MIGHTY MACROS AND POWERFUL PARAMETERS: MAXIMIZING EFFICIENCY AND FLEXIBILITY IN HDL PROGRAMMING

Muneeb Ulla Shariff, Vineeth Kumar Veepuri,
Nancy Dimri and Mahadevaswamy B N

Mirafra Technologies, Whitefield, Bengaluru, Karnataka, India

ABSTRACT

This paper explores the use of macros and parameters in Hardware Description Language (HDL) programming. Macros and parameters are powerful tools that allow for efficient and reusable code, yet their full potential is often underutilized. By examining the advantages of macros and parameters, this paper aims to demonstrate how these features can enhance the flexibility, readability, and maintainability of HDL code. Additionally, the paper discusses the use cases of mixing macros and parameters in HDL programming, highlighting their applicability in a range of scenarios. Furthermore, the paper addresses the challenges that arise from the mix use of macros and parameters and provides best practices to overcome these challenges. Overall, this paper aims to encourage HDL programmers to fully explore the capabilities of macros and parameters in their code, leading to more efficient and effective hardware designs and verification.

KEYWORDS

Precompilation, Compilation, Elaboration, Argument, SV, UVM , HDL

1. INTRODUCTION

In modern digital circuit design, Hardware Description Language (HDL) programming has become crucial. As designs become more complex, writing efficient and reusable code has become more significant than ever. Fortunately, HDL programmers have access to powerful tools that can help them achieve this goal. Macros and parameters are two such tools that offer a range of advantages to HDL programmers.

Macros are a means to generate code on-the-fly and reduce redundancy, allowing programmers to write code more efficiently. On the other hand, parameters offer flexibility in design by enabling quick and easy changes to values. Together, macros and parameters can improve the readability and maintainability of HDL code, while also reducing the risk of errors and increasing design efficiency.

2. COMPILER TOOL FLOW

The compiler tool flow consists of several stages, including pre-compiler, compilation, elaboration, and simulation, as depicted in Figure 1. Each stage has a specific purpose and function in the overall design flow, and understanding these stages is essential for successful circuit design.

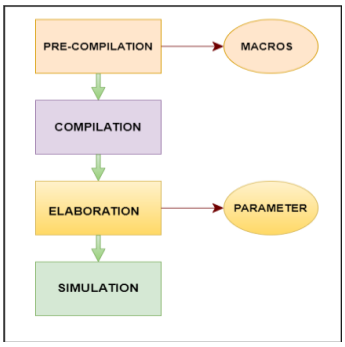

Figure 1. Stages of compiler tool flow

2.1. Pre-Compilation

The pre-compiler stage is the initial stage in the compilation process of a hardware description language (HDL) code. The pre-compiler stage is responsible for processing the code and making it ready for the main compilation process. It typically involves handling any preprocessing directives, such as define, *`ifdef*, *`ifndef*, *`include*, and *`timescale*, which are used to customize the code or include external libraries.

During the pre-compiler stage, the pre-processor reads through the code and replaces any instances of preprocessing directives with their defined values or corresponding code. For example, the *`define* directive defines a macro, which can be used throughout the code to simplify code and make it more readable. The pre-processor replaces any instances of the defined macro with its corresponding value.

2.2. Compilation

In the compilation stage, the input Verilog code is parsed and checked for syntax errors, and then converted into an intermediate format such as an Abstract Syntax Tree (AST) or an Intermediate Representation (IR). The AST or IR is then optimized and transformed by the compiler to generate optimized code for the target architecture.

2.3. Elaboration

In the elaboration stage, the tool reads the intermediate code representation generated in the compilation stage and builds a hierarchical design structure with instantiated modules, connections, and parameter values. It resolves all module and interface references and performs type checking to ensure that the design is structurally and syntactically correct. The elaboration stage also performs several optimizations, such as constant propagation and dead code elimination, to improve simulation performance. Once the elaboration is complete, the tool generates a design database that is used for simulation.

2.4. Simulation

After the elaboration stage, the design is ready for simulation. The simulation stage involves applying input stimuli to the design and observing its behavior. The simulator executes the design, and output results are generated and collected for analysis. The simulation stage can be used for both functional and timing verification of the design.

Point to remember: Macros are processed in the pre-compiler stage and parameters in the elaboration stage.

3. PARAMETER

In HDL, parameters are used to define constant values that are used throughout the design. Parameters are used to define values such as data widths, address ranges, clock periods, and other values that remain constant throughout the design.

Parameter values are evaluated during the elaboration phase of the simulation, before any code is executed. This means that parameter values are set at the beginning of the simulation and remain constant throughout the simulation. Parameter values are defined using the parameter keyword, and can be either integer or real values [1]

```
// Syntax
parameter PAR_VAR=value;
//example
parameter DATA_WIDTH=32;
```

Here *DATA_WIDTH* is the entity that stores ***32*** and wherever there is a usage of parameter *DATA_WIDTH* it will be replaced by the value assigned to it during the elaboration stage.

3.1. Basic Usage

```
module define_para;
parameter NUM=90,SUM="Hello";
initial begin
  $display("...................");
  $display("parameter num is %0d and sum is %0s",NUM,SUM);
end
endmodule
```

In the above code two parameters are being defined *NUM* and *SUM* has no type specified in such case the type shall be of final value assigned to it after any overrides, in this case the type of *NUM* is *int* and for *SUM* is *string*.

Parameters can have data type and default data type depends on the value. To give your own data type we can use the below syntax.

```
parameter <datatype> PAR_NAME=DEFAULT_VALUE
```

example

```
parameter int NUM = 90;
```

Output

```
...................
parameter num is 90 and sum is Hello
```

3.2. Parameterized Class Usage

```
class hello #(int SIZE=0);
  function new(string t);bit[SIZE-1:0] a;bit[3:0] count;
    for (int i=SIZE;i>=0;i--)begin
      $display(count+1,". Hello agent \"%0d\" ",i,t);count++; end
  endfunction
endclass
module para;
hello #(.SIZE(3)) h1;
initial begin
  h1 = new(.t("Welcome"));
end
endmodule
```

Here the entity ***hello*** refers to a parameterized class and anything within with parameter ***SIZE*** and the loop in the ***new constructor*** runs based on the parameter value.
Output

```
1. Hello agent "3" Welcome
2. Hello agent "2" Welcome
3. Hello agent "1" Welcome
4. Hello agent "0" Welcome
```

3.3. Parameterized Module Usage

In the module ***top,*** *as shown below* there are two instances of parameterized module ***adder*** one for 4-bit input and other for 8-bit input. If we need the same functionality without using parameters then we would need two different modules with the same logic which is time-consuming. But with the help of parameters, a lot of code space is saved along with the time.

```
module adder #(NO_OF_BITS)
(
input bit [NO_OF_BITS-1:0]a,input bit [NO_OF_BITS-1:0]b,
output bit [NO_OF_BITS-1:0]sum,output bit carry
);
assign {carry,sum}=a+b;
endmodule
module top;
adder#(4) add_a(12,12);adder#(8) add_b(36,127);
initial begin
  $display("a is %0d,b is %0d,sum is %0d and carry is
%0d",add_a.a,add_a.b,add_a.sum,add_a.carry*16);
  $display("a is %0d,b is %0d,sum is %0d and carry is
%0d",add_b.a,add_b.b,add_b.sum,add_b.carry*128);
end
endmodule
```

Output

```
a is 12,b is 12,sum is 8 and carry is 16
a is 36,b is 127,sum is 163 and carry is 0
```

The output shows the values of inputs ***a,b*** and outputs ***sum*** and ***carry***.

3.4. Parameters in *Generate* Block

During the elaboration stage, if there is a generate block in the code, it is executed to generate new code that becomes part of the module hierarchy. The generate block can be used to create different instances of a module based on parameter values or other conditions. The generated code is then added to the hierarchy of instantiated modules [4].

```
module d_ff#(ID)
(input bit clk,d,
output bit q);
always @(posedge clk)
assign d=q;
endmodule

module top;
parameter NO_OF_INST=1;
bit clk;
always #5 clk=~clk;
genvar i;
generate
defparam NO_OF_INST=2;
  for(i=0;i<NO_OF_INST;i++)
  begin
    d_ff#(.ID(i)) d(.clk(clk),.d(a[i]));
    $info("ID is %0d and %0d",i,d.ID);
  end
endgenerate
endmodule
```

In the code the ***generate*** block uses parameter ***NO_OF_INST*** to modify the number of instances of the module. Moreover, the ***defparam*** is used to modify the value of parameter.

Output

```
** Info: ID is 0 and 0
   Time: 0 ns  Scope: top1.genblk1[0] File: para_basic.sv Line: 71
** Info: ID is 1 and 1
   Time: 0 ns  Scope: top1.genblk1[1] File: para_basic.sv Line: 71
```

As you observe in the output, using parameters which are evaluated in elaboration time, along with ***generate*** blocks multiple instances of ***d_ff*** module are created and the count can be increased by modifying parameter ***NO_OF_INST.***

4. MACROS

Macros are preprocessor directives that allow the programmer to define a string of text that can be substituted in the code with another string. Macros can be defined using the `define directive and can take arguments, which makes them useful for generating code on-the-fly [1].

When a macro is defined, any instance of the macro name in the code is replaced by the text associated with the macro. This replacement occurs before the code is compiled, during the pre-processing stage. Macros can be used to reduce code redundancy, make code more readable, and to define parameters that can be used throughout the code.

```
// syntax
`define MESSAGE "Hello, world!"
module test;
initial begin
  $display(`MESSAGE);
end
endmodule
```

When the code is compiled, the preprocessor replaces the macro with the text "Hello, world!", resulting in the following code:

```
module test;
initial begin
  $display("Hello, world!");
end
endmodule
```

Thus, macros can simplify code and make it easier to read and maintain.

4.1. Single-Line Macros

```
`define fav_fruit(B) `"B apple`"
`define how_much_kg 23
module mod;
string fruit;
int kg;
initial begin: start
  fruit=`fav_fruit( Fresh);
  kg=`how_much_kg;
 $display("My favorite fruit is the %0s . Give me = %0d
kg",fruit,kg);
end: start
endmodule : mod
```

In precompilation time , the macros ***fav_fruit*** and ***how_much_kg*** is replaced by the ***Fresh apple*** and **23** respectively. In particular the data enclosed between ***`"(tick followed by double quotes)*** pair is to say that if there is any substitution available do it and consider the whole data as string.

So in the above code the macro ***fav_fruit*** has an argument **B** to be passed and in the value(the part enclosed within `"), wherever **B** is present will be replaced by the value passed as argument in this case it is ***Fresh***.

Output

```
My favorite fruit is the Fresh apple. Give me = 23 kg
```

4.2. Multiline Macros

We can declare macros in multiline also by using the **(**backslash**)** at the end of each line of macros. There are multiple cases where multi-line macros can be useful to make code more refined and reusable.

Some use cases are as discussed below:

4.2.1. Simple Multiline Macros

In this case below, declaring the macros in multiple lines using \(backslash). Check below code snippet for a better understanding:

```
`define MY_MACROS(X,Y,EXPRESSION)\
X EXPRESSION Y\
module mod;
int m=23;
int n=32;
initial begin
$display("The value of sum = %0d",`MY_MACROS(m,n,+));
$display("The value of sum = %0d",`MY_MACROS(m,n,-));
end
endmodule
```

In the above code, macros are declared as ***MY_MACROS(X,Y, EXPRESSION)*** on the top of the code. Inside the module ***`MY_MACROS(m,n,+)*** and ***`MY_MACROS(m,n-)***, where ***X*** is replaced by ***m***, ***Y*** is replaced by ***n*** and ***EXPRESSION*** is replaced by **+** and **-**.

For this code, the conclusion is a single macro can be used for getting multiple outputs. The below output snippet shows the output after simulation.

Output

```
# The value of sum = 55
# The value of sum = -9
```

4.2.2. Replace macros as function definition

Macros can be replaced by the whole function also [2].

```
`define MY_MACROS(X,Y,CUSTOME_NAME)\
function int CUSTOM_NAME(int X,Y);\
 int a;\
a=0;\
  a=X+Y;\
  return a;\
endfunction\
module mod;
int m,z;
`MY_MACROS(m,z,adder)
int n;
initial begin
  m=56;
  z=6;
  n=adder(m,z);
  $display("Sum = %0d",n);
end
endmodule
```

In the above code macros declare ***MY_MACROS(X,Y,CUSTOM_NAME)*** is replaced by ***CUSTOM_NAME*** function . Here, function arguments ***X***, ***Y*** and function are replaced by ***m***,***z***, and ***adder*** respectively. As we can see in the above code, we can customize the function using macros. Below is the output snippet of the above code.

Output

```
Sum = 62
```

4.2.3. Replace Macros as Assertion

Macros can be replaced by assertions also. Using macros will increase the reusability of assertion. The below code is the replacement of macros with assertions [2].

```
`define MACRO_DATA(X,Y) \
 sequence seqA;\
  x ##5 y ;\
endsequence\
property prop;\
  @(posedge clk) seqA ;\
endproperty\
assert property(prop) $info("assertion passed");\
  else $error("assertion failed");\
module MACRO_AS_ASSERTION;
bit clk, x, y;
always #1 clk = ~clk;
  `MACRO_DATA(x,y)
initial begin
  x =  0;
  y  = 0;
  #3 x = 1;
    y = 1;
  #2 x = 0;
  #4 y = 0;
  #5 x = 1;
  #5 y = 1;
  #10; $finish;
end
initial begin
  $dumpfile
  ("waveform.vcd");
  $dumpvars();
end
endmodule
```

In the above, macros declared as ***MACRO_DATA(X,Y)*** which will be replaced assertion. Inside the module ***clock(clk)*** is declared, replacing ***X*** and ***Y*** with ***x*** and ***y***. The below snippet shows the output with the time when the assertion passed and the assertion failed.

Output

```
# ** Error: assertion failed
#    Time: 1 ns Started: 1 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Error: assertion failed
#    Time: 3 ns Started: 3 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Error: assertion failed
#    Time: 7 ns Started: 7 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Error: assertion failed
#    Time: 9 ns Started: 9 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Error: assertion failed
#    Time: 11 ns Started: 11 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Error: assertion failed
#    Time: 13 ns Started: 13 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Error: assertion failed
#    Time: 15 ns Started: 5 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Info: assertion passed
#    Time: 25 ns Started: 15 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Info: assertion passed
#    Time: 27 ns Started: 17 ns  Scope: MARCO_AS_ASSERTION File:
testbench.sv Line: 59
# ** Note: $finish    : testbench.sv(69)
#    Time: 29 ns  Iteration: 0  Instance: /MARCO_AS_ASSERTION
# End time: 05:20:25 on Mar 11,2023, Elapsed time: 0:00:02
# Errors: 7, Warnings: 0
```

5. MIX USE OF MACROS AND PARAMETERS

Mixing macros and parameters can help to reduce errors in the code by making it easier to maintain and update. By reducing the amount of manual coding required, the likelihood of errors is reduced, and the overall quality of the code can be improved.

5.1. Using Macros in Parameters

Macros will get executed in the pre-compilation stage and parameters will be evaluated in elaboration time. In precompilation time macros are replaced by the text and get executed then parameters will execute in elaboration time.

```
`define DATA 56
parameter data_p = `DATA+2;
module mod;
int x;
initial begin
  $display("MACROS VALUE = %0d",`DATA);
  if(x==data_p%2)
  begin
  $display("The data is divisible by 2");
end
end
endmodule
```

In the above code,`DATA is replaced by 56 in pre-compilation time and then parameter data_p executed in elaboration time and takes the value of 56+2=58. After simulation, the output is shown below:

Code after pre-compiler stage:

```
parameter data_p = 56+2; // macro replaced with value
module mod;
int x;
initial begin
  $display("MACROS VALUE = %0d",56);
  if(x==data_p%2)
  begin
  $display("The data is divisible by 2");
end
end
endmodule
```

Code after elaboration stage:

```
parameter data_p = 58; // final evaluation done
module mod;
int x;
initial begin
  $display("MACROS VALUE = %0d",56);
  if(x==data_p%2)
  begin
  $display("The data is divisible by 2");
 end
end
endmodule
```

Output

```
# Value of data = 56
# The data is divisible by 2
```

5.2. Using Macros within Macros

As depicted in the below code, each macro will be replaced by its value and the value can be another reference to a macro. The macros **Boy** and **Dear** are using another macro **C**.

```
`define A "welcome"
`define NAME "Hari"
`define C `NAME
`define Boy `"A, you are here`"
`define Dear(A) `"`C, Hi A, How are you`"
module test#(num=0);
  initial begin
    $display(`Boy);
    $display(`Dear(Shiva));
  end
endmodule
```

During pre-compilation stage1:

```
`define A "welcome"
`define NAME "Hari"
`define C NAME
`define Boy "welcome, you are here" // Used the macro A
`define Dear(A) `"NAME, Hi A, How are you`" // Used macro C
module test#(num=0);
  initial begin
    $display(`Boy);
    $display(`Dear(Shiva));
  end
endmodule
```

During pre-compilation stage2:

```
`define A "welcome"
`define NAME "Hari"
`define C NAME
`define Boy "welcome, you are here"
`define Dear(A) "Hari, Hi A, How are you" // Used macro NAME
module test#(num=0);
  initial begin
    $display(`Boy);
    $display(`Dear(Shiva));
  end
endmodule
```

After pre-compilation:

```
`define A "welcome"
`define NAME "Hari"

`define C NAME
`define Boy "welcome, you are here"
`define Dear(A) "Hari, Hi A, How are you" // Used macro NAME

module test#(num=0);
  initial begin
    // macro Boy was replaced
    $display("welcome, you are here");
    // macro Dear with Shiva was used
    $display("Hari, Hi Shiva, How are you");
end
endmodule
```

Output

```
# welcome, you are here
# Hari, Hi Shiva, How are you
```

5.3. Using Parameter within Macros

Since the macro is replaced in the pre-compiler stage and the parameter is evaluated in the elaboration stage, we can use them together.

```
parameter PCIE_MAP_START = 32'h3000_1000;
parameter PCIE_MAP_END   = 32'h3000_4000;

`define TB_MEMORY_MAP_PCIE_START PCIE_MAP_START
`define TB_MEMORY_MAP_PCIE_END PCIE_MAP_END

module test;
  int check_value;
  initial begin
    if(check_value >= `TB_MEMORY_MAP_PCIE_START &&
       check_value <  `TB_MEMORY_MAP_PCIE_END) begin
       $display("Within the range");
    end
  end
endmodule
```

Code after the pre-compiler stage:

```
parameter PCIE_MAP_START = 32'h3000_1000;
parameter PCIE_MAP_END   = 32'h3000_4000;

`define TB_MEMORY_MAP_PCIE_START PCIE_MAP_START
`define TB_MEMORY_MAP_PCIE_END PCIE_MAP_END

module test;
  int check_value;
  initial begin
    // Replaced with macro value
    if(check_value >= PCIE_MAP_START &&
       check_value <  PCIE_MAP_END) begin
       $display("Within the range");
    end
  end
endmodule
```

As depicted in the above code, after the pre-compilation stage the macros TB_MEMORY_MAP_PCIE_START and TB_MEMORY_MAP_PCIE_END are replaced with its value, which is in-turn the parameter name.

Code after the elaboration stage:

```
parameter PCIE_MAP_START = 32'h3000_1000;
parameter PCIE_MAP_END   = 32'h3000_4000;

`define TB_MEMORY_MAP_PCIE_START PCIE_MAP_START
`define TB_MEMORY_MAP_PCIE_END PCIE_MAP_END

module test;
  int check_value;
  initial begin
    // Replaced with parameter value
    if(check_value >= 32'h3000_1000 &&
      check_value < 32'h3000_4000) begin
      $display("Within the range");
    end
  end
endmodule
```

As depicted above, during the elaboration stage, the parameter value is evaluated and used.

5.4. Using parameter within parameter

```
parameter DATA_WIDTH = 64;
parameter STROBE_WIDTH = DATA_WIDTH/8;

interface axi_intf;
  logic [DATA_WIDTH-1:0] AWDATA;
  logic [STROBE_WIDTH-1:0] AWSTROBE;
endinterface
```

As described in the above code, the final value of the STROBE_WIDTH will be evaluated during the elaboration phase and the resulted code will be as shown below:

```
parameter DATA_WIDTH = 64;
parameter STROBE_WIDTH = DATA_WIDTH/8; // 64/8 = 8

interface axi_intf;
  logic [DATA_WIDTH-1:0] AWDATA;
  logic [STROBE_WIDTH-1:0] AWSTROBE;
endinterface
```

6. BEST PRACTICES WHILE MIXING MACROS AND PARAMETERS

Mixing macros and parameters can create some challenges in HDL programming. For example, macros can obscure the values of parameters and make it difficult to determine how they are being used in the code. Additionally, changing a parameter value can require modifying multiple macro definitions, which can be time-consuming and error-prone. To overcome these challenges, it's important to follow some best practices when using macros and parameters together.

1. Using named parameters in macro definitions can make code more readable and easier to understand, especially for others who might be reviewing or modifying the code. It also helps prevent errors that can arise from using incorrect parameter values in the macro call. Additionally, named parameters provide a way to document the purpose of each parameter, which can be helpful for future reference.
2. Defining all parameters in a single location, such as a header file or a top-level module, makes it easier to modify their values as needed. This also ensures that all instances of a parameter use the same value throughout the design.
3. Use default values for parameters: When defining a macro with multiple parameters, it can be useful to provide default values for some or all of the parameters. This can make it easier to use the macro in a variety of scenarios without having to specify every parameter every time.
4. Use parameter constraints: If a parameter has a limited range of valid values, it can be helpful to define constraints on the parameter to prevent errors. For example, if a parameter must be a power of 2, a constraint can be added to ensure that the parameter is always a power of 2.
5. Use consistent naming conventions: When defining macros and parameters, it's important to use consistent naming conventions to make the code easier to read and understand. For example, using a prefix or suffix for all macro names and parameter names can make it clear that these elements are related.

6. Use comments to explain macros and parameters: When defining macros and parameters, it's important to include comments that explain what they do and how they are used. This can help other programmers understand the code and make modifications more easily.

The below code demonstrates and incorporates the six best practices we discussed above:

```
`define WIDTH 8 // define parameter with default value
`define ENABLE // define macro without parameters

module my_module #(parameter DATA_SIZE = 16) (input clk, input reset,
input [`WIDTH-1:0] data_in, output reg [`WIDTH-1:0] data_out);

// use of named parameters in macro definitions
`ifdef ENABLE
  always @(posedge clk) begin
    // use of parameter in procedural code
    if (reset) begin
      data_out <= {`WIDTH{1'b0}};
    end else begin
      data_out <= data_in + DATA_SIZE;
    end
  end
`endif

// define all parameters in a single location
localparam MAX_SIZE = 1024;

// use of parameter constraints
parameter MIN_SIZE = 8;
parameter MAX_VALUE = 255;
parameter MIN_VALUE = 0;
parameter [`WIDTH-1:0] MASK = {`WIDTH{1'b1}};

// use consistent naming conventions
`define PREFIX_PARAM "MY_PARAM_"
parameter [`WIDTH-1:0] MY_PARAM_DATA_IN = 8'hAA;
parameter MY_PARAM_ENABLE = 1'b1;

// use comments to explain macros and parameters
/*
 * This macro is used to enable or disable the data processing logic.
 * If ENABLE is defined, the data_out signal will be updated on every
clock cycle.
 */
`define ENABLE

endmodule
```

In this example, we have defined a parameter DATA_SIZE with a default value of 16, and a macro ENABLE without parameters. We have also used named parameters in the macro

definition and defined all parameters in a single location using localparam. We have added parameter constraints to MIN_SIZE, MAX_VALUE, MIN_VALUE, and MASK, and used consistent naming conventions by adding a prefix to all parameter names using PREFIX_PARAM. Finally, we have used comments to explain the purpose of the ENABLE macro and the parameters in the code.

7. CONCLUSIONS

In summary, the use of macros and parameters in HDL programming provides a powerful means of creating efficient and reusable code. Through the examination of the advantages, real-world use cases, and best practices of macros and parameters, this paper has demonstrated their potential to enhance the flexibility, readability, and maintainability of HDL code. The ability to quickly and easily modify values through parameters and generate code on-the-fly with macros can greatly improve the efficiency of hardware design and verification processes. Additionally, by implementing best practices such as defining parameters in a single location and using macros sparingly, programmers can ensure that their code is maintainable and scalable. As designs grow more complex, it is vital for HDL programmers to have an in-depth knowledge of macros and parameters to effectively utilize these powerful tools. By fully exploring their capabilities, HDL programmers can develop more efficient and effective hardware designs and verification.

REFERENCES

[1] "Verilog HDL: A Guide to Digital Design and Synthesis" by Samir Palnitkar
[2] "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features" by Chris Spear and Greg Tumbush
[3] "ASIC Design in the Silicon Sandbox: A Complete Guide to Building Mixed-Signal Integrated Circuits" by Keith Barr
[4] "The SystemVerilog Language Reference Manual" by Accellera Systems Initiative

AUTHORS

Muneeb Ulla Shariff, Staff Design Verification Engineer at Mirafra Technologies. Bachelor of Engineering from B.M.S Institute of Technology, Bangalore in 2013. Research Interests are open-source design verification methodologies, RISCv and low power design-verification. muneebullashariff@mirafra.com

Vineeth Kumar Veepuri, Verification Engineer at Mirafra Technologies. Bachelor of Technology from Jawaharlal Nehru Technological University Ananthapuramu in 2022. Research Interests are HDL programming. vineethkumarvms@gmail.com

Nancy Dimri, Verification Engineer at Mirafra Technologies. Bachelor of Technology from Uttarakhand Technical University. Interests to research in HDL programming. nancydimri@mirafra.com

Mahadevaswamy B N, Verification Engineer at Mirafra Technologies. Bachelor of Technology from Visveswaraiah Technological University. HDL programming is the area of interest for research. mahadevaswamy@mirafra.com