VHDP Overview

Inhalt

Structural Syntax

Main

Main translates to the top-level entity in VHDL. This is the main file for the project.

A CLK input is always created and used for all processes as default.

Example:

```
Main
(
  LED : OUT STD_LOGIC := ‘0’;
)
{
  ...
}
```

In the brackets, Signals are declared that will be connected to the FPGA I/Os. Package can be added in the Area too.

In the brace is the area for Process, NewComponent and Signal declarations.

Component

Component is a component for your FPGA design.
It works the same as Main except of three main differences: It has a name, you can add Generic to set parameters of the component and the I/Os will be connected to Signals in a different Component or in Main.
You can add an instance with NewComponent.

Example:

```
Component <Name>
(
  Generic ( CLK_Frequency : NATURAL := 50000000; );
  LED : OUT STD_LOGIC := ‘0’;
  En   : IN STD_LOGIC := ‘0’;
)
{
  ...
}
```

NewComponent

NewComponent adds an instance of a Component in a different Component or in Main.

Example:

```
NewComponent <Name>
(
  CLK_Frequency => 50000000,

  LED => LED,
  En   => ‘1’,
);
```

In front of NewComponent stands the name of the Component.
In the brackets, parameters from Generic can be assigned to values, constants or can be removed to use the default value (here 50000000 from Component).
The I/Os can be assigned to values, constants, signals and outputs can be removed if not used.

Generic

Is used to set parameters of a component, like the number of inputs to debounce or the clock frequency.

Example:

```
Component <Name>
(
 Generic (
  CLK_Frequency : NATURAL := 12000000;
  Inputs range 1 to 8 : NATURAL;
  Error_Correction : BOOLEAN := false;
 );
 …
)


NewComponent <Name>
(
 CLK_Frequency => 50000000,
 Inputs => 1,
 --Error_Correction is false if not set

 …
);
```

Package/Include

Can be added to Main or Component in the brackets. Package allows to use own datatypes and constants in the whole project. Packages are included in all files. If you only want to use some of them, use Inlcude.

Example:

```
Main
(
 Include ( <Package Name>, … );
 Package <Package Name>
 (
  TYPE <Type Name> (a, b, c …);
 )
 LED : OUT STD_LOGIC := '0';
)
{
 ...
}
```

You can put Package in a separate file with braces and can declare functions next to types and constants.

Example:

```
Package <Package Name>
{
  TYPE <Type Name> (a, b, c …);

  Function <Function_Name> (return INTEGER; value_in : INTEGER)
  {
    ...
  }
}
```

Process

In Process, you write your code. Every CLK cycle this code is executed. If you want to use a different CLK, surround your code with If(rising_edge(CLK2)) { … }.

Example

```
Process <Name (optional)>
(
  VARIABLE var : NATURAL range 0 to 3 := 0;
)
{
  If (…)
  {
    …
  }
}
```

A name can be assigned to a process.
In the brackets, variables can be declared that can be assigned and read only in this process (they can be declared in the brace too).
In the brace, the code can be written. Only If, Else, Elsif, Case, When and For should be used in the brace.
See Thread for the other operations.

Thread

Thread allows programming like with process-oriented programming languages.
You can use While and Wait together with If and Case. Everything will be converted to if-structures afterwards.
Thread also converts If, Case and signal assignments to work together with the procedure operations, because the If without Thread wouldn't wait for the While loop.

Example

```
Process()
{
  Thread
  {
    While(Btn = '0'){}
    Led <= '1';
    Wait(10000);
    While(Btn = '1'){}
    Wait(10000);
  }
}
```

What you must look for: Sometimes a signal e.g. has to be set from '0' to '1', but if you write

```
En <= '0';
En <= '1';
```

both assignments will be executed in the same cycle and a different Process would not see the change of the state.
What you can do is

```
En <= '0';
Step{ En <= '1'; }
```

This way En <= '1' is executed in a separate step.

Warning: Variables are created in the compilation process. Names of procedure operations in combination with numbers should not be used as names for own signals or variables.

Function

Functions can be used for implementing frequently used algorithms in your code. It takes zero or more values and always returns a value, but you can only use If, Case or For, because it has to run in one cycle.

Example (overcomplicated)

```
Function add_two (return INTEGER; value_in : INTEGER)
{
  VARIABLE example_var : INTEGER;

  example_var := value_in;
  For(i IN 0 TO 1)
  {
    example_var := example_var + 1;
  }
  return example_var;
}
```

StepFunction

StepFunctions can be used for implementing frequently used algorithms in your code that also uses e.g. While or Wait.
The content inside the function is inserted at the position of NewFunction and names of the parameters are replaced with the connected signals. Instead of a return value you can assign the value to a parameter and read the value of the connected signal inside the Process. Variables that are declared in the function, are added to the process variables and can also be used in the Process.

Example

```
StepFunction printChar
(
  char : STD_LOGIC_VECTOR(7 downto 0);

  ena : STD_LOGIC;
  busy : STD_LOGIC;
  data : STD_LOGIC_VECTOR(7 downto 0);
)
{
  While(busy = '1') {}
  data <= char;
  ena <= '1';
  While(busy = '0') {}
  ena <= '0';
  While(busy = '1') {}
}

NewFunction printChar (x"56", UART_Enable, UART_Busy, UART_Data);
```

Advanced: If you add FunctionContent in the function, the sections outside of FunctionContent will be added outside of the Process where you add NewFunction.

Generate

Allows to add a component or an operation to the project multiple times or if a condition is met.

Example:

```
Generate (for i in 0 to 7)
{
  NewComponent PWM_Generator
  (
    Duty  => dutySig(i),
    PWM_Out(0) => LEDs(i),
  );
}
```

Connections

Connections can be used to help with connecting the Component I/Os with the correct FPGA I/Os. If there are I/Os in Main with the same name that aren't already connected, they will be connected automatically with the given FPGA Pins.

Example:

```
Connections
{
  RX =>D5,
  TX => F4,
}
```

VHDL (Generate, When)

For all functions that are not (already) implemented, with VHDL{} you can add VHDL code to your VHDP code. If you want to add something to the signal declaration area, you have to write AttributeDeclaration{VHDL{}}.

Example Generate

```
AttributeDeclaration
{
  VHDL
  {
    COMPONENT BinaryToBcdDigit IS PORT(
      CLK           : IN            STD_LOGIC;
      Reset         : IN            STD_LOGIC;
      …);
    END COMPONENT;
  }
}

VHDL
{
  digit_0: BinaryToBcdDigit PORT MAP (
    CLK,
    Reset,
    …
  );
}
```

Datatypes

BIT, BIT_VECTOR, BOOLEAN

Represents logic values and can be used together with logical operators.

Example BIT_VECTOR

```
VARIABLE BitVectorSig : BIT_VECTOR (7 downto 0) := (others => '0');
```

Equals "00000000" (Order = "76543210")

Example BIT

```
VARIABLE BitSig1 : BIT := '0';
VARIABLE BitSig2 : BIT := '1';

BitSig1 := '1';
If((BitSig1 AND BitSig2) = '1')
```

Example BOOLEAN

```
VARIABLE BoolSig1 : BOOLEAN := false;
VARIABLE BoolSIg2 : BOOLEAN := true;

BoolSig1 := true;
If(BoolSig1 AND BoolSig2)
```

With Boolean you don't have to write (BoolSig1 AND BoolSig2) = true

STD_LOGIC, STD_LOGIC_VECTOR

STD_LOGIC is often used with VHDL, because it allows to set or read all possible states of an I/O. For example, for I²C, the output must switch between '0' and 'Z' (not connected to GND or 3V3). In addition, it is helpful to know if an input has a different state.

STD_LOGIC can be used like BIT, but with the different states

- 'U' <- Uninitialized
- 'X' <- Unknown
- '0' <- Low
- '1' <- High
- 'Z' <- High Impedance
- 'W' <- Weak Unknown
- 'L' <- Weak Low
- 'H' <- Weak High
- '-' <- Don't Care

UNSIGNED, SIGNED

Signed and unsigned variables are bit vectors that can be used for mathematical operations. If a variable is a signed, the range (7 downto 0) means a range from -128 to 127. If a variable is an unsigned, the range (7 downto 0) would mean a range from 0 to 255.

Example

```
SIGNAL counter : UNSIGNED (7 downto 0) := (others => '0');
counter <= counter + 1;
```

INTEGER, NATURAL, POSITIVE

Integer, natural and positive are numbers that have a range between two numbers and can be assigned to numbers.
Integer allows numbers from -2,147,483,647 to 2,147,483,647
Natural from 0 to 2,147,483,647 and Positive from 1 to 2,147,483,647

Example

```
SIGNAL counter : NATURAL range 0 to 255 := 0;
counter <= counter + 1;
```

User defined types (Enum, Array, Record)

Enum

Has different states with a custom name.

Example

```
TYPE EnumType IS (start, running, end);
SIGNAL EnumSig : EnumType := start;

EnumSig <= running;
```

Array

Allows to have a group of elements in one variable.

Example

```
TYPE PixelType IS ARRAY (0 to 2) OF NATURAL range 0 to 255;
TYPE RowType IS ARRAY (0 to 639) OF PixelType;
(TYPE RowType IS ARRAY (0 to 639, 0 to 2) OF NATURAL range 0 to 255;)
SIGNAL Row : RowType := (others => (others => '0'));

Row(0)(0) <= 0;
Row <= ((0, 1, 2),
        (3, 4, 5),
        …
        (6, 7, 8));
```

Record

Allows different types to be combined in one.

Example

```
TYPE RGBType IS RECORD
  R : NATURAL range 0 to 255;
  G : NATURAL range 0 to 255;
  B : NATURAL range 0 to 255;
END RECORD RGBType;

SIGNAL RGBSig : RGBType := (R => 0, G => 0, B => 0);

RGBSig.R <= 128;
RGBSig.G <= 0;
RGBSig.B <= 255;
```

Declarations and assignments

I/O

Needed to use the FPGA I/Os or as interface for a Component. (See Main)

Declaration

Structure: <Name> : <IN/OUT/INOUT/BUFFER> <Type> <Range> := <default>;
Example: Btn : IN STD_LOGIC;
Led : OUT STD_LOGIC_VECTOR (7 downto 0) := (others => '0');

IN = Signal that can be read but not assigned a value
OUT = Signal that can be assigned a value but not be read
BUFFER = Signal that can output a value but this value can be read
INOUT = Signal that can be used as in- or output

Type: Unsualy STD_LOGIC or STD_LOGIC_VECTOR

Range: For STD_LOGIC_VECTOR either (… downto 0) or (0 to …)

Default: Either none for e.g. an input or a value like '0' (or (others => '0') to set every bit to '0' in a vector)

Assignment

Structure: <Name> <= <Value>;
Example: Led <= "00011100"; (same as x"1C" or (4 downto 2 => '1', others => '0'))

Generic

Needed as parameter of a Component. (See Component)

Declaration

Structure: <Name> : <Type> := <default>;
Example: CLK_Frequency : NATURAL := 50000000;

Type: Often numbers (INTEGER or NATURAL)

Default: Useful, if you want to remove it in NewComponent

(Can only be read like a constant)

Signal

Needed as variable in a process or to let different processes talk to each other.
They can be declared everywhere where signals can be assigned or use the dedicated AttributeDeclaration{} section.
Signals can be written by one process but can be read in the whole file. If they are assigned in the process, the signal will have the value in the next cycle (use Variable to set instantly).

Declaration

Structure: SIGNAL <Name> : <Type> <Range> := <default>;
Example: SIGNAL Counter : NATURAL 0 to 255 := 0;

Range: For INTEGER, NATURAL and POSITIVE: … to … -> will allow numbers from … to …
For STD_LOGIC_VECTOR or other vectors: (… to/downto …) -> will have bit ... to …

Default: Important if you e.g. write Counter <= Counter + 1; to set the value to start with.

Assignment

Structure: <Name> <= <Value>;
Example: Counter <= Counter + 1;

Different parts of a vector can be set in one cycle, but the value can be read in the next.

Variable

Needed as variable in a process and for fast processing.
They can be declared in a process or in the dedicated brackets of Process.
Variables can be written and read by one Process, but they will be set instantly. So you can write count := count + 1; If (count = …) with the increased value.

Declaration

Structure: VARIABLE <Name> : <Type> <Range> := default;
Example: VARIABLE Counter : UNSIGNED (7 downto 0) := (others => '0');

Range & Default: same as Signal

Assignment

Same as with Signal, but use := except of <= (The VHDP IDE converts = to :=/<= automatically)

Constant

Needed as parameter in Main or a Component. Can be used like a Signal but can only be read.

Declaration

Structure: CONSTANT <Name> : <Type> := <value>;
Example: CONSTANT Width : NATURAL := 8;

Standard operations

If, Elsif, Else

Example

```
If (a = 0) { a := 1; }
Elsif (a = 1) { a := 2; }
Else { a := 0; }
```

Case, When

Example

```
Case (a)
{
  When (0) { a := 1; }
  When (1 to 3) { a := a + 1; }
  When (4 | 6 | 8) { a := 0; }
  When (others) { a := 0; }
}
```

All possible numbers defined by range must have one when, but you can add When (others) { null; }.

For

Example

```
For (i IN 7 DOWNTO 1)
{
  LED(i) <= LED(i-1);
  If(i = exitValue) { exit; }
}
```

The for loop can count up (TO) or down (DOWNTO) between constant values. With exit; you can leave the loop before it is finished. The name of the counter (i) can be used as variable.

Procedure operations

Verhalten in Process und Thread

StepIf, StepElsif, StepElse

Example

```
Thread
{
 If (BTN = '1') { LED <= '1'; }
 Wait (10000);
}
```

```
StepIf (BTN = '1') { LED <= '1'; }
Wait(10000);
```

First StepIf checks if the button is pressed, sets the led and afterwards Wait waits some time.
If you write the first code without Thread, "If" would ignore Wait and check if the button is pressed every cycle.
Thread converts the "If" to StepIf like in the second example. The difference in the second example is that the next operations will also wait for the previous and you only have one "thread".

StepCase, StepWhen

Example

```
Thread
{
 Case (a)
 {
  When (0)
  {
   Wait(1000);
  }
  When (others)
  {
   Wait(2000);
  }
 }
}
```

```
StepCase (a)
{
 StepWhen (0)
 {
  Wait(1000);
 }
 StepWhen (others)
 {
  Wait(2000);
 }
}
```

Step

Example

```
Thread
{
 Length <= 100;
 En <= '1';
 Step { En <= '0'; }
}
```

```
Step
{
 Length <= 100;
 En <= '1';
}
Step { En <= '0'; }
```

Thread will automatically put a Step around signal assignments, but Step is needed to wait one cycle between the assignments.

While

Example

```
Thread
{
 VARIABLE counter : INTEGER := 0;
 While (counter < 8)
 {
  LED(counter) <= '1';
  counter := counter + 1;
  Wait(1000);
 }
}
```

```
While (counter < 8)
{
 Step
 {
  LED(counter) <= '1';
  counter := counter + 1;
 }
 Wait(1000);
```

StepFor

Example

```
Thread
{
 StepFor (VARIABLE counter : INTEGER := 0;
          counter < 8;
          counter := counter + 1)
 {
  LED(counter) <= '1';
  Wait(1000);
}
```

Same with While:

```
Thread
{
 VARIABLE counter : INTEGER := 0;

 counter := 0;
 While (counter < 8)
 {
  LED(counter) <= '1';
  Wait(1000);
  counter := counter + 1;
 }
}
```

Wait

Example

```
Thread
{
 LED <= '1';
 Wait(1000);
 LED <= '0';
 Wait(1000);
}
```

```
Step{ LED <= '1'; }
Wait(1000);
Step{ LED <= '0'; }
Wait(1000);
```

Math, conversion and other operators

Operators

Example

```
If ((a AND b) = '1')
```

NOT	complement
AND	logical and
OR	logical or
NAND	logical complement of and
NOR	logical complement of or
XOR	logical exclusive or
XNOR	logical complement of exclusive or

=	test for equality
/=	test for inequality
<	test for less than
<=	test for less than or equal
>	test for greater than
>=	test for greater than or equal

Use a different clock for Process:

rising_edge([clk name])
falling_edge([clk name])

Example:

```
If (Reset = '0') { … }
Elsif (rising_edge(clk_50)) { … }
```

Type conversion

TO_BIT(<STD_LOGIC>)	STD_LOGIC to BIT
TO_BITVECTOR(<STD_LOGIC_VECTOR>)	STD_LOGIC_VECTOR to BIT_VECTOR
TO_STDLOGICVECTOR(<BIT_VECTOR>)	BIT_VECTOR to STD_LOGIC_VECTOR
SIGNED(<STD_LOGIC_VECTOR>)	STD_LOGIC_VECTOR to SIGNED
UNSIGNED(<STD_LOGIC_VECTOR>)	STD_LOGIC_VECTOR to UNSIGNED
STD_LOGIC_VECTOR(<SIGNED/UNSIGNED>)	SIGNED/UNSIGNED to STD_LOGIC_VECTOR
TO_SIGNED(<INTEGER>, <SIGNED>'LENGTH)	INTEGER/NATURAL/POSITIVE to SIGNED
TO_UNSIGNED(<NATURAL>, <UNSIGNED>'LENGTH)	NATURAL/INTEGER/POSITIVE to UNSIGNED
TO_INTEGER(<SIGNED/UNSIGNED>)	SIGNED/UNSIGNED to INTEGER

Mathematical operations

+	addition
-	subtraction
*	multiplication
/	division
**	exponential
abs	absolute value

Example

```
Result := ValA * ValB;
Result := abs(Result);
```

Concatenation (&, =>, others)

LED : OUT STD_LOGIC_VECTOR (7 downto 0) := (others => '0');

LED <= "1111" & "0000"; -> "11110000"
LED <= (7 => '1', 6 downto 4 => "111", others => '0'); -> "11110000"