(continued from previous page)

```

        multiply(multiplier, counter, counter);
        if multiplier_is_ready(multiplier);
            multiplier_result <= get_multiplier_result(multiplier, 15);
        end if;

    end if; --rising_edge
end process;
end rtl;

```

## 5.2 Divider

In addition to the sources in the division folder, divider also requires a multiplier. The divider is based on inverting the divider and then multiplying the result. The divider has a range reduction function which allows the inverting and resulting multiplication to work with numbers in [0.5, 1] range. A more thorough explanation of the divider is given in <https://hardwaredescriptions.com/conquer-the-divide/>

Listing 2: example of using divider

```

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

    use work.multiplier_pkg.all;
    use work.divider_pkg.all;

entity test is
    port ( clock : in std_logic);
end entity test;

architecture rtl of test is

    signal multiplier : multiplier_record := init_multiplier;
    signal divider    : divider          := init_divider;

    signal counter      : integer := 0;
    signal divider_result : integer := 0;

begin

    process(clock)
    begin
        if rising_edge(clock) then
            counter <= (counter + 1) mod 2**15;
            create_multiplier(multiplier);
            create_multiplier(divider);

            if division_is_not_busy(divider) then
                request_division(divider, counter, counter);
            end if;
        end if;
    end process;
end architecture rtl;

```

(continues on next page)

(continued from previous page)

```

        multiply(multiplier, counter, counter);
        if division_is_ready(multiplier, divider) then
            divider_result <= get_division_result(multiplier, divider, 15);
        end if;

    end if; --rising_edge
end process;
end rtl;

```

## 5.3 Sine and Cosine

Sine and cosine functions are calculated using polynomial approximation. We calculate both of them in at the same time, since the multiplication has pipeline stages, the cosine polynomial is evaluated in the pipeline stages of the sine polynomial and this allows us to save a blocking multiplier stage. Because of this, we get both at the same time.

Listing 3: example of using sine and cosine

```

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

    use work.multiplier_pkg.all;
    use work.sincos_pkg.all;

entity test is
    port ( clock : in std_logic);
end entity test;

architecture rtl of test is

    signal multiplier : multiplier_record := init_multiplier;
    signal sincos      : sincos_record    := init_sincos;

    signal counter : integer := 0;
    signal sine    : integer := 0;
    signal cosine  : integer := 0;

begin

    process(clock)
    begin
        if rising_edge(clock) then
            counter <= (counter + 1) mod 2**15;
            create_multiplier(multiplier);
            create_sincos(sincos);

            request_sincos(sincos, counter*64 mod 2**16);

            if sincos_is_ready(sincos);

```

(continues on next page)

(continued from previous page)

```
        sine    <= get_sine(sincos);  
        cosine  <= get_sine(sincos);  
    end if;  
  
    end if; --rising_edge  
end process;  
end rtl;
```

## 5.4 synchronous coordinate transforms



## FLOATING POINT MATH

Floating point math library has pipelined versions of multiplier and summation/subtraction modules that allow high level interface to basic floating point operations

[https://github.com/hVHDL/hVHDL\\_floating\\_point](https://github.com/hVHDL/hVHDL_floating_point)

The floating point module is configured using packages and libraries. There is also a blog post on the alu found at

<https://hardwaredescriptions.com/high-level-floating-point-alu-in-synthesizable-vhdl/> Multiplier ———

### 6.1 Adder/subtractor



## FPGA MEMORY

High level interface for rom and ram modules

[https://github.com/hVHDL/hVHDL\\_memory\\_library](https://github.com/hVHDL/hVHDL_memory_library)



## FPGA INTERCONNECT

High level bus for inter process communication inside fpga.

[https://github.com/hVHDL/hVHDL\\_fpga\\_interconnect](https://github.com/hVHDL/hVHDL_fpga_interconnect)



## **DYNAMIC VERIFICATION MODULE**

Dynamic verification libraries for power electronic converters and motor control

[https://github.com/hVHDL/hVHDL\\_dynamic\\_model\\_verification\\_library](https://github.com/hVHDL/hVHDL_dynamic_model_verification_library)





## GIGABIT ETHERNET

Gigabit ethernet module using RGMII interface

[https://github.com/hVHDL/hVHDL\\_gigabit\\_ethernet](https://github.com/hVHDL/hVHDL_gigabit_ethernet)



## UART

Uart with simple interface. Tested with various fpgas.

[https://github.com/hVHDL/hVHDL\\_uart](https://github.com/hVHDL/hVHDL_uart)



## CREATE HIGH LEVEL INTERFACES IN VHDL

The interfaces are designed to greatly increase abstraction level of VHDL source code. The key idea behind the patterns are 1) all code should be shareable 2) all code should be changeable when needed. To accomplish these two almost opposite ideas we have designed specific patterns for coding - all functionality should be behind abstract interfaces - all code modules should have the possibility to exert backpressure - there should be no need to time code - Use any IP from any vendor, add abstract interface if not already given by vendor

The main property of an interface is to create a well defined methods for accessing some functionality. The reason for having this interface is that the code behind the interface is indirectly used. This allows the code to be changed and modified without the changes being propagated to the application. This also allows sharing code accross multiple projects.

We can create an interface to either an IP component, that is an entity with a port as well as a set of functionality by using record as a minimal entity that is then used inside a process.

In VHDL records define the abstract data types and functions and procedures then define the way these data types are interacted with.

### 12.1 Records as abstract data types in VHDL

The main abstract data type in VHDL is called a record. Records are aggregates of any other types. Thus they can be made from anything you can make a signal from. This includes all types like arrays of records.

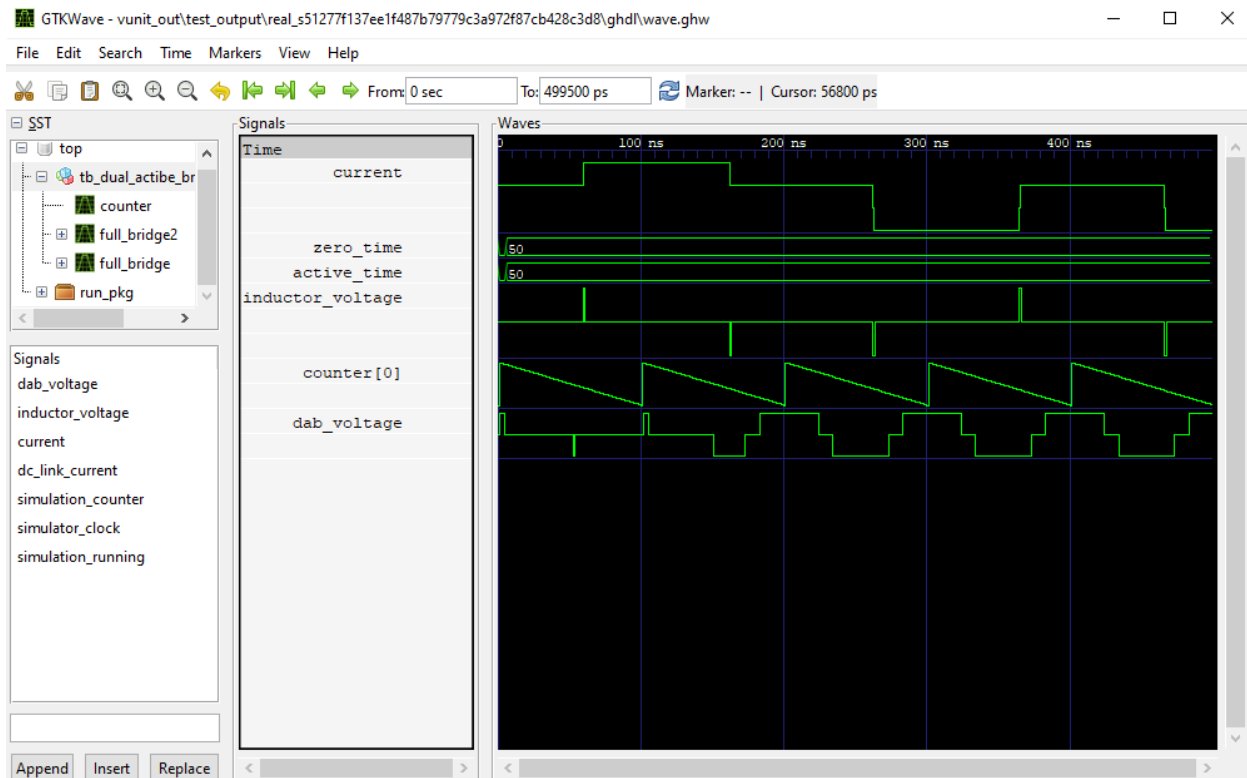
This record object is then added to a project by simply adding a signal of this record type. Since the entire functionality of a multiplier is encapsulated in this one signal of record type, we only need to instantiate a signal of this multiplier\_record type.

### 12.2 Subroutines

The magic happens when we use inout as the access type to the signal that is input to the module. With the inout declaration, we can put any multi cycle logic into a record.

## 12.3 Using a simple interface

The way we use these high level objects is through the subroutine interface. These interfaces are used with the signal of object record type as the argument. This way all of the specifics of the code remain in one place and therefore any changes in the code are also propagated to everywhere the code is used. Because of this we can modify code inside the abstracted interface and changes are propagated everywhere the code is being used.



## USING LIBRARIES AND PACKAGES TO ABSTRACT VHDL

Libraries in VHDL play a special role. Together with the abstracted interfaces libraries allow us to abstract code even further. With libraries we can change the insides of a record. This way we can change both the data and the interface with which the data is being accessed.

We commonly add the same sources to multiple libraries when we are reusing code. This is because this allows us to change things inside code like functions, procedures and records.

A few examples of this is for example using lookup tables. Here we have an example of a lookup table generator.

### 13.1 The specialty of use work.package

The mechanism in VHDL that allows us to do this is the work library. Referencing package through work library allows us to leave the library undefined. Thus we can use this as an interface for the insides of the records and functions. By compiling a module together with a package which defines some functions and records, we can change the package which we compile it with and thus change the implementation.

This allows us to create abstract interfaces that only define a behavior, like a function is called with an integer and when it is ready it returns an integer. We do not need know if the module uses one clock cycle or multiple clock cycles.





## AUTOMATIC CODE TIMING WITH HANDSHAKE INTERFACES

The way we allow any code to be allow to be changed is by using a handshake mechanism. This handshaking allows the application code to remain insensitive towards the timing of the code. This is very important, as without this feature the entire code is depending on there being no changes to the latencies and pipeline cycles. With this hanshaking we can only use the design through the behavioral description and we can change any part of a code and it's latency without it affecting the users.

### 14.1 Backpressure instead of timing

The huge benefit of using the concept of backpressure, or handshake is that the code it self manages timing relation between modules. This is absolutely mandatory to allow for the code to be changed later. The `is_ready` function also makes writing tests exceptionally simple as we can easily combine together multiple modules with differing latencies.

### 14.2 Adding backpressure to sequential / blocking function

Adding a ready functionality to a sequential module is done by assinging the output to a signal at the same time as the module is ready. For an easy example, we have a counter which flags a true for one clock cycle upon the count being ready. Since this ready flag is added in the same clock edge as the counter is incremented, the module is ready returns true at the same time as the counter is 0. Thus we can check for the module running by checking whether the counter is zero and checking that it is done by checking the boolean.

### 14.3 Adding backpressure to pipelined function

The difference between a blocking function and pipelined function is that a pipelined function can be called every clock cycle. The module takes some number of clock cycles for the result to be ready, but we can add a new instruction to the pipeline every clock cycle.

The way we add a ready feature to this type of a module is by adding a pipeline that has equal length to the actual module pipeline. The ready function then just checks for the last bit of the pipeline to be '1'. This way we can vary the pipeline length and still have the application code function as intended.

## 14.4 Adding ability to do backpressure to a IP module

It is very common for an IP module not to have a ready-request interface. However we can add this to the IP module along with the interface wrapper. A simple example of this is a RAM IP. This is pipelined function, that is we can request a new register from ram every clock cycle.

## REAL NUMBERS IN SYNTHESIZABLE VHDL

Real numbers can be used as a form of an abstract interface to either fixed or floating point arithmetic functions.

### 15.1 Real numbers as a form of abstract interfaces

Real numbers can be used as constants to any synthesizable module as long as they are not signals and they are used in functions that return a synthesizable type.



## HIGH LEVEL CODING PATTERNS IN SYNTHESIZABLE VHDL

The main things that make up of a high level coding pattern is abstraction.

### 16.1 Creating callable object with records and subroutines

### 16.2 Creating objects from objects

### 16.3 Simple state machines

We should strive for making only linear state machines. Here we will show how to achieve arbitrarily complex behavior with only linear state machines

Separate trigger and catch counters for timing insensitive design



## REUSING VHDL SOURCE CODE

The most important feature of high level coding patterns is the possibility for code reuse. This means that we can design incrementally. By designing incrementally we can have many small modules that form more complex features. This way we can use only a small part of a functionality.

### 17.1 Packages

In vhdl the way we share code is by putting it into packages. Packages can have type definitions for arrays and records as well as functions and procedures and constants. Packages can also reference further packages, thus functions can be built as part of other functions.

In hardware this means that we are chaining together logic without registers. This is a fantastic feature since we can trivially pipeline a chain of functions by just registering the function outputs.





## SHARING HARDWARE RESOURCES

There are many different resources that need to be shared accross designs like memories, multiplier, communication lines etc.

### 18.1 Sharing hw resouces inside a procedure

This is exceptionally simple to do. Due to the created interfaces, we can simply call the same request functionality from a created object as many times as required.

### 18.2 Sharing hardware accross processes with a bus pattern

When we reuse hardware accross processes, we need to create a way to access the same resource from two places. VHDL does not allow us to simply have two processes driving a signal, thus we need to create a way to do this. This is called a bus pattern. The idea is very simple, just OR/AND together the input record signal of a port signal. We do need to use three records, the two that are combined as the input and one for the output. However with this combined bus, we now can access the data from two processes.

This simple way of creating buses allows us to access a shared resource from multiple processes thus gives the possibility for accessing shared resouce



## **SOLVING DIFFERENTIAL EQUATIONS ON FPGA**

FPGAs are really good for running dynamical system simulation in real time. Here we will explain how to do that.

### **19.1 Numerical integration**