1/26/2022

Christopher Thomas

CST 231 – Digital System Design I

Lab 02

16-bit Counter

Christopher Thomas

Table of Contents

[Abstract 4](#_Toc94351654)

[Introduction 5](#_Toc94351655)

[Design 5](#_Toc94351656)

[Physical hardware design 5](#_Toc94351657)

[Synthesized hardware design 5](#_Toc94351658)

[Design Structure 6](#_Toc94351659)

[Modules 6](#_Toc94351660)

[Simulation and Testing 7](#_Toc94351661)

[Problems 7](#_Toc94351662)

[Results and Conclusion 7](#_Toc94351663)

[Appendix – A 8](#_Toc94351664)

[A-1 Pin Mapping Table 8](#_Toc94351665)

[A-2 RTL Diagrams 9](#_Toc94351666)

[A-2.a clk\_divider 9](#_Toc94351667)

[A-2.b counter 9](#_Toc94351668)

[A-2.c bcd 9](#_Toc94351669)

[A-2.d bcd\_decimal 10](#_Toc94351670)

[A-2.e top module 11](#_Toc94351671)

[A-3 Module Code 11](#_Toc94351672)

[A-3.a clk\_divider 11](#_Toc94351673)

[A-3.b counter 12](#_Toc94351674)

[A-3.c counter 13](#_Toc94351675)

[A-3.d bcd\_decimal 13](#_Toc94351676)

[A-3.e top module 14](#_Toc94351677)

[Table of Figures 16](#_Toc94351678)

[References 17](#_Toc94351679)

# Abstract

For this lab a 16-bit counter was created and converted from a binary value into BCD and that BCD value was then decoded and outputted on 5 seven-segment displays. This was achieved by creating a module that handled each of the following tasks. The on board 50MHz clock being divided so that the counting output could be visible to the human eye. By dividing the signal, the counter could now simply count by 1 each clock signal. Using the double dabble algorithm to convert the 16-bit binary value to a BCD value. And finally decoding the BCD for display on the seven-segment displays. A parameterized double dabble algorithm was essential to the logic of converting a 16-bit number to BCD.

# Introduction

Diagram

Description automatically generated

The goal of this lab is to create a functional 16-bit counter that counts from 0 – 65,536 fast enough that it doesn’t take all day but slow enough to still be visible to anyone looking at the seven-segment displays.

# Design

## Physical hardware design

The DE1 device was the only hardware used for this lab. The onboard seven-segment displays were used, and the rest of the hardware was synthesized onto the board.

## Synthesized hardware design

The synthesized hardware consists of a clock divider module which handles dividing the clock to the speed we desire. This was achieved by using the formula in Figure 1.

*Figure 1: Clock Divider Equation*

The clk\_divider signal drives the counter module which will continuously count from 0 to 65,536 until the reset is activated or the enable is not longer activated. The counter module will then feed into the bcd module which converts that binary value to a bcd value. That is then received by the decimal\_bcd module which takes the bcd value and displays the appropriate decimal value on the seven-segment display.

### Design Structure

Chart

Description automatically generated

The first module had to be the clk\_divider since the on-chip clock is 50MHz which would be far to fast to be visible on the seven-segment displays. The clk\_divider receives its input from the on-chip clock and outputs a slower clock to be used by the rest of the design. Next the clk\_divider feeds the counter with its output and the counter increments its output value on each positive edge of the clk\_divider output. The counter then feeds into the bcd. This is necessary to convert the binary value of count to a bcd value that will be easier to convert to a value that can be displayed by the seven-segment displays. Finally, that bcd value is sent to the bcd\_decimal module which uses a case statement to light up the appropriate portions of the seven-segment display to represent the decimal value needed.

### Modules

#### Module: clk\_divider

This module utilizes a counter that increments each time it receives a positive edge signal from the onboard clock. A value is then determined that will be reached by the counter before it inverts the output signal so that the device connected to the clk\_divider is only receiving a fraction of the signals and thus dividing the clock signal to a slow rate.

#### Module: counter

The counter is parametrized and receives input from the clk\_divider module and increments count on each positive edge which is the output. The counter module also includes an enable and a reset\_n, the enable determines whether the counter will continue to count if enable is low it will count if not it will pause the count. The reset\_n will start the count over if it becomes a low.

#### Module: bcd

The bcd module is parametrized and receives its input from counter. This module utilizes the double dabble algorithm for converting the binary value of the input to an appropriate bcd value. Once the binary value has been converted to the appropriate bcd value it is output.

#### Module: bcd\_decimal

The bcd\_decimal module decodes the bcd value to be used to output appropriate value for the seven-segment displays. This is achieved by using a case statement that takes the bcd value and outputs the appropriate values to the seven-segment display to make the decimal value needed.

# Simulation and Testing

The testing was done by using a testbench for each module in the design other than the clk\_divider. The counter was tested incrementing the value with each clock pulse. The output that is expected is for each time the clock goes high the count would increase by 1. The bcd was tested by having the input incrementing every 5 cycles. The output should be 0-9 and after 9 is reached it should turn back over to 0-9 again. Finally, the bcd\_decimal was tested in the same was the bcd module was tested but the corresponding outputs should be a 7-bit value that represents the segments of the display. The output should directly correspond to the case statements in the module.

# Problems

There was an initial issue with the counter never beginning to count which was later attributed to having a reset input in my clk\_divider module. The reset was not necessary in the clk\_divider module since the clock can run continuously even when the need to reset the counter value arises. All this did was make it, so it was waiting on an input instead of doing what the clock needed to do which was provide repeated input to drive my counter. There was also an issue with the algorithm used for the bcd module initially which led to looking for alternative and eventually the double dabbler algorithm that is currently implemented in the project.

# Results and Conclusion

This lab consisted of 4 modules each one handling a specific piece of the overall design. The clock divider slows down the on-chip clock so that the results will be visible to the human eye. The counter handles the count that is ultimately display out to the seven-segment display. The bcd converts a raw binary value into something better unjderstood for output to the seven-segment display. The bcd\_decimal decode the bcd value and turns on the appropriate segments of the seven-segment display to create the decimal value.

A reset should not be included in the clk\_divider module as a clock does not need to be reset it should just run. The double dabbler algorithm is the best algorithm for converting a binary value to a bcd value and can be easily parameterized to be used with many different projects.

The result of this project requires 1 instantiation of each module except for the seven-segment module which needs 5 individual instantiations each to display a digit of the possible 65,536 values of the 16-bit counter.

# Appendix – A

## A-1 Pin Mapping Table

|  |  |  |  |
| --- | --- | --- | --- |
| Signal Name | FPGA Pin No. | Description | I/O Standard |
| SW[0] | PIN\_AB12 | Slide Switch [0] | 3.3V |
| SW[1] | PIN\_AC12 | Slide Switch [1] | 3.3V |
| HEX0[0] | PIN\_AE26 | Seven Segment Digit 0[0] | 3.3V |
| HEX0[1] | PIN\_AE27 | Seven Segment Digit 0[1] | 3.3V |
| HEX0[2] | PIN\_AE28 | Seven Segment Digit 0[2] | 3.3V |
| HEX0[3] | PIN\_AG27 | Seven Segment Digit 0[3] | 3.3V |
| HEX0[4] | PIN\_AF28 | Seven Segment Digit 0[4] | 3.3V |
| HEX0[5] | PIN\_AG28 | Seven Segment Digit 0[5] | 3.3V |
| HEX0[6] | PIN\_AH28 | Seven Segment Digit 0[6] | 3.3V |
| HEX1[0] | PIN\_AJ29 | Seven Segment Digit 1[0] | 3.3V |
| HEX1[1] | PIN\_AH29 | Seven Segment Digit 1[1] | 3.3V |
| HEX1[2] | PIN\_AH30 | Seven Segment Digit 1[2] | 3.3V |
| HEX1[3] | PIN\_AG30 | Seven Segment Digit 1[3] | 3.3V |
| HEX1[4] | PIN\_AF29 | Seven Segment Digit 1[4] | 3.3V |
| HEX1[5] | PIN\_AF30 | Seven Segment Digit 1[5] | 3.3V |
| HEX1[6] | PIN\_AD27 | Seven Segment Digit 1[6] | 3.3V |
| HEX2[0] | PIN\_AB23 | Seven Segment Digit 2[0] | 3.3V |
| HEX2[1] | PIN\_AE29 | Seven Segment Digit 2[1] | 3.3V |
| HEX2[2] | PIN\_AD29 | Seven Segment Digit 2[2] | 3.3V |
| HEX2[3] | PIN\_AC28 | Seven Segment Digit 2[3] | 3.3V |
| HEX2[4] | PIN\_AD30 | Seven Segment Digit 2[4] | 3.3V |
| HEX2[5] | PIN\_AC29 | Seven Segment Digit 2[5] | 3.3V |
| HEX2[6] | PIN\_AC30 | Seven Segment Digit 2[6] | 3.3V |
| HEX3[0] | PIN\_AD26 | Seven Segment Digit 3[0] | 3.3V |
| HEX3[1] | PIN\_AC27 | Seven Segment Digit 3[1] | 3.3V |
| HEX3[2] | PIN\_AD25 | Seven Segment Digit 3[2] | 3.3V |
| HEX3[3] | PIN\_AC25 | Seven Segment Digit 3[3] | 3.3V |
| HEX3[4] | PIN\_AB28 | Seven Segment Digit 3[4] | 3.3V |
| HEX3[5] | PIN\_AB25 | Seven Segment Digit 3[5] | 3.3V |
| HEX3[6] | PIN\_AB22 | Seven Segment Digit 3[6] | 3.3V |
| HEX4[0] | PIN\_AA24 | Seven Segment Digit 4[0] | 3.3V |
| HEX4[1] | PIN\_Y23 | Seven Segment Digit 4[1] | 3.3V |
| HEX4[2] | PIN\_Y24 | Seven Segment Digit 4[2] | 3.3V |
| HEX4[3] | PIN\_W22 | Seven Segment Digit 4[3] | 3.3V |
| HEX4[4] | PIN\_W24 | Seven Segment Digit 4[4] | 3.3V |
| HEX4[5] | PIN\_V23 | Seven Segment Digit 4[5] | 3.3V |
| HEX4[6] | PIN\_W25 | Seven Segment Digit 4[6] | 3.3V |

## A-2 RTL Diagrams

### A-2.a clk\_divider

Diagram, schematic

Description automatically generated

### A-2.b counter

Diagram, schematic

Description automatically generated

### A-2.c bcd

Diagram, schematic

Description automatically generated

### A-2.d bcd\_decimalDiagram, waterfall chart Description automatically generated

### A-2.e top module

Chart, waterfall chart

Description automatically generated

## A-3 Module Code

### A-3.a clk\_divider

module clk\_divider(

input clk,

//input reset,

output reg clk\_out

);

reg [24:0] count; // Make the counter big enough to hold our number

// Always block to handle the counter

always @(posedge clk)

begin

//if(reset)

//count <= 25'b0;

if(count < 250000 - 1) // 50 MHz / (2 x 250 kHz) = 100 Hz

count <= count + 1;

else

count <= 25'b0;

end

// Always block to handle the flip flop portion

always @(posedge clk)

begin

//if(reset)

//clk\_out <= 1'b0;

if(count == 250000 - 1)

clk\_out <= ~clk\_out;

else

clk\_out <= clk\_out;

end

endmodule

### A-3.b counter

module counter

#(parameter BIT\_COUNT = 16)(

input clk,

input reset\_n,

input enable,

output reg[BIT\_COUNT-1:0] count

);

always @(posedge clk)

begin

if(reset\_n == 0)

count <= 0;

else if(enable == 0)

count <= count + 1;

else

count <= count;

end

endmodule

### A-3.c counter

module bcd

#( parameter W = 16) // input width

( input [W-1 :0] bin , // binary

output reg [W+(W-4)/3:0] bcd ); // bcd {...,thousands,hundreds,tens,ones}

integer i,j;

always @(bin)

begin

for(i = 0; i <= W+(W-4)/3; i = i+1) bcd[i] = 0; // initialize with zeros

bcd[W-1:0] = bin; // initialize with input vector

for(i = 0; i <= W-4; i = i+1) // iterate on structure depth

for(j = 0; j <= i/3; j = j+1) // iterate on structure width

if (bcd[W-i+4\*j -: 4] > 4) // if > 4

bcd[W-i+4\*j -: 4] = bcd[W-i+4\*j -: 4] + 4'd3; // add 3

end

endmodule

### A-3.d bcd\_decimal

module bcd\_decimal(

input [3:0] hex,

output reg [6:0] display

);

always @(hex)

case(hex)

4'h0: display = 7'b1000000;

4'h1: display = 7'b1111001;

4'h2: display = 7'b0100100;

4'h3: display = 7'b0110000;

4'h4: display = 7'b0011001;

4'h5: display = 7'b0010010;

4'h6: display = 7'b0000010;

4'h7: display = 7'b1111000;

4'h8: display = 7'b0000000;

4'h9: display = 7'b0011000;

default : display = 7'b1111111;

endcase

endmodule

### A-3.e top module

module Lab02(

input CLOCK\_50,

input [1:0] SW,

output [6:0] HEX0,

output [6:0] HEX1,

output [6:0] HEX2,

output [6:0] HEX3,

output [6:0] HEX4

);

wire q;

wire [15:0] p;

wire [19:0] g;

clk\_divider divider(CLOCK\_50, q);

counter count(q, SW[1], SW[0], p);

bcd U1(p, g);

bcd\_decimal M1(g[3:0], HEX0);

bcd\_decimal M2(g[7:4], HEX1);

bcd\_decimal M3(g[11:8], HEX2);

bcd\_decimal M4(g[15:12], HEX3);

bcd\_decimal M5(g[19:16], HEX4);

endmodule

# Table of Figures

[Figure 1: Clock Divider Equation 4](#_Toc91856676)

# References

[*"Binary-to-BCD Converter: "Double-Dabble Binary-to-BCD Conversion Algorithm"*](https://web.archive.org/web/20120131075956/http:/edda.csie.dyu.edu.tw/course/fpga/Binary2BCD.pdf)

DE1-SoC User Manual