IP in Vivado HDL

Leomar Durán
https://github.com/lduran2

Summary

This lab will introduce in detail how to create a custom IP for use in a hardware design. The process is done through Vivado like a normal hardware design, with a few differences such as that it is written as a hardware description in Verilog, and uses a different process to finalize the component for use, rather than generating a bitstream.

The advantage of a hardware IP is that it can be customized for a specific job freeing up the software from complexity, such as the IP that we will create that displays the length of the longest sequence of 1s in a number onto the LED display.

Introduction

So far in the course, all of the components used have been ready packaged components, the Zybo board, the GPIOs, the Pmods and the connectors. After this lab, the student will be able to create a custom Intellectual Property (IP) and use it as a component in a Vivado hardware design.

This experiment builds on Tutorial 4A, which shows how to create an IP using a prepared hardware description. Additionally, we will simulate the hardware in software.

The alternative to this lab is to use smaller components, that can be combined as desired. However, this can get unwieldy as the hardware design gets more complicated. The simulation is very similar to all software we have worked on along the way.

In this program, we will see that the software design is more simplified than the equivalent using a GPIO because it is tailored to custom hardware, which can do some of the heavy lifting.

In this experiment, we will use the 10 integer values `0b011001111001011`, `0b011001111001110`, `0b011001111000111`, `0b011001111001111`, `0b011011111001011`, `0b000000000000000`, `0b011111111111111`, 5446, 9150, and 4889 to analyze the IP created.

We will look at the process of how to update an IP, the custom IP we create, how it fits in the hardware and software designs, the special analysis which involves software simulation of hardware, and the unique problems present in working with a custom IP in Verilog.

Discussion

Updating an IP

The process for updating an existing IP is noteworthy. It is performed as follows

1. In the hardware design, right click the IP to edit. In our case, the IP is named

`led_controller`.

2. Select “Edit in IP Packager”.
3. Open Sources Window > Design Sources > `led_controller_v1_0` > `led_controller_v1_0_S00_AXI_inst`, or the instance of your IP.
4. Edit the source code, for example `led_controller_v1_0_S00_AXI.v`, as need be.
5. Open the Package IP - `led_controller` in the Project Summary Window.
   (a) Go to the File Groups tab.
      i. Merge changes from File Groups Wizard.
   (b) Go to the Review and Package tab.
      i. Verify and click “IP has been modified.”
      ii. Select “Re-Package IP.”
      iii. Reply “Yes” to “Finished packaging ... Do you want to close the project?”

Now you are back to the main project.

1. Run “Show IP Status” in the banner.
2. Rerun in IP Status.
3. Upgrade Selected.
4. Accept to ”Generate”.
5. Rerun again in IP Status.
6. Continue from Validate the Design as usual until after launching the SDK.

After launching the SDK, go to system.mss and Re-generate BSP Sources.

The custom `led_controller` IP

The hardware will accepts an integer. Then it iterates through the lower 15 bits of the integer, counting the number of set bits in a sequence, and returns the maximum size of such sequences. The returned value is written to the LED display on the Zybo board.

The hardware does this by looping through the lowest 15 bits in the register `slv_reg0` start at bit 0, checking whether the current bit was 1. If it was, the count would be increased and compared with the maximum length so far. The larger of the two would become the new maximum. If the current bit was 0, then the count would be reset to 0.

Complete hardware design

The complete hardware design is shown in Fig. 1. This is the same design used in Tutorial 4A. The `led_controller_v1.0` IP can take the place of a GPIO, by making the `LEDs_out` an external connection.

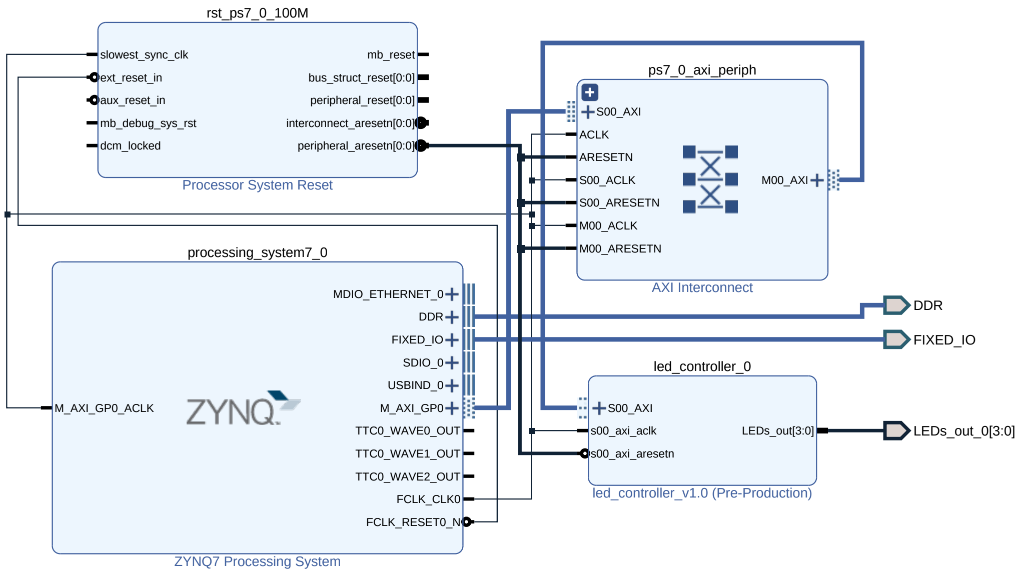


Figure 1: The complete hardware design.

The software

To analyze the hardware design, the template software from Tutorial 4A was adapted to repeat a sequence of 10 integers, rather than the numbers 0 to 15 used in the template.

Like the template all that the software does is pass the values to the hardware in this phrase. No maximum sequence length is sent to the hardware, just each integer.

The first integer used $13\,259$ was given in the lab manual as an example. The next 2 integers are permutations of its bits. The next 2 integers are permutations of the original integer with an additional set bit. The next integer is 0, followed by the allowed maximum value $16\,383$. These were used in earlier tests to check consistent counts. The last three integers were randomly generated within the range $[0, 16383]$ before the software was compiled.

In addition to writing the 10 integers to the hardware, the software performs the same task as the hardware in parallel to verify its results.

The software does this by first masking the current integer to the 15 lowest bits. It then loops, shifting the bits to the right until there are no more bits left. The LSB is checked, and if it is 1, then the count would be increased and compared with the maximum length so far. The larger of the two would become the new maximum. If the LSB was 0, then the count would be reset to 0.

The analysis

From the parallel simulation in the software, we expect the counts to be 4, 4, 4, 4, 5, 0, 14, 2, 5, 2.

The results of the analysis are shown in Table 1.

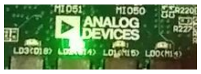
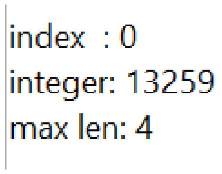
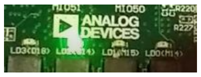
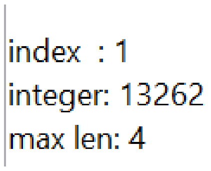
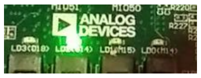
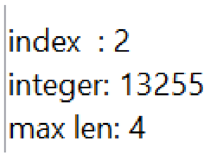
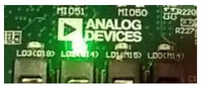
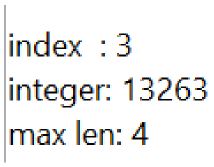
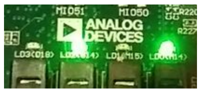
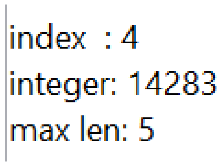
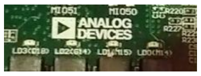
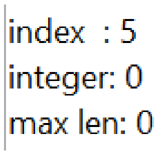
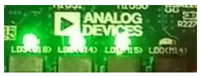
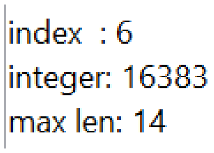
LED display	simulation terminal	binary value	calculated result
	 index : 0 integer: 13259 max len: 4	`14'b11 001 111 001 011`	4
	 index : 1 integer: 13262 max len: 4	`14'b11 001 111 001 110`	4
	 index : 2 integer: 13255 max len: 4	`14'b11 001 111 000 111`	4
	 index : 3 integer: 13263 max len: 4	`14'b11 001 111 001 111`	4
	 index : 4 integer: 14283 max len: 5	`14'b11 011 111 001 011`	5
	 index : 5 integer: 0 max len: 0	`14'b00 000 000 000 000`	0
	 index : 6 integer: 16383 max len: 14	`14'b11 111 111 111 111`	14

Table 1: The results of the analysis (continues in Table 2).

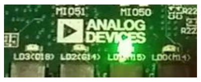
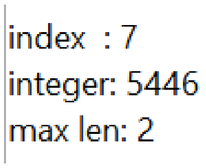
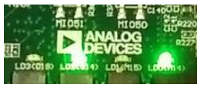
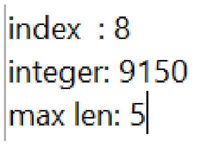
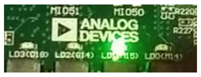
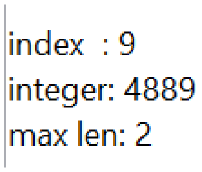
LED display	simulation terminal	binary value	calculated result
	 index : 7 integer: 5446 max len: 2	`14'b01 010 101 000 110`	2
	 index : 8 integer: 9150 max len: 5	`14'b10 001 110 111 110`	5
	 index : 9 integer: 4889 max len: 2	`14'b01 001 100 011 001`	2

Table 2: The results of the analysis (continuing from Table 1).

Problems

Since I was new to the process of creating IPs, but so used to the process of the hardware design, and both used Vivado, I first confused the two, attempting to Generate a Bitstream on the IP, which was interpreted as an incomplete design by Vivado. I got better at this with practice.

When I was writing the hardware description in Verilog, I did not know what to synchronize on, so I tried the positive clock edge `( posedge S_AXI_ACLK )`. I realized that since we needed to update whenever the input `slv_reg0` updated, `slv_reg0` was the value to synchronize on. However, I accidentally left it on `posedge`, which I fixed eventually.

I originally wanted to use a loop such as the one I used in the software in the hardware, but as a `while` loop since I was not familiar with the `for` loop in Verilog. However, the Verilog compiler complained that the loop did not have a consistent end, so the `for` loop fixed from 0 to 15 made more sense.

Finally, I was fixated on the Verilog non-blocking assignment operator $<=$ since I was in sequential logic. However, in this situation, the blocking assignment $=$ worked better. I have to review why that is.

Conclusions

The objective of this project is developing the skill of designing an IP and using it in hardware. Additionally, there is the secondary goal of using software to simulate a hardware component.

As can be seen in Appendix B, since the hardware is specialized for this task, the software is shorter than the equivalent would be, using a GPIO.

This hardware design meets both tasks of the hardware and the simulation in software.

The analysis can be found at `https://youtu.be/NgJu_1vSTZA`.

Appendices

Appendix A: The IP hardware description in Verilog

This hardware description may also be found at `https://github.com/lduran2/ece3623-lab9-ip_in_vivado_hdl/blob/master/ip_repo/led_controller_1.0/hdl/led_controller_v1_0_S00_AXI.v`

```
`timescale 1 ns / 1 ps

    module led_controller_v1_0_S00_AXI #
    (
        // Users to add parameters here

        // User parameters ends
        // Do not modify the parameters beyond this line

        // Width of S_AXI data bus
        parameter integer C_S_AXI_DATA_WIDTH    = 32,
        // Width of S_AXI address bus
        parameter integer C_S_AXI_ADDR_WIDTH    = 4
    )
    (
        // Users to add ports here
        output wire [3:0] LEDs_out,

        // User ports ends
        // Do not modify the ports beyond this line

        // Global Clock Signal
        input wire  S_AXI_ACLK,
        // Global Reset Signal. This Signal is Active LOW
        input wire  S_AXI_ARESETN,
        // Write address (issued by master, acceped by Slave)
        input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
        // Write channel Protection type. This signal indicates the
            // privilege and security level of the transaction, and whether
            // the transaction is a data access or an instruction access.
        input wire [2 : 0] S_AXI_AWPROT,
        // Write address valid. This signal indicates that the master
signaling
            // valid write address and control information.
        input wire  S_AXI_AWVALID,
        // Write address ready. This signal indicates that the slave is ready
            // to accept an address and associated control signals.
        output wire  S_AXI_AWREADY,
        // Write data (issued by master, acceped by Slave)
        input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
        // Write strobes. This signal indicates which byte lanes hold
            // valid data. There is one write strobe bit for each eight
            // bits of the write data bus.
        input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
        // Write valid. This signal indicates that valid write
            // data and strobes are available.
        input wire  S_AXI_WVALID,
        // Write ready. This signal indicates that the slave
            // can accept the write data.
        output wire  S_AXI_WREADY,
        // Write response. This signal indicates the status
            // of the write transaction.
        output wire [1 : 0] S_AXI_BRESP,
        // Write response valid. This signal indicates that the channel
            // is signaling a valid write response.
        output wire  S_AXI_BVALID,
    // Response ready. This signal indicates that the master
        // can accept a write response.
    input wire  S_AXI_BREADY,
    // Read address (issued by master, acceped by Slave)
    input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
    // Protection type. This signal indicates the privilege
        // and security level of the transaction, and whether the
        // transaction is a data access or an instruction access.
    input wire [2 : 0] S_AXI_ARPROT,
    // Read address valid. This signal indicates that the channel
        // is signaling valid read address and control information.
    input wire  S_AXI_ARVALID,
    // Read address ready. This signal indicates that the slave is
        // ready to accept an address and associated control signals.
    output wire  S_AXI_ARREADY,
    // Read data (issued by slave)
    output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
    // Read response. This signal indicates the status of the
        // read transfer.
    output wire [1 : 0] S_AXI_RRESP,
    // Read valid. This signal indicates that the channel is
        // signaling the required read data.
    output wire  S_AXI_RVALID,
    // Read ready. This signal indicates that the master can
        // accept the read data and response information.
    input wire  S_AXI_RREADY
);

// AXI4LITE signals
reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_awaddr;
reg     axi_awready;
reg     axi_wready;
reg [1 : 0]     axi_bresp;
reg     axi_bvalid;
reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_araddr;
reg     axi_arready;
reg [C_S_AXI_DATA_WIDTH-1 : 0]  axi_rdata;
reg [1 : 0]     axi_rresp;
reg     axi_rvalid;

// Example-specific design signals
// local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
// ADDR_LSB is used for addressing 32/64 bit registers/memories
// ADDR_LSB = 2 for 32 bits (n downto 2)
// ADDR_LSB = 3 for 64 bits (n downto 3)
localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
localparam integer OPT_MEM_ADDR_BITS = 1;
//----------------------------------------------
//-- Signals for user logic register space example
//------------------------------------------------
//-- Number of Slave Registers 4
reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg0;
reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg1;
reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg2;
reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg3;
wire     slv_reg_rden;
wire     slv_reg_wren;
```

```
reg [C_S_AXI_DATA_WIDTH-1:0]     reg_data_out;
integer  byte_index;
reg  aw_en;

// used for bit counting algorithm
reg [3:0] i_low_bits;   // index of the low bits
reg [3:0] count;   // current the count so far
reg [3:0] max;     // the maximum count

// I/O Connections assignments

assign S_AXI_AWREADY    = axi_awready;
assign S_AXI_WREADY = axi_wready;
assign S_AXI_BRESP  = axi_bresp;
assign S_AXI_BVALID = axi_bvalid;
assign S_AXI_ARREADY    = axi_arready;
assign S_AXI_RDATA  = axi_rdata;
assign S_AXI_RRESP  = axi_rresp;
assign S_AXI_RVALID = axi_rvalid;
// Implement axi_awready generation
// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_awready <= 1'b0;
      aw_en <= 1'b1;
    end
  else
    begin
      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
        begin
          // slave is ready to accept write address when
          // there is a valid write address and write data
          // on the write address and data bus. This design
          // expects no outstanding transactions.
          axi_awready <= 1'b1;
          aw_en <= 1'b0;
        end
        else if (S_AXI_BREADY && axi_bvalid)
            begin
              aw_en <= 1'b1;
              axi_awready <= 1'b0;
            end
      else
        begin
          axi_awready <= 1'b0;
        end
    end
end

// Implement axi_awaddr latching
// This process is used to latch the address when both
// S_AXI_AWVALID and S_AXI_WVALID are valid.
```

```
    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_awaddr <= 0;
        end
      else
        begin
          if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
            begin
              // Write Address latching
              axi_awaddr <= S_AXI_AWADDR;
            end
        end
    end

    // Implement axi_wready generation
    // axi_wready is asserted for one S_AXI_ACLK clock cycle when both
    // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
    // de-asserted when reset is low.

    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_wready <= 1'b0;
        end
      else
        begin
          if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID && aw_en )
            begin
              // slave is ready to accept write data when
              // there is a valid write address and write data
              // on the write address and data bus. This design
              // expects no outstanding transactions.
              axi_wready <= 1'b1;
            end
          else
            begin
              axi_wready <= 1'b0;
            end
        end
    end

    // Implement memory mapped register select and write logic generation
    // The write data is accepted and written to memory mapped registers when
    // axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
Write strobes are used to
    // select byte enables of slave registers while writing.
    // These registers are cleared when reset (active low) is applied.
    // Slave register write enable is asserted when valid address and data
are available
    // and the slave is ready to accept the write address and write data.
    assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready &&
S_AXI_AWVALID;
```

```
    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          slv_reg0 <= 0;
          slv_reg1 <= 0;
          slv_reg2 <= 0;
          slv_reg3 <= 0;
        end
      else begin
        if (slv_reg_wren)
          begin
            case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
              2'h0:
                for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write
strobes
                    // Slave register 0
                    slv_reg0[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              2'h1:
                for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write
strobes
                    // Slave register 1
                    slv_reg1[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              2'h2:
                for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write
strobes
                    // Slave register 2
                    slv_reg2[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              2'h3:
                for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write
strobes
                    // Slave register 3
                    slv_reg3[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              default : begin
                          slv_reg0 <= slv_reg0;
                          slv_reg1 <= slv_reg1;
                          slv_reg2 <= slv_reg2;
                          slv_reg3 <= slv_reg3;
                        end
              endcase
            end
        end
    end

    // Implement write response logic generation
    // The write response and response valid signals are asserted by the
slave
    // when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are
asserted.
    // This marks the acceptance of address and indicates the status of
    // write transaction.

    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_bvalid  <= 0;
          axi_bresp   <= 2'b0;
        end
      else
        begin
          if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready &&
S_AXI_WVALID)
            begin
              // indicates a valid write response is available
              axi_bvalid <= 1'b1;
              axi_bresp  <= 2'b0; // 'OKAY' response
            end                   // work error responses in future
          else
            begin
              if (S_AXI_BREADY && axi_bvalid)
                //check if bready is asserted while bvalid is high)
                //(there is a possibility that bready is always asserted
high)
                begin
                  axi_bvalid <= 1'b0;
                end
            end
        end
    end

    // Implement axi_arready generation
    // axi_arready is asserted for one S_AXI_ACLK clock cycle when
    // S_AXI_ARVALID is asserted. axi_awready is
    // de-asserted when reset (active low) is asserted.
    // The read address is also latched when S_AXI_ARVALID is
    // asserted. axi_araddr is reset to zero on reset assertion.

    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_arready <= 1'b0;
          axi_araddr  <= 32'b0;
        end
      else
        begin
          if (~axi_arready && S_AXI_ARVALID)
            begin
              // indicates that the slave has acceped the valid read address
              axi_arready <= 1'b1;
              // Read address latching
              axi_araddr  <= S_AXI_ARADDR;
            end
          else
            begin
              axi_arready <= 1'b0;
            end
        end
    end

    // Implement axi_arvalid generation
    // axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
    // S_AXI_ARVALID and axi_arready are asserted. The slave registers
    // data are available on the axi_rdata bus at this instance. The
    // assertion of axi_rvalid marks the validity of read data on the
    // bus and axi_rresp indicates the status of read transaction.axi_rvalid
    // is deasserted on reset (active low). axi_rresp and axi_rdata are
    // cleared to zero on reset (active low).
    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_rvalid <= 0;
          axi_rresp  <= 0;
        end
      else
        begin
          if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
            begin
              // Valid read data is available at the read data bus
              axi_rvalid <= 1'b1;
              axi_rresp  <= 2'b0; // 'OKAY' response
            end
          else if (axi_rvalid && S_AXI_RREADY)
            begin
              // Read data is accepted by the master
              axi_rvalid <= 1'b0;
            end
        end
    end

    // Implement memory mapped register select and read logic generation
    // Slave register read enable is asserted when valid address is available
    // and the slave is ready to accept the read address.
    assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
    always @(*)
    begin
          // Address decoding for reading registers
          case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
            2'h0   : reg_data_out <= slv_reg0;
```

```
            2'h1   : reg_data_out <= slv_reg1;
            2'h2   : reg_data_out <= slv_reg2;
            2'h3   : reg_data_out <= slv_reg3;
            default : reg_data_out <= 0;
          endcase
    end

    // Output register or memory read data
    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_rdata  <= 0;
        end
      else
        begin
          // When there is a valid read address (S_AXI_ARVALID) with
          // acceptance of read address by the slave (axi_arready),
          // output the read dada
          if (slv_reg_rden)
            begin
              axi_rdata <= reg_data_out;     // register read data
            end
        end
    end

    // Add user logic here
    // begin sequential code
    always @( slv_reg0 )
    begin
       // initialize all counts to 0
       count = 0;
       max = 0;
       // right shift bits until there are no more bits
       // to ensure a constant loop, we instead loop through the bits
starting at 0
       for ( i_low_bits = 0; i_low_bits < 15; i_low_bits = i_low_bits+1 )
       begin
           // if current bit is 1, increase the current count
           if (slv_reg0[i_low_bits] == 1'b1)
           begin
               count = count + 1;
               // if the current count exceeds the max, use it as the max
               if (max < count)
               begin
                   max = count;
               end // if (max < count)
           end // if (slv_reg0[i_low_bits] == 1'b1)
           // otherwise reset the count
           else
           begin
               count = 0;
           end // if (slv_reg0[i_low_bits] == 1'b1) else
       end // for ( i_low_bits = 0; i_low_bits < 15; i_low_bits =
i_low_bits+1 )
    end
```

```
assign LEDs_out = max;

// User logic ends

endmodule
```

Appendix B: The software design in C

The software design may also be found at `https://github.com/lduran2/ece3623-lab9-ip_in_vivado_hdl/blob/master/led_controller/led_controller.sdk/led_controller_test/src/led_controller_test_tut_4A.c`

```
/*
/*     led_test.c
 *
       ##############################################################################
#
 *
 *     A simple software application to test the functionality of the led_controller
 *     IP core. The value of a counter is output to the LEDs.
 *
 *
       ##############################################################################
#
 *     v1.1 -- 01/05/2015
 *                  Updated for Zybo ~ DN
 *
 *     v1.0 -- 25/10/2013
 *                  First Version Created
 *
       ##############################################################################
# */

/* Generated driver function for led_controller IP core */
#include "led_controller.h"
#include "xparameters.h"
#include "xil_io.h"

// Number of integers to count the bits
#define N_INTS 10
// Define delay length
#define DELAY 100000000uL
// Blank LED delay is divided by this value
#define BLANK_INC   3


/*     Define the base memory address of the led_controller IP core */
#define LED_BASE XPAR_LED_CONTROLLER_0_S00_AXI_BASEADDR

/* the mask for lowest 15-bits */
#define      LOW_MASK     0b111111111111111

/* @return the length of the longest sequence of set bits in i */
int max_seq_len(u32 i);

/* main function */
int main(void){
       /* index of the integer to count */
       int unsigned i_int = 0;
       /* unsigned 32-bit variables for storing current LED value */
       u32 ints[N_INTS] =
       { 0b011001111001011, 0b011001111001110, 0b011001111000111,
         0b011001111001111, 0b011011111001011, 0b000000000000000,
         0b011111111111111,      // 7 test values
         5446, 9150, 4889        // 3 randomly generated values
       };
       // test values:
       // 3 permutations of ints[0],
       // 2 permutations of ints[0] with an extra 1

       /* 32-bit for longer delay */
       u32 i_delay=0;

       xil_printf("led_controller IP test begin.\r\n");
       xil_printf("--------------------------------------------\r\n\n");

       /* Loop forever */
       while(1){
                       while(i_int<N_INTS){
                               /* Print index to terminal */
                               xil_printf("index  : %d\r\n", i_int);
                               /* Print integer to terminal */
                               xil_printf("integer: %d\r\n", ints[i_int]);
                               /* Print the maximum count to the terminal */
                               xil_printf("max len: %d\r\n\r\n",
max_seq_len(ints[i_int]));

                               /* Write value to led_controller IP core using generated
driver function */
                               LED_CONTROLLER_mWriteReg(LED_BASE, 0, ints[i_int]);
                               /* next integer */
                               i_int++;
                               /* run a simple delay to allow changes on LEDs to be
visible */
                               for(i_delay=0;i_delay<DELAY;i_delay++);

                               /* pause between bit counts to separate them */
                               LED_CONTROLLER_mWriteReg(LED_BASE, 0, 0);
                               for(i_delay=0;i_delay<DELAY;i_delay+=BLANK_INC);
                       }
                       /* Reset LED value to zero */
                       i_int = 0;

                       /* wait on blank between cycles */
                       LED_CONTROLLER_mWriteReg(LED_BASE, 0, 0);
                       for(i_delay=0;i_delay<DELAY;i_delay++);
               }
       return 1;
}

/* find the length of the longest sequence of set bits in i */
int max_seq_len(u32 i) {
       int count = 0;      /* the current count */
       int max = 0; /* the maximum count so far */
       for (u32 low_bits = (i & LOW_MASK);    /* take the 15 lowest bits */
                       low_bits;                          /* loop until no bits left
*/
                       low_bits >>= 1)                    /* arithmetic shifting
right by 1 */
       {
               /* if the LSB is 1, count*/
               if (low_bits & 0b1) {
                    count++;
                    /* if count exceeds max, it replaces max */
                    if (max < count) {
                          max = count;
                    }
              }
              /* otherwise, reset the count */
              else {
                    count = 0;
              }
        }
        /* return the maximum found */
        return max;
}
```