**ASSESSMENT 17**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | 05-06-2020 | **Name:** | Sheela Golasangi |
| **Course:** | Digital Design using HDL | **USN:** | 4AL16EC068 |
| **Topic:** | 1. Verilog Tutorials and practice programs 2. Building/ Demo projects using FPGA | **Semester & Section:** | VIII  ‘B’ |
| **Github Repository:** | Sheela - Course |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of the session** |
| **REPORT**  **Verilog Tutorials and practice programs**  **Introduction to Verilog:**  Verilog is a **HARDWARE DESCRIPTION LANGUAGE (HDL)**. A hardware description Language is a language used to describe a digital system, for example, a network switch, a microprocessor or a memory or a simple flip−flop. This just means that, by using a HDL one can describe any hardware (digital) at any level.    1// D flip−flop Code  2module d\_ff ( d, clk, q, q\_bar); 3input d ,clk;  4output q, q\_bar; 5wire d ,clk;  6reg q, q\_bar; 7  8always @ (posedge clk) 9begin  10 q <= d;  11 q\_bar <= !d; 12end  13  14endmodule  One can describe a simple Flip flop as that in above figure as well as one can describe complicated designs having 1 million gates. Verilog is one of the HDL languages available in the industry for designing the Hardware. Verilog allows us to design a Digital design at Behavior Level, Register Transfer Level (RTL), and Gate level and at switch level. Verilog allows hardware designers to express their designs with behavioral constructs, deterring the details of implementation to a later stage of design in the final design.  Many engineers who want to learn Verilog, most often ask this question, how much time it will take to learn Verilog?, Well my answer to them is "It may not take more than one week, if you happen to know at least one programming language".  **DESIGN AND TOOL FLOW**  **Various stages of ASIC/FPGA**   * **Specification:** Word processor like Word, Kwriter, AbiWord, Open Office. * **High Level Design:** Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word, Open Office. * **Micro Design/Low level design:** Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word. For FSM State CAD or some similar tool, Open Office. * **RTL Coding :** Vim, Emacs, conTEXT, HDL TurboWriter * **Simulation:** Modelsim, VCS, Verilog−XL, Veriwell, Finsim, iVerilog, VeriDOS. * **Synthesis:** Design Compiler, FPGA Compiler, Synplify, Leonardo Spectrum. You can download this from FPGA vendors like Altera and Xilinx for free. * **Place & Route:** For FPGA use FPGA' vendors P&R tool. ASIC tools require expensive P&R tools like Apollo. Students can use LASI, Magic. * **Post Si Validation:** For ASIC and FPGA, the chip needs to be tested in real environment. Board design, device drivers needs to be in place.   **Programs:**  **Hello world Program**    If you refer to any book on programming language it starts with "hello World" program, once you have written the program, you can be sure that you can do something in that language .  Well I am also going to show how to write a **"hello world"** program in Verilog, followed by **"counter"** design in Verilog.  Words in green are comments, blue are reserved words, Any program in Verilog starts with reserved word module, In the above example line 7 contains module hello\_world.  module hello\_world ;  initial begin  $display ( "Hello World by Sheela" );  # 10 $finish;  end  endmodule  Line 9 contains the initial block, this block gets executed only once after the simulation starts and at time=0 (0ns). This block contains two statements, which are enclosed within begin at line 7 and end at line 12. In Verilog if you have multiple lines within a block, you need to use begin and end.  **Hello world output**  Hello World by Sheela    **Abstraction Levels of Verilog**  Verilog supports a design at many different levels of abstraction. Three of them are very important:  • Behavioral level  • Register−Transfer Level  • Gate Level  **Behavioral level**  This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.  Register−Transfer Level Designs using the Register−Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing possibility, operations are scheduled to occur at certain times. Modern definition of a RTL code is "Any code that is synthesizable is called RTL code".  **Gate Level**  Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.  **Data Types**  Verilog Language has two primary data types  • Nets − represents structural connections between components.  • Registers − represent variables used to store data.  Every signal has a data type associated with it:  • Explicitly declared with a declaration in your Verilog code.  Implicitly declared with no declaration but used to connect structural building blocks in your code.  •Implicit declaration is always a net type "wire" and is one bit wide.  **Types of Nets**  Each net type has functionality that is used to model different types of hardware (such as PMOS,NMOS, CMOS, etc)    **Register Data Types**  Registers store the last value assigned to them until another assignment statement  changes their value.  • Registers represent data storage constructs.  • You can create arrays of the regs called memories.  • register data types are used as variables in procedural blocks.  • A register data type is required if a signal is assigned a value within a procedural block  • Procedural blocks begin with keyword initial and always.    Some of the FPGA projects can be FPGA tutorials such as [What is FPGA Programming](https://www.fpga4student.com/2017/08/what-is-fpga-programming.html), [image processing on FPGA](https://www.fpga4student.com/2016/11/image-processing-on-fpga-verilog.html), [matrix multiplication](https://www.fpga4student.com/2016/11/matrix-multiplier-core-design.html) on FPGA Xilinx using Core Generator, [Verilog vs VHDL: Explain by Examples](https://www.fpga4student.com/2017/08/verilog-vs-vhdl-explain-by-example.html) and [how to load text files or images into FPGA](https://www.fpga4student.com/2016/11/two-ways-to-load-text-file-to-fpga-or.html). Many others FPGA projects provide students with full Verilog/ VHDL source code to practice and run on FPGA boards. Some of them can be used for another bigger FPGA projects.  **Building/ Demo projects using FPGA**  **Image processing on FPGA using Verilog HDL** This FPGA project is aimed to show in details how to process an image using Verilog from reading an input bitmap image (.bmp) in Verilog, processing and writing the processed result to an output bitmap image in Verilog. The full Verilog code for reading image, image processing, and writing image is provided. Image processing on FPGA using Verilog HDL    In this FPGA Verilog project, some simple processing operations are implemented in Verilog such as inversion, brightness control and threshold operations. The image processing operation is selected by a "parameter.v" file and then, the processed image data are written to a bitmap image output.bmp for verification purposes. The image reading Verilog code operates as a Verilog model of an image sensor/ camera, which can be really helpful for functional verifications in real-time FPGA image processing projects. The image writing part is also extremely useful for testing as well when you want to see the output image in BMP format. In this project, I added some simple image processing code into the reading part to make an example of image processing, but you can easily remove it to get raw image data. All the related questions asked by students are answered at the bottom of this article. First of all, Verilog cannot read images directly. To read the .bmp image on in Verilog, the image is required to be converted from the bitmap format to the hexadecimal format. Below is a Matlab example code to convert a bitmap image to a .hex file. The input image size is 768x512 and the image .hex file includes R, G, B data of the bitmap image.  b=imread('kodim24.bmp');  k=**1**;  **for** i=**512**:-**1**:**1**  **for** j=**1**:**768**  a(k)=b(i,j,**1**);  a(k+**1**)=b(i,j,**2**);  a(k+**2**)=b(i,j,**3**);  k=k+**3**;  **end**  **end**  fid = fopen('kodim24.hex', 'wt');  fprintf(fid, '%x\n', a);  disp('Text file write done');disp(' ');  fclose(fid);  To read the image hexadecimal data file, Verilog uses this command: $readmemh or $readmemb if the image data is in a binary text file. After reading the image .hex file, the RGB image data are saved into memory and read out for processing. Below is the Verilog code to the image reading and processing part: **`include** "parameter.v"  // fpga4student.com: [FPGA projects](https://www.fpga4student.com/p/fpga-projects.html) for students  // [FPGA project](https://www.fpga4student.com/p/fpga-projects.html): Image processing in Verilog  **module** image\_read  #(  **parameter** **WIDTH** = **768**,  **HEIGHT** = **512**,  **INFILE** = "./img/kodim01.hex",  **START\_UP\_DELAY** = **100**,  **HSYNC\_DELAY** = **160**  **VALUE**= **100**,  **THRESHOLD**= **90**,  **SIGN**=**1**  )  (  **input** **HCLK**,  **input** HRESETn,  **output** **VSYNC**,  **output** **reg** **HSYNC**,  **output** **reg** [**7**:**0**] **DATA\_R0**,  **output** **reg** [**7**:**0**] **DATA\_G0**,  **output** **reg** [**7**:**0**] **DATA\_B0**,  **output** **reg** [**7**:**0**] **DATA\_R1**,  **output** **reg** [**7**:**0**] **DATA\_G1**,  **output** **reg** [**7**:**0**] **DATA\_B1**,  **output** ctrl\_done  );  **parameter** sizeOfWidth = **8**;  **parameter** sizeOfLengthReal = **1179648**;  **localparam** **ST\_IDLE** = **2'b00**,  **ST\_VSYNC** = **2'b01**,  **ST\_HSYNC** = **2'b10**,  **ST\_DATA** = **2'b11**;  **reg** [**1**:**0**] cstate,  nstate;  **reg** start;  **reg** HRESETn\_d;  **reg** ctrl\_vsync\_run;  **reg** [**8**:**0**] ctrl\_vsync\_cnt;  **reg** ctrl\_hsync\_run;  **reg** [**8**:**0**] ctrl\_hsync\_cnt;  **reg** ctrl\_data\_run;  **reg** [**7** : **0**] total\_memory [**0** : sizeOfLengthReal-**1**];  **integer** temp\_BMP [**0** : **WIDTH**\***HEIGHT**\***3** - **1**];  **integer** org\_R [**0** : **WIDTH**\***HEIGHT** - **1**];  **integer** org\_G [**0** : **WIDTH**\***HEIGHT** - **1**];  **integer** org\_B [**0** : **WIDTH**\***HEIGHT** - **1**];  **integer** i, j;  **integer** tempR0,tempR1,tempG0,tempG1,tempB0,tempB1;  **integer** value,value1,value2,value4**reg** [ **9**:**0**] row;  **reg** [**10**:**0**] col;  **reg** [**18**:**0**] data\_count;  **initial** **begin**  $readmemh(**INFILE**,total\_memory,**0**,sizeOfLengthReal-**1**);  **end**  **always**@(start) **begin**  **if**(start == **1'b1**) **begin**  **for**(i=**0**; i<**WIDTH**\***HEIGHT**\***3** ; i=i+**1**) **begin**  temp\_BMP[i] = total\_memory[i+**0**][**7**:**0**];  **end**    **for**(i=**0**; i<**HEIGHT**; i=i+**1**) **begin**  **for**(j=**0**; j<**WIDTH**; j=j+**1**) **begin**    org\_R[**WIDTH**\*i+j] = temp\_BMP[**WIDTH**\***3**\*(**HEIGHT**-i-**1**)+**3**\*j+**0**]; org\_G[**WIDTH**\*i+j] = temp\_BMP[**WIDTH**\***3**\*(**HEIGHT**-i-**1**)+**3**\*j+**1**  org\_B[**WIDTH**\*i+j] = temp\_BMP[**WIDTH**\***3**\*(**HEIGHT**-i-**1**)+**3**\*j+**2** **end**  **end**  **end**  **end**  **always**@(**posedge** **HCLK**, **negedge** HRESETn)  **begin**  **if**(!HRESETn) **begin**  start <= **0**;  HRESETn\_d <= **0**;  **end**  **else** **begin**  HRESETn\_d <= HRESETn;  **if**(HRESETn == **1'b1** && HRESETn\_d == **1'b0**)  start <= **1'b1**;  **else**  start <= **1'b0**;  **end**  **end**  **always**@(**posedge** **HCLK**, **negedge** HRESETn)  **begin**  **if**(~HRESETn) **begin**  cstate <= **ST\_IDLE**;  **end**  **else** **begin**  cstate <= nstate;  **end**  **end**  **always** @(\*) **begin**  **case**(cstate)  **ST\_IDLE:** **begin**  **if**(start)  nstate = **ST\_VSYNC**;  **else**  nstate = **ST\_IDLE**;  **end**  **ST\_VSYNC:** **begin**  **if**(ctrl\_vsync\_cnt == **START\_UP\_DELAY**)  nstate = **ST\_HSYNC**;  **else**  nstate = **ST\_VSYNC**;  **end**  **ST\_HSYNC:** **begin**  **if**(ctrl\_hsync\_cnt == **HSYNC\_DELAY**)  nstate = **ST\_DATA**;  **else**  nstate = **ST\_HSYNC**;  **end**  **ST\_DATA:** **begin**  **if**(ctrl\_done)  nstate = **ST\_IDLE**;  **else** **begin**  **if**(col == **WIDTH** - **2**)  nstate = **ST\_HSYNC**;  **else**  nstate = **ST\_DATA**;  **end**  **end**  **endcase**  **end**  **always** @(\*) **begin**  ctrl\_vsync\_run = **0**;  ctrl\_hsync\_run = **0**;  ctrl\_data\_run = **0**;  **case**(cstate)  **ST\_VSYNC:** **begin** ctrl\_vsync\_run = **1**; **end**  **ST\_HSYNC:** **begin** ctrl\_hsync\_run = **1**; **end**  **ST\_DATA:** **begin** ctrl\_data\_run = **1**; **end**  **endcase**  **end**  **always**@(**posedge** **HCLK**, **negedge** HRESETn)  **begin**  **if**(~HRESETn) **begin**  ctrl\_vsync\_cnt <= **0**;  ctrl\_hsync\_cnt <= **0**;  **end**  **else** **begin**  **if**(ctrl\_vsync\_run)  ctrl\_vsync\_cnt <= ctrl\_vsync\_cnt + **1**;  **else**  ctrl\_vsync\_cnt <= **0**;    **if**(ctrl\_hsync\_run)  ctrl\_hsync\_cnt <= ctrl\_hsync\_cnt + **1**;  **else**  ctrl\_hsync\_cnt <= **0**;  **end**  **end**  **always**@(**posedge** **HCLK**, **negedge** HRESETn)  **begin**  **if**(~HRESETn) **begin**  row <= **0**;  col <= **0**;  **end**  **else** **begin**  **if**(ctrl\_data\_run) **begin**  **if**(col == **WIDTH** - **2**) **begin**  row <= row + **1**;  **end**  **if**(col == **WIDTH** - **2**)  col <= **0**;  **else**  col <= col + **2**;  **end**  **end**  **end**  **always**@(**posedge** **HCLK**, **negedge** HRESETn)  **begin**  **if**(~HRESETn) **begin**  data\_count <= **0**;  **end**  **else** **begin**  **if**(ctrl\_data\_run)  data\_count <= data\_count + **1**;  **end**  **end**  **assign** **VSYNC** = ctrl\_vsync\_run;  **assign** ctrl\_done = (data\_count == **196607**)? **1'b1**: **1'b0**;  **always** @(\*) **begin**    **HSYNC** = **1'b0**;  **DATA\_R0** = **0**;  **DATA\_G0** = **0**;  **DATA\_B0** = **0**;  **DATA\_R1** = **0**;  **DATA\_G1** = **0**;  **DATA\_B1** = **0**;  **if**(ctrl\_data\_run) **begin**    **HSYNC** = **1'b1**;  **`ifdef** **BRIGHTNESS\_OPERATION**    **if**(**SIGN** == **1**) **begin**    tempR0 = org\_R[**WIDTH** \* row + col ] + **VALUE**;  **if** (tempR0 > **255**)  **DATA\_R0** = **255**;  **else**  **DATA\_R0** = org\_R[**WIDTH** \* row + col ] + **VALUE**;    tempR1 = org\_R[**WIDTH** \* row + col+**1** ] + **VALUE**;  **if** (tempR1 > **255**)  **DATA\_R1** = **255**;  **else**  **DATA\_R1** = org\_R[**WIDTH** \* row + col+**1** ] + **VALUE**;    tempG0 = org\_G[**WIDTH** \* row + col ] + **VALUE**;  **if** (tempG0 > **255**)  **DATA\_G0** = **255**;  **else**  **DATA\_G0** = org\_G[**WIDTH** \* row + col ] + **VALUE**;  tempG1 = org\_G[**WIDTH** \* row + col+**1** ] + **VALUE**;  **if** (tempG1 > **255**)  **DATA\_G1** = **255**;  **else**  **DATA\_G1** = org\_G[**WIDTH** \* row + col+**1** ] + **VALUE**;  // B  tempB0 = org\_B[**WIDTH** \* row + col ] + **VALUE**;  **if** (tempB0 > **255**)  **DATA\_B0** = **255**;  **else**  **DATA\_B0** = org\_B[**WIDTH** \* row + col ] + **VALUE**;  tempB1 = org\_B[**WIDTH** \* row + col+**1** ] + **VALUE**;  **if** (tempB1 > **255**)  **DATA\_B1** = **255**;  **else**  **DATA\_B1** = org\_B[**WIDTH** \* row + col+**1** ] + **VALUE**;  **end**  **else** **begin**    tempR0 = org\_R[**WIDTH** \* row + col ] - **VALUE**;  **if** (tempR0 < **0**)  **DATA\_R0** = **0**;  **else**  **DATA\_R0** = org\_R[**WIDTH** \* row + col ] - **VALUE**;    tempR1 = org\_R[**WIDTH** \* row + col+**1** ] - **VALUE**;  **if** (tempR1 < **0**)  **DATA\_R1** = **0**;  **else**  **DATA\_R1** = org\_R[**WIDTH** \* row + col+**1** ] - **VALUE**;    tempG0 = org\_G[**WIDTH** \* row + col ] - **VALUE**;  **if** (tempG0 < **0**)  **DATA\_G0** = **0**;  **else**  **DATA\_G0** = org\_G[**WIDTH** \* row + col ] - **VALUE**;  tempG1 = org\_G[**WIDTH** \* row + col+**1** ] - **VALUE**;  **if** (tempG1 < **0**)  **DATA\_G1** = **0**;  **else**  **DATA\_G1** = org\_G[**WIDTH** \* row + col+**1** ] - **VALUE**;    tempB0 = org\_B[**WIDTH** \* row + col ] - **VALUE**;  **if** (tempB0 < **0**)  **DATA\_B0** = **0**;  **else**  **DATA\_B0** = org\_B[**WIDTH** \* row + col ] - **VALUE**;  tempB1 = org\_B[**WIDTH** \* row + col+**1** ] - **VALUE**;  **if** (tempB1 < **0**)  **DATA\_B1** = **0**;  **else**  **DATA\_B1** = org\_B[**WIDTH** \* row + col+**1** ] - **VALUE**;  **end**  **`endif**    **`ifdef** **INVERT\_OPERATION**  value2 = (org\_B[**WIDTH** \* row + col ] + org\_R[**WIDTH** \* row + col ] +org\_G[**WIDTH** \* row + col ])/**3**;  **DATA\_R0**=**255**-value2;  **DATA\_G0**=**255**-value2;  **DATA\_B0**=**255**-value2;  value4 = (org\_B[**WIDTH** \* row + col+**1** ] + org\_R[**WIDTH** \* row + col+**1** ] +org\_G[**WIDTH** \* row + col+**1** ])/**3**;  **DATA\_R1**=**255**-value4;  **DATA\_G1**=**255**-value4;  **DATA\_B1**=**255**-value4;  **`endif**    **`ifdef** **THRESHOLD\_OPERATION**  value = (org\_R[**WIDTH** \* row + col ]+org\_G[**WIDTH** \* row + col ]+org\_B[**WIDTH** \* row + col ])/**3**;  **if**(value > **THRESHOLD**) **begin**  **DATA\_R0**=**255**;  **DATA\_G0**=**255**;  **DATA\_B0**=**255**;  **end**  **else** **begin**  **DATA\_R0**=**0**;  **DATA\_G0**=**0**;  **DATA\_B0**=**0**;  **end**  value1 = (org\_R[**WIDTH** \* row + col+**1** ]+org\_G[**WIDTH** \* row + col+**1** ]+org\_B[**WIDTH** \* row + col+**1** ])/**3**;  **if**(value1 > **THRESHOLD**) **begin**  **DATA\_R1**=**255**;  **DATA\_G1**=**255**;  **DATA\_B1**=**255**;  **end**  **else** **begin**  **DATA\_R1**=**0**;  **DATA\_G1**=**0**;  **DATA\_B1**=**0**;  **end**  **`endif**  **end**  **end**  **endmodule**  The image processing operation is selected in the following "parameter.v" file. To change the processing operation, just switch the comment line.  `define INPUTFILENAME "your\_image.hex"  `define OUTPUTFILENAME "output.bmp"  `define INVERT\_OPERATION  The "parameter.v" file is also to define paths and names of the input and output file. After processing the image, it is needed to write the processed data to an output image for verifications. The following Verilog code is to write the processed image data to a bitmap image for verification: **module** image\_write #(**parameter**  **WIDTH** = **768**,  **HEIGHT** = **512**,  **INFILE** = "output.bmp",  **BMP\_HEADER\_NUM** = **54**  )  (  **input** **HCLK**,  HRESETn,  **input** hsync,  **input** [**7**:**0**] **DATA\_WRITE\_R0**,  **input** [**7**:**0**] **DATA\_WRITE\_G0**,  **input** [**7**:**0**] **DATA\_WRITE\_B0**,  **input** [**7**:**0**] **DATA\_WRITE\_R1**,  **input** [**7**:**0**] **DATA\_WRITE\_G1**,  **input** [**7**:**0**] **DATA\_WRITE\_B1**,)  **output** **reg** Write\_Done  );  **initial** **begin**  BMP\_header[ **0**] = **66**;BMP\_header[**28**] =**24**;  BMP\_header[ **1**] = **77**;BMP\_header[**29**] = **0**;  BMP\_header[ **2**] = **54**;BMP\_header[**30**] = **0**;  BMP\_header[ **3**] = **0**;BMP\_header[**31**] = **0**;  BMP\_header[ **4**] = **18**;BMP\_header[**32**] = **0**;  BMP\_header[ **5**] = **0**;BMP\_header[**33**] = **0**;  BMP\_header[ **6**] = **0**;BMP\_header[**34**] = **0**;  BMP\_header[ **7**] = **0**;BMP\_header[**35**] = **0**;  BMP\_header[ **8**] = **0**;BMP\_header[**36**] = **0**;  BMP\_header[ **9**] = **0**;BMP\_header[**37**] = **0**;  BMP\_header[**10**] = **54**;BMP\_header[**38**] = **0**;  BMP\_header[**11**] = **0**;BMP\_header[**39**] = **0**;  BMP\_header[**12**] = **0**;BMP\_header[**40**] = **0**;  BMP\_header[**13**] = **0**;BMP\_header[**41**] = **0**;  BMP\_header[**14**] = **40**;BMP\_header[**42**] = **0**;  BMP\_header[**15**] = **0**;BMP\_header[**43**] = **0**;  BMP\_header[**16**] = **0**;BMP\_header[**44**] = **0**;  BMP\_header[**17**] = **0**;BMP\_header[**45**] = **0**;  BMP\_header[**18**] = **0**;BMP\_header[**46**] = **0**;  BMP\_header[**19**] = **3**;BMP\_header[**47**] = **0**;  BMP\_header[**20**] = **0**;BMP\_header[**48**] = **0**;  BMP\_header[**21**] = **0**;BMP\_header[**49**] = **0**;  BMP\_header[**22**] = **0**;BMP\_header[**50**] = **0**;  BMP\_header[**23**] = **2**;BMP\_header[**51**] = **0**;  BMP\_header[**24**] = **0**;BMP\_header[**52**] = **0**;  BMP\_header[**25**] = **0**;BMP\_header[**53**] = **0**;  BMP\_header[**26**] = **1**; BMP\_header[**27**] = **0**;  **end**  **initial** **begin**  fd = $fopen(**INFILE**, "wb+");  **end**  **always**@(Write\_Done) **begin**   **if**(Write\_Done == **1'b1**) **begin**  **for**(i=**0**; i<**BMP\_HEADER\_NUM**; i=i+**1**) **begin**  $fwrite(fd, "%c", BMP\_header[i][**7**:**0**]);  **end**    **for**(i=**0**; i<**WIDTH**\***HEIGHT**\***3**; i=i+**6**) **begin**    $fwrite(fd, "%c", out\_BMP[i ][**7**:**0**]);  $fwrite(fd, "%c", out\_BMP[i+**1**][**7**:**0**]);  $fwrite(fd, "%c", out\_BMP[i+**2**][**7**:**0**]);  $fwrite(fd, "%c", out\_BMP[i+**3**][**7**:**0**]);  $fwrite(fd, "%c", out\_BMP[i+**4**][**7**:**0**]);  $fwrite(fd, "%c", out\_BMP[i+**5**][**7**:**0**]);  **end**  **end**  **end**  The header data for the bitmap image is very important and it is published here. If there is no header data, the written image could not be correctly displayed. In Verilog HDL, $fwrite command is used to write data to file. Next, let's write a test bench Verilog code to verify the image processing operations. **`timescale** **1**ns/**1**ps  **`include** "parameter.v"  **reg** **HCLK**, HRESETn;  **wire** vsync;  **wire** hsync;  **wire** [ **7** : **0**] data\_R0;  **wire** [ **7** : **0**] data\_G0;  **wire** [ **7** : **0**] data\_B0;  **wire** [ **7** : **0**] data\_R1;  **wire** [ **7** : **0**] data\_G1;  **wire** [ **7** : **0**] data\_B1;  **wire** enc\_done;  image\_read #(.**INFILE**(**`INPUTFILENAME**))  u\_image\_read  ( .**HCLK** (**HCLK** ),  .HRESETn (HRESETn ),  .**VSYNC** (vsync ),  .**HSYNC** (hsync ),  .**DATA\_R0** (data\_R0 ),  .**DATA\_G0** (data\_G0 ),  .**DATA\_B0** (data\_B0 ),  .**DATA\_R1** (data\_R1 ),  .**DATA\_G1** (data\_G1 ),  .**DATA\_B1** (data\_B1 ),  .ctrl\_done (enc\_done)  );  image\_write #(.**INFILE**(**`OUTPUTFILENAME**))  u\_image\_write (  .**HCLK**(**HCLK**),  .HRESETn(HRESETn),  .hsync(hsync),  .**DATA\_WRITE\_R0**(data\_R0),  .**DATA\_WRITE\_G0**(data\_G0),  .**DATA\_WRITE\_B0**(data\_B0),  .**DATA\_WRITE\_R1**(data\_R1),  .**DATA\_WRITE\_G1**(data\_G1),  .**DATA\_WRITE\_B1**(data\_B1),  .Write\_Done()  );  **initial**  **begin**  **HCLK** = **0**;  **forever** #**10** **HCLK** = ~**HCLK**;  **end**  **initial**  **begin**  HRESETn = **0**;  #**25** HRESETn = **1**;  **end** **endmodule**  Finally, we have everything to run a simulation to verify the image processing code. Let's use the following image as the input bitmap file:  image processing on FPGA verilog  Input bitmap image  Run the simulation for 6ms, close the simulation and open the output image for checking the result. Followings are the output images which are processed by the selected operations in parameter.v:  image processing on FPGA verilog  Output bitmap image after inverting  image processing on FPGA verilog  Output bitmap image after threshold operation  image processing on FPGA verilog  Output bitmap image after subtracting brightness  Since the reading code is to model an image sensor/camera for simulation purposes, it is recommended not to synthesize the code. If you really want to synthesize the processing code and run this directly on FPGA, you need to replace these image arrays (total\_memory, temp\_BMP, org\_R, org\_B, org\_G) in the code by block memory (RAMs) and design address generators to read image data from the block memory.  **Task**  **code**  module num\_zero(input [15:0]A, output reg [4:0]zeros);  integer i;  always@(A)  begin  zeros=0;  for(i=0;i<16;i=i+1)  zeros=zeros+A[i];  end  endmodule  **test bench code**  module test;  reg [15:0]A;  wire [4:0] zeros;  num\_zero out (.A(A), .zeros(zeros));  initial begin  $dumpfile("dumo.vcd");  $dumpvars(1,test);  A=16'hFFFF; #100;  A=16'hF56F; #100;  A=16'h3FFF; #100;  A=16'h0001; #100;  A=16'hF10F; #100;  A=16'hF822; #100;  A=16'h7ABC; #100;  end  endmodule       |  |  |  |  | | --- | --- | --- | --- | | **Date:** | 05-06-2020 | **Name:** | Sheela Golasangi | | **Course:** | PYTHON | **USN:** | 4AL16EC068 | | **Topic:** | Application 9: Build a Data Collector Web App with PostGreSQL and Flask | **Semester & Section:** | VIII  ‘B’ | | **Github Repository:** | Sheela-Course |  |  |   **Report**      Flask startup and configuration Like most widely used Python libraries, the Flask package is installable from the Python Package Index (PPI). First create a directory to work in (something like flask\_todo is a fine directory name) then install the flask package. You'll also want to install flask-sqlalchemy so your Flask application has a simple way to talk to a SQL database.A good way to get moving is to turn the codebase into an installable Python distribution. At the project's root, create setup.py and a directory called todo to hold the source code. The setup.py should look something like this:  requires = [  'flask',  'flask-sqlalchemy',  'psycopg2',  ]  setup(  name='flask\_todo',  version='0.0',  description='A To-Do List built with Flask',  author='<Your actual name here>',  author\_email='<Your actual e-mail address here>',  keywords='web flask',  packages=find\_packages(),  include\_package\_data=True,  install\_requires=requires  )  This way, whenever you want to install or deploy your project, you'll have all the necessary packages in the requires list. You'll also have everything you need to set up and install the package in sitepackages. For more information on how to write an installable Python distribution, check out the docs on setup.py.Within the tododirectory containing your source code, create an app.py file and a blank \_\_init\_\_.py file. The \_\_init\_\_.py file allows you to import from todo as if it were an installed package. The app.py file will be the application's root. This is where all the Flask application goodness will go, and you'll create an environment variable that points to that file. If you're using pipenv (like I am), you can locate your virtual environment with pipenv --venv and set up that environment variable in your environments activate script. |