Version

.8

DIGITAL RTL DESIGN PROJECT

PLL FUNCTION

Senior Project Design Specification

Dylan Iverson
Jisu Park
Zachary Weiss
Ketan Ramani
Primary Owner: Zachary Weiss

05/22/2023

Revision History Block

Table 1: Revision History Block

Revision Number	Date	Revision Changes	Author(s)
0.1	10/22/2022	Document Created	Dylan Iverson
0.1	10/23/2022	Migrated to Word Template	Zachary Weiss
0.2	10/29/2022	Added "Changes made by" Section to Revision History, Moved Revision history above TOC	Dylan Iverson
0.2	10/30/2022	Migrated to Docs Template, Finished 0.2 Requirements: Introduction, References, and System Overview First Draft	Zachary Weiss
0.2	10/30/2022	Inserted and documented architecture 1 ver 0.1 under system architecture section	Jisu Park
0.3	11/11/2022	Began Glossary, Added Block Diagram and Description For Second Architecture	Zachary Weiss
0.3	11/17/2022	Began User Interface and Software	Jisu Park
0.3	11/24/2022	Replaced Architecture 1. Added Architecture 3	Jisu Park
0.4	11/27/2022	Added information on how the comparing frequency code / and odd divider code work. In the software design portion.	Dylan Iverson
0.4	11/27/2022	Added Ethical Considerations Section	Zachary Weiss
0.4	1/14	Updated the system architecture, and the system design (modules for the architecture)	Jisu Park
0.5	01/21/2023	Updated Project Requirements, and Divider explanation.	Dylan Iverson
0.5	1/22/2023	Incorporated feedback from Spec 0.4 comments including a rewrite of *Project Requirements,* switching *Reference Section* to the IEEE standard, adding Figure labels and	Zachary Weiss

		correcting formatting issues throughout. Updated Glossary	
0.5	1/22/2023	Proofread and edited minor details.	Ketan Ramani
0.6	2/15/2023	Updated Figure 1. Architecture diagram on 3. System architecture	Jisu Park
0.6	2/16/2023	Changed "If time permits there will be a phase locking feedback loop" to "We have implemented a one shot into our Ring Oscillator to match our output phase to the input." in the system overview.	Dylan Iverson
0.6	2/19/2023	Renamed "Frequency Difference" to "Frequency Ratio", and edited the description to be easier to understand Deleted the comparing Frequencies module description in the software description. This is from an old architecture and is no longer used in the current iteration.	Dylan Iverson
0.6	2/19/2023	Added to *System Architecture.* Added to *Ring Oscillator* under *System Design - a. software.*	Zachary Weiss
0.6	2/19/2023	Added the architecture's logic elements number and the choice of protocol on 4.b Hardware. Added how ADPLL takes data of input from the user on 4.c Human Interface.	Jisu Park
0.7	4/9/2023	Added *Hardware* section added/edited *System Overview, Project Requirements, System Architecture and Design.*	Zachary Weiss
0.7	4/9/2023	Updated System Design C. user interface Rewrote the first sentence of Section 3 to be coherent.	Dylan Iverson
0.8	5/18/2023	Made changes according to other's peer review	Ketan Ramani
0.8	5/20/2023	Added references for I2C code, and long division codes.	Dylan

0.8	5/20/2023	Edited & Formatted *Revision Block* & *Table of Contents.* Added to & edited *Introduction*, *System Architecture*, *Project Requirements, System Design, Bill of Materials* & *Ethical Considerations.* Created *Reference* subcategories & added references. Added RTL to *Appendix.*	Zachary Weiss

Table of Contents

1. INTRODUCTION

This document was created in collaboration to report on a year-long senior capstone project into the research, development, design, and testing of an all digital phase locking loop (ADPLL). Phase locked loops are a control system that generates an output signal whose phase is related to the input signal. Historically PLL's were analog components. The topology used a VCO, phase detector and feedback loop. By increasing and decreasing the voltage the system could modulate the clocks to obtain the desired output.

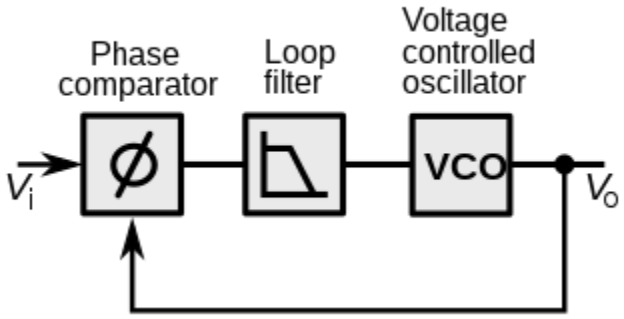


Figure 1. Simple Analog PLL

Although this topology works well the VCO, LPF, and Phase detector/comparator are manufactured using RC or LC pairs. This makes analog PLLs physically large and expensive. VCO's also have to be manufactured to narrow specifications. These properties result in a product

that has costly validation processes, and requires complete hardware updates.

A digital alternating solves these problems by removing the large capacitors and Inductors and having a much broader range of operation. Furthermore, ADPLLs can be integrated on a single integrated circuit making them more compatible with modern electronic devices.

Texas Instruments has tasked us with designing, simulating and emulating a digital PLL. This specification document is designed for those of a technical background with an emphasis on electrical and computer engineering. It describes the project requirements, system architecture, system design, project bill of materials, ethical considerations and references. The document's primary goal is to facilitate others to understand and implement the process we took in designing and testing an ADPLL.

2. SYSTEM OVERVIEW

PLLs are implemented in computers, radios and telecommunication systems to synthesize, demodulate, and synchronize clocks. Although ADPLLs of this complexity exist, they are beyond the scope of this project. This project's focus is on clock synthesize alone, specifically frequency multiplication. Our architecture focuses on generating a high internal frequency, and then dividing that frequency to the desired output. The system can output two clocks simultaneously and has two modes of operation. In mode one the user can multiply their input clock by a number they specify, in mode two the user can target a specific frequency they would like the system to output.

Our design uses a feed forward topology with no feedback loop. The phase is locked to the input using a one shot circuit that continually syncs a ring oscillator to the input signal. The ADPLL has three subsystems. The one shot enabled ring oscillator for signal generation, a controller for selecting output modes, and a system for dividing and targeting signals.

The code is HDL, simulated in software and emulated on an FPGA. Accuracy of the system was measured by outputting the signals through the GPIO interface to a mixed signal oscilloscope.

3. PROJECT REQUIREMENTS

1. System should contain no (or very little) analog components. The goal is to create a purely digital system.
2. Output clocks should be capable of speeds 10x greater than the input clocks and exceed any system's 50 MHz crystal oscillator.
3. Emulated results should be within a 2 percent error of theoretical calculations.
4. The system must auto-calibrate & continue to output consistent signals in the event the input is disrupted or terminated.

4. SYSTEM ARCHITECTURE

a. ADPLL

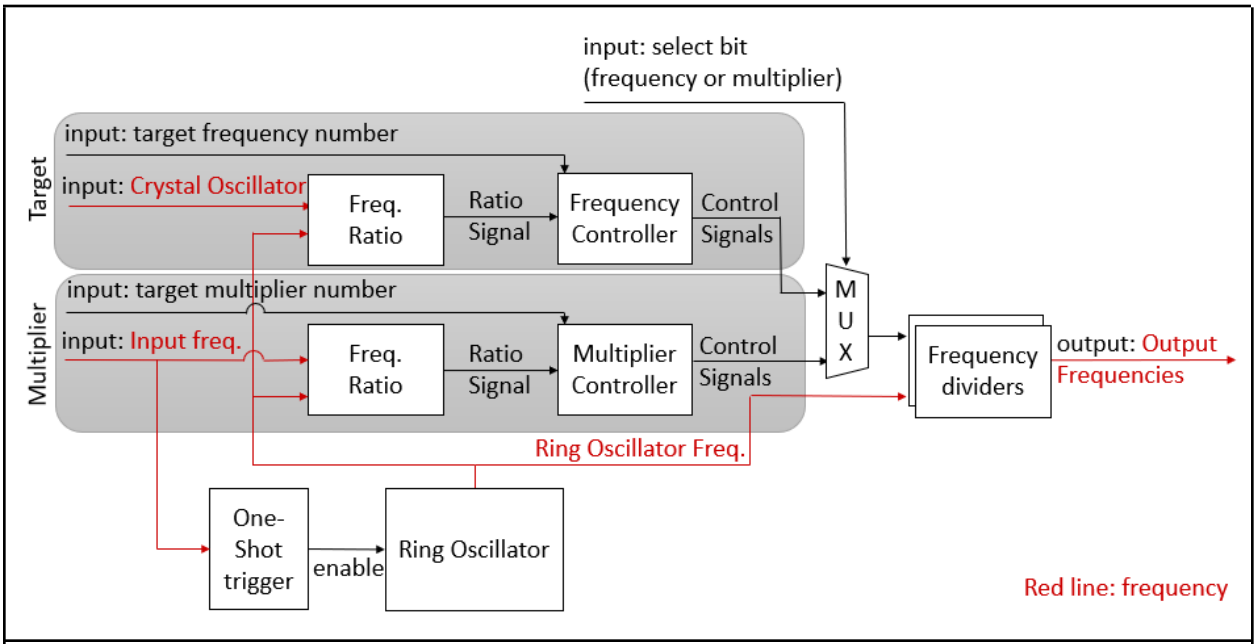


Figure 1. Architecture

This architecture is able to generate frequencies higher than the input frequency by first using the ring oscillator to generate a very high frequency, and then using dividers to divide that down. The frequency difference module compares the speed difference between input frequency and the ring oscillator frequency. The controller module divides the frequency difference by the multiplier, and sends the result to the frequency divider. The frequency divider divides the ring oscillator frequency with the results from the controller.

For example, if a user wants to input 1MHz, and he wants to multiply input by 3.25 times, he gets output of 3.25 MHz.

Input = 1MHz	Multiplier = 3.25	Output = 1MHz x 3.25 = 3.25 MHz
Input = 3MHz	Multiplier = 5.00	Output = 3MHz x 5.00 = 15.00 MHz

b. Frequency Ratio

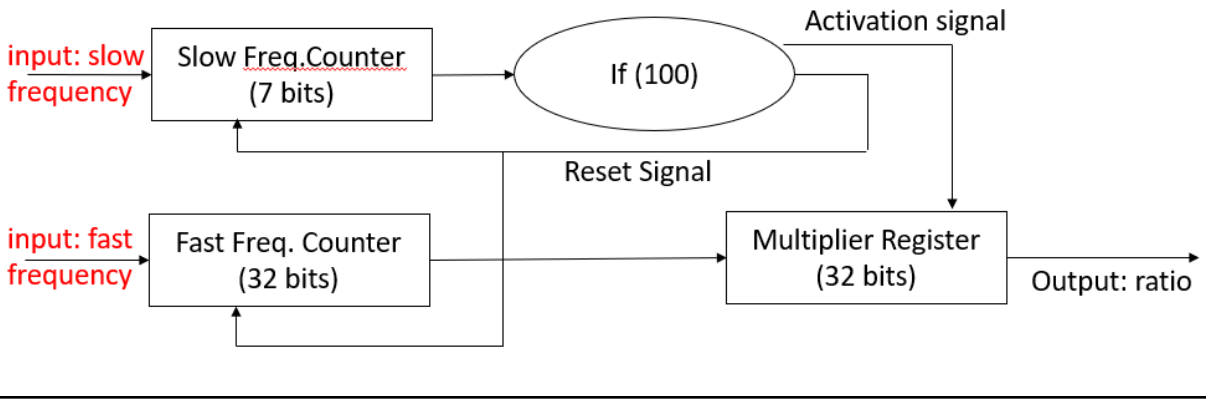


Figure 4. The diagram of the module for two different frequencies (see Appendix a, and Appendix b.Calculators: Ratio).

The Frequency Ratio module is for measuring the ratio of a faster frequency over a slower frequency. To do that, it has 2 counters for each frequency. It takes 100 cycles to have the average frequency. It needs a longer time to measure the frequency, but it can assure more steady input frequency reading.

The slower clock has 7 bits, so it can count up to 100. When it hits 100, send a signal to capture the faster counter number and reset both of the counters. Even if one of the frequencies is disconnected, the module still sends out the previous output signal.

c. Frequency Divider

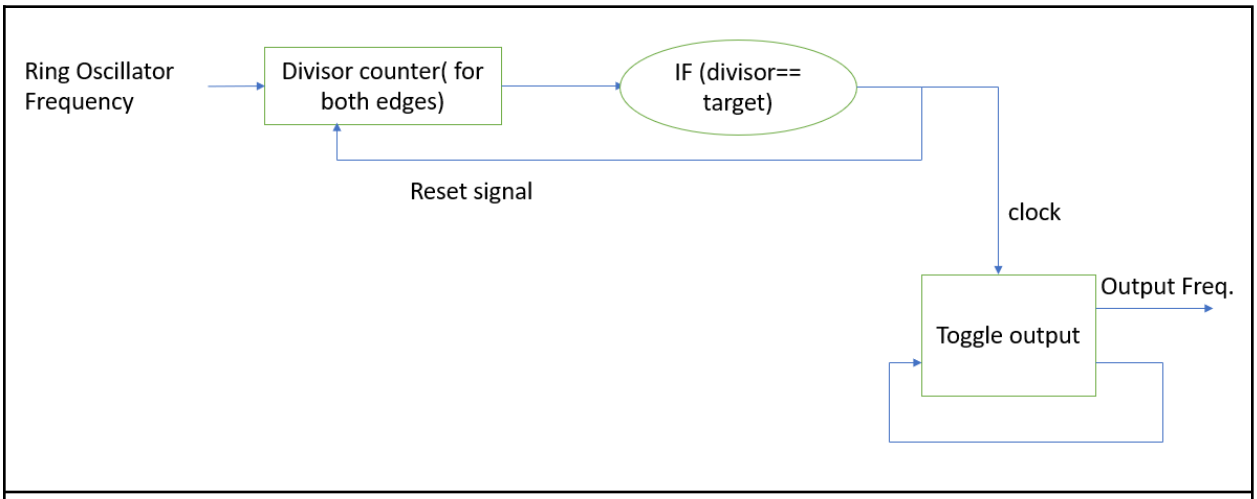


Figure. Frequency divider(See appendix to see the circuit design).

We had run into a problem where we could only divide frequencies in half, and thus could not divide by odd integers. We have since realized that getting a perfect division for odd integers is not strictly necessary. Most system's only rely on the rising edge of the clock with little regard for duty cycle. This means it is acceptable that our output doesn't have a 50/50 duty cycle, as long as its period and phase are correct.

Our new divider works by calculating half the multiplier and assigning it to a value Half_way. We then set a counter iterating on each positive edge of the ring oscillator. If the counter is less than Half_way we set our output clock to 1. If the counter is equal to or higher than the counter, then we set it to zero. Finally when our counter reaches the target multiplier, we know we have reached the end of our output period, so we reset the counter and repeat.

This current iteration of the divider is the most simple and intuitive version we have made. It trades duty cycle accuracy for the benefit of being able to only change our output off of one edge.

d. Ring Oscillator

The ring oscillator exploits the natural propagation delay of an electrical signal. The small amount of time the inverter takes to flip is multiplied by the number of inverters in the chain. In our design the first inverter input is fed from the output of a three input NAND. The other inputs to the NAND are the final output of the inverter chain, and a separate enable toggling on when the system voltage supply reaches steady state.

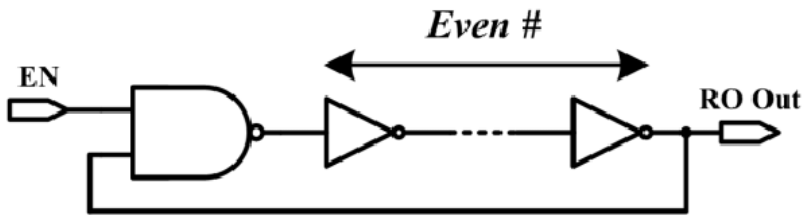


Figure. 3: Ring Oscillator with NAND gate

The one shot or Multstable Monovibrator, sends out a short pulse to reset the ring oscillator and prevent phase drift. The architecture is a DFF with Q being buffered and then fed into the reset. Q or Q_bar is then fed into the NAND.

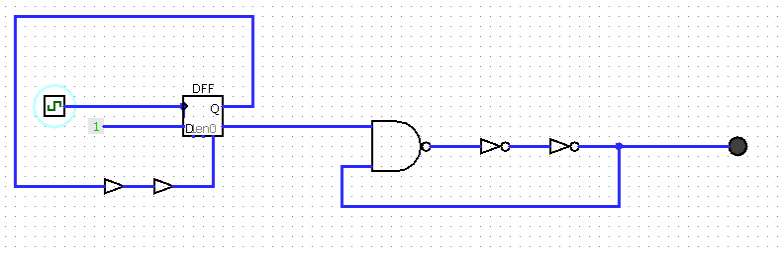


Figure. 3: Ring Oscillator with one-shot enable

The completed system works by sending the system clock into the one shot. Only when the voltage supply reaches full operating power does the enable on the ring oscillator get toggled. At that point the ring oscillator begins outputting a frequency at a rate of 5.56 Ghz. Periodically every 60 ps the DFF in the one shot is triggered and the ring oscillator holds its value for a system clock cycle before resuming its output.

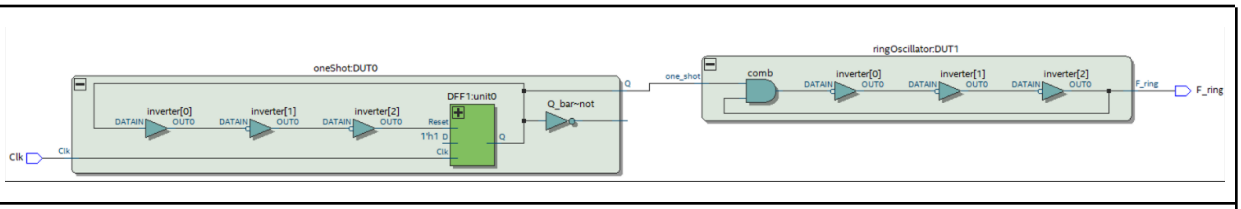


Figure. 4: RTL-view of one shot feeding into ring oscillator

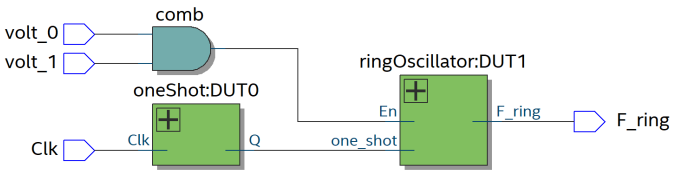


Figure. 5: Voltage supply controlled enable.

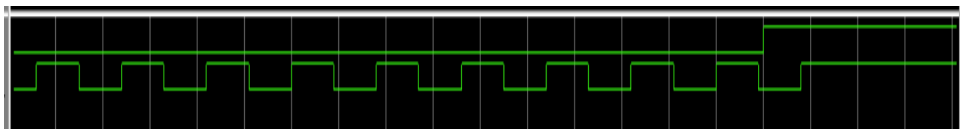


Figure. 5: Timing Diagram of input ‘Clk’ (top) and output: ‘F_ring’ (bottom).

5. SYSTEM DESIGN

a. HARDWARE DESCRIPTION LANGUAGE

This project is written Verilog and SystemVerilog in accordance with the IEEE 1364-2005 and IEEE 1800-2017 standards. It was coded to be synthesizable and physically realized by synthesis software available to specific FPGA technology.

b. SOFTWARE

This project was simulated in ModelSim - Intel FPGA Starter Edition Model Technology FPGA Edition version 2020.1 (Quartus Prime 20.1).

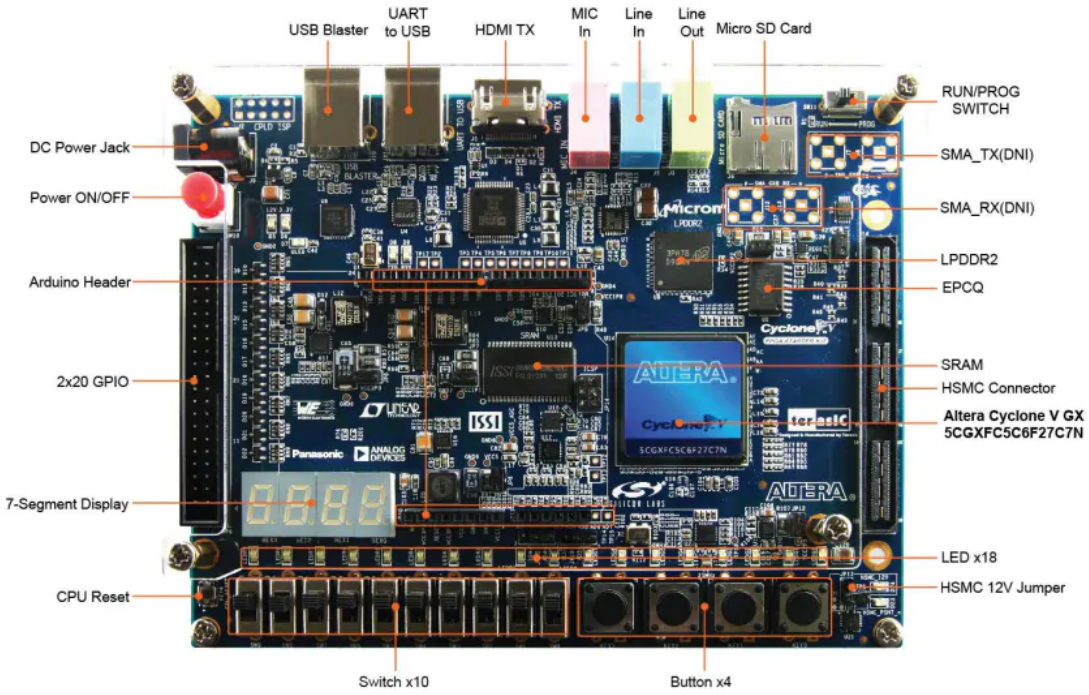
Timing analysis and RTL generation was implemented in Intel® Quartus® Prime Lite Edition Design Software Version 22.1 for Windows.

Terasic Cyclone V GX Starter Kit V1.01 System Builder was used to generate a top level design file, along with pin assignments, and I/O standard settings.

c. HARDWARE

FPGA Device

- Cyclone V GX 5CGXFC5C6F27C7N Device
- 77K Programmable Logic Elements
- 4884 Kbits embedded memory
- Six Fractional PLLs
- Two Hard Memory Controllers
- Six 3.125G Transceivers



Configuration and Debug

- Quad Serial Configuration device – EPCQ256 on FPGA
- On-Board USB Blaster (Normal Type-B USB connector)
- JTAG and AS mode configuration supported

Memory Device

- 4Gb LPDDR2 x32 bits data bus
- 4Mb SRAM x16 bits data bus

Communication

- UART to USB

Expansion I/O

- HSMC x 1(voltage levels: 2.5V), including 4-lanes 3.125G transceiver
- 2x20 GPIO Header (voltage levels: 3.3V)
- Arduino header, including analog pins
- SMA x 4 (DNI), one-lane 3.125G transceiver

Display

- HDMI TX, compatible with DVI v1.0 and HDCP v1.4

Audio

- 24-bit CODEC, Line-in, line-out, and microphone-in jacks

Switches, Buttons, LED, and 7-Segments

- 18 LEDs
- 10 Slide Switches
- 4 Debounced Push Buttons
- 1 CPU reset Push Buttons
- Four 7-Segments

Micro SD Card Socket

- Provides SPI and 4-bit SD mode for SD Card access

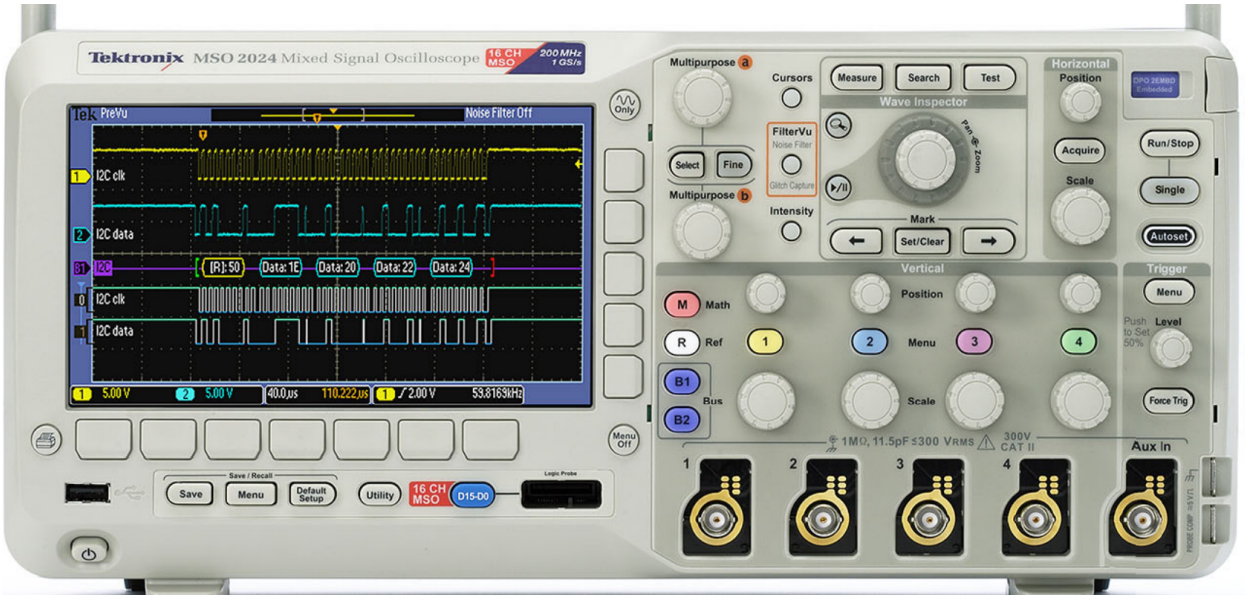
ADC

- 12-Bit Resolution, 500Ksps Sampling Rate. SPI Interface.
- 8-Channel Analog Input. Input Range : 0V ~ 4.096V.

Power

- 12V DC input

Tektronix MSO 2024 Mixed Signal Oscilloscope



Key performance specifications

- 200, 100, 70 MHz bandwidth models
- 2 and 4 analog channel models
- 16 digital channels (MSO series)
- 1 GS/s sample rate on all channels
- 1 megapoint record length on all channels
- 5,000 wfm/s maximum waveform capture rate
- Suite of advanced triggers

Key features

- Wave Inspector® controls provide easy navigation and automated search of waveform data
- FilterVu™ variable low-pass filter allows for removal of unwanted signal noise while still capturing high-frequency events
- 29 automated measurements, and FFT analysis for simplified waveform analysis
- TekVPI® probe interface supports active, differential, and current
- probes for automatic scaling and units
- p7 in. (180 mm) wide-screen TFT-LCD color display

Connectivity

- USB 2.0 host port on the front panel for quick and easy data storage

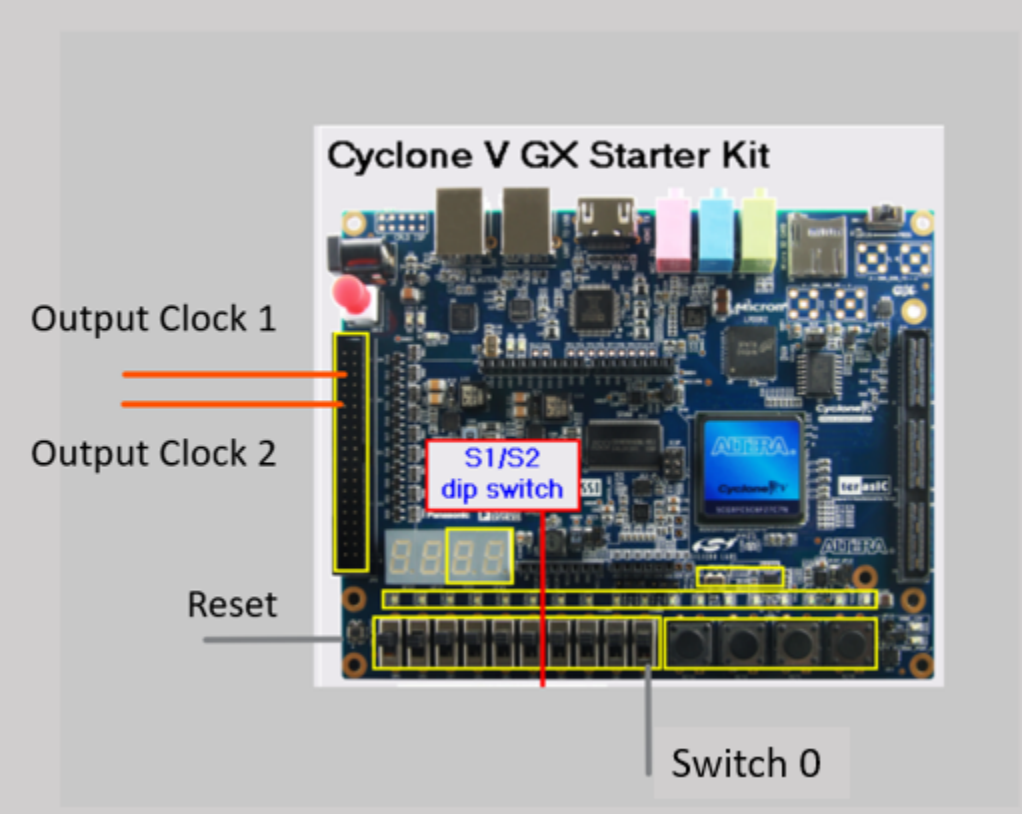
- USB 2.0 device port on rear panel for easy connection to a PC or direct printing to a PictBridge®-compatible printer
- Optional 10/100 Ethernet port for network connection and video-out port to export the oscilloscope display to a monitor or projector

Optional serial triggering and analysis

- Automated serial triggering, decode, and search options for I2C, SPI, CAN, LIN, and RS-232/422/485/UART

d. HUMAN INTERFACE

The system is emulated on the FPGA and oscilloscope using the built in buttons, switches and 7-segment display. The clock signals are output through the onboard GPIO to the oscilloscope.



The ADPLL has two functions, the multiply mode and the target mode. In the multiplier mode the buttons (keys) control incrementing and decrementing. The far right button increases the multiplier, while the preceding button decrements. Using switch 9 and 8 the user selects either the whole number portion of the multiplier or the decimal portion of the multiplier. The pattern is provided below.

Switch 9-8	Mode
00	Clock 1 whole number
01	Clock 1 decimal number
10	Clock 2 whole number
11	Clock 2 decimal number

This gives the system a theoretical multiplication range of 0 to 99.99. It is of note that the FPGA is limited to using only two of the 7-segment displays while using the GPIO. This constraint means we can only display either the whole number portion or the decimal portion of our multiplier at any given time. The current implementation displays whatever value the switches have selected.

Switches 0-2 are used as enables for the system. Switch 0 is a top level enable and switch 1 and 2 simulate a voltage protection scheme. If manufactured these switches would represent a minimum and fully operational voltage level for the PLL.

Lastly switch 4 is used to toggle the mode. 0 corresponds to multiplication mode while 1 corresponds to target frequency mode. While in the target frequency mode you can select a target frequency in the MHz scale from 0.01 to 511.99.

6. BILL OF MATERIALS

a. Software

Quartus & Modelsim are Intel products with free student licenses.

b. Hardware

Two Terasic Cyclone V GX 5CGXFC5C6F27C7N Devices were purchased courtesy of Texas Instruments. Retail price for each board prior to shipping, handling and taxes is ~$270 USD.

Tektronix MSO 2024 Mixed Signal Oscilloscope and Jumper wires to connect the GPIO to oscilloscope were provided by the University of Washington Tacoma electrical engineering laboratory at no additional cost to us. Retail price of equivalent oscilloscopes ~$2100 USD, jumper wires ~$8.00 USD.

7. ETHICAL CONSIDERATIONS

This project was implemented using software and hardware that has been manufactured in accordance with standards laid out by the FCC. This project was a student exercise, and will not be manufactured, bought, sold or implemented in other schemes. Under these circumstances there are no safety, risk or ethical factors to consider.

To examine the ethical issues related to this project we have reviewed the NSPE code of ethics. Within the *Fundamental Canons,* section 6 it states engineers must, “Conduct themselves honorably, responsibly, ethically, and lawfully so as to enhance the honor, reputation and usefulness of the profession.” For students such as ourselves this means to seek and offer honest criticism for technical work, to acknowledge mistakes and credit each other fairly.

PLL’s are used across industry for computer systems, radio and telecommunications. Each of these industries comes with their own ethical, and safety risks and must be considered individually. A faulty or malfunctioning PLL can cause total SoC performance failure, which in turn can cause catastrophic failure to computer systems and may lead to injury or death.

PLL’s are sensitive to parametric deviations or process defects and must be tested before being implemented into production. Those that plan to implement a PLL should verify: output duty cycle, output frequency, output power, current consumption, phase lock and capture time, jitter for purity and phase noise, and bandwidth/settling time.

8. GLOSSARY

ADPLL	All Digital Phase Locking Loop
DCO	Digitally Controlled Oscillator
DFF	D Flip-Flop
PLL	Phase Locking Loop
VCO	Voltage Controlled Oscillator
NAND	Equivalent to NOT AND

9. REFERENCES

a. BOOKS & JOURNALS

- D. Banerjee, *PLL performance, simulation and Design*. Indianapolis, IN: Dog Ear Publishing, 2006.
- D. Johns and K. W. Martin, “Phase Locked Loops,” in *Analog Integrated Circuit Design*, Chichester: Wiley, 2010, pp. 648–689.
- J. M. Rabaey, “Timing Issues in Digital Circuits,” in *Digital Integrated Circuits: A design perspective*, Upper Saddle River: Prentice-Hall, 2003, pp. 42–98.
- A. Asquini. Technique de BIST pour synthétiseurs de fréquence RF. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2010. Français. ffNNT : ff. fftel-00482401

b. WEBSITES

- A. Isagholian, "Tiny_But_Mighty_I2C_Master_Verilog" [Online]. Available: https://github.com/0xArt/Tiny_But_Mighty_I2C_Master_Verilog. Accessed on: Feb. 28, 2023.
- Vipin, "Synthesisable Verilog Code for Division of Two Binary Numbers." [Online]. Available: https://verilogcodes.blogspot.com/2015/11/synthesisable-verilog-code-for-division.html. Accessed on: Feb 22, 2023.

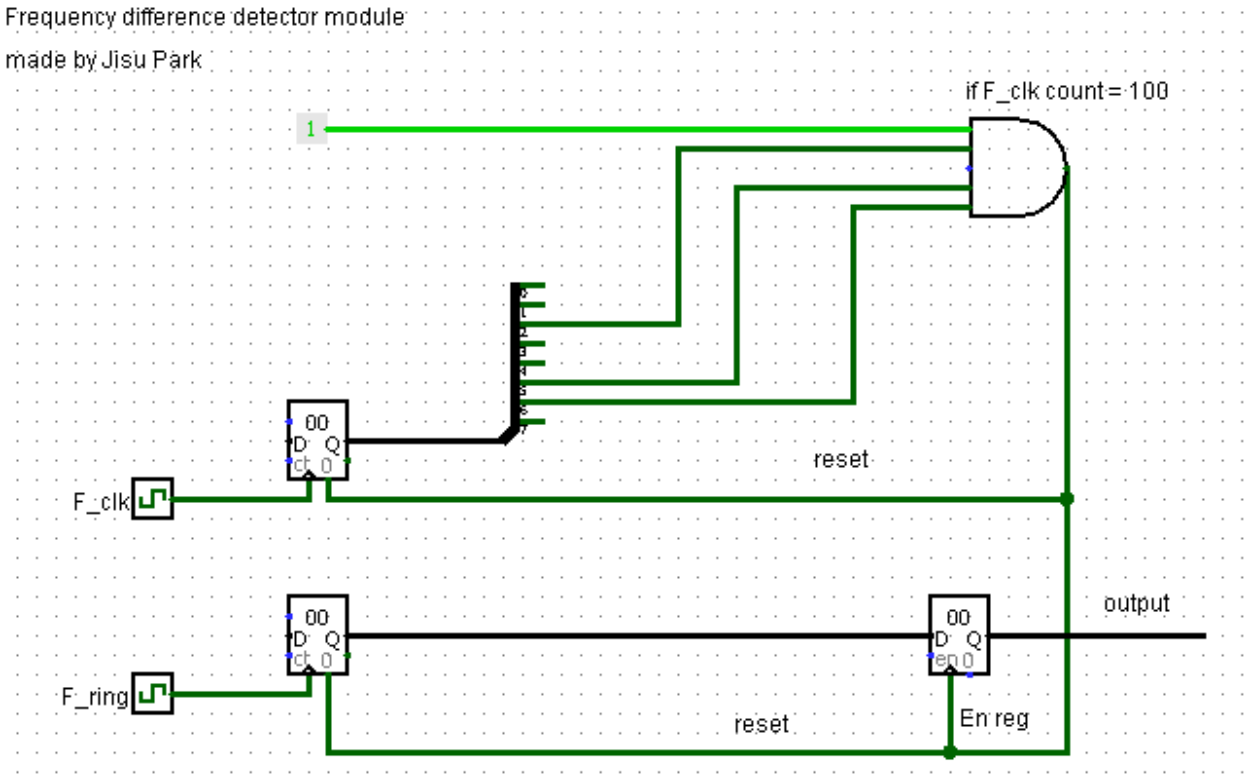
c. SOFTWARE & HARDWARE

- ModelSim-Intel® FPGAs Standard Edition Software Version 20.1.1
- Intel® Quartus® Prime Lite Edition Design Software Version 22.1 for Windows
- Terasic - All FPGA Boards - Cyclone V - Cyclone V GX Starter Kit
- MSO2024B Tektronix | Mouser

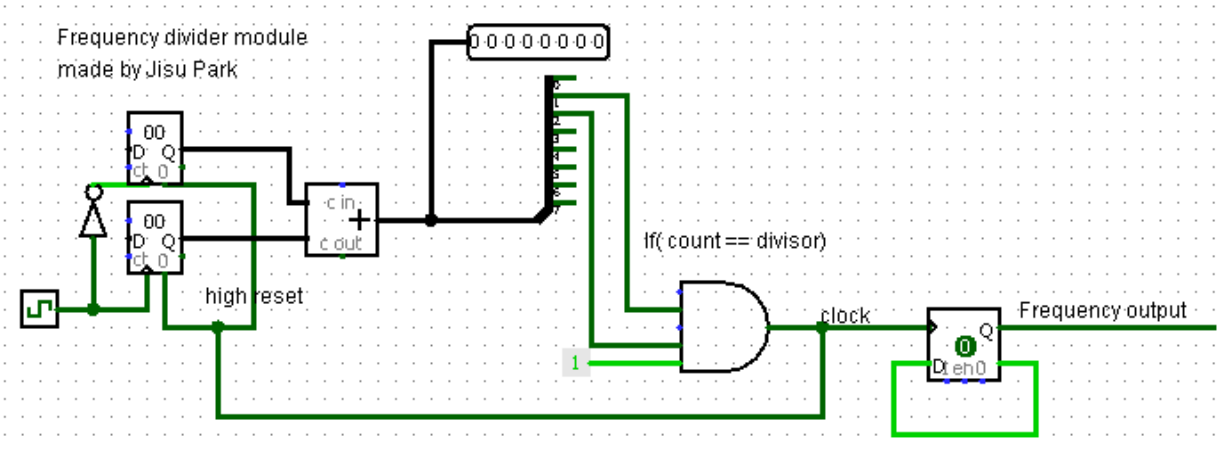
10. APPENDIX

a. Visualizations

A. Frequency Ratio Visualization in Logisim



B. Frequency divider Visualization in Logisim



b. RTL

Top Level ADPLL

Date: May 21, 2023 APDLL.sv Project: ADPLL_5_3

```
1   //Author: Jisu
2
3
4   //C stands for control signals
5   //F stands for frequencies
6
7   //todo: implement a register that convert 1 channal I2C bursts to 32bits
8
9   `timescale 1ps/1ps
10  module ADPLL(En, volt_0, volt_1, Reset,F_input,C_info,C_target, F_xtal,F_output1,F_output2);
11     input En,Reset, volt_0, volt_1;     // enable, Reset
12     input C_info;         //information whether the target is frequency(1) or multipler(0)
13     input F_input;        //input frequency
14     input [31:0]C_target;      //target frequency or multiplier which has one channel input
    with bursts of data (I2C)
15     input F_xtal;         //Crystal oscillator : 50mhz
16     output reg F_output1,F_output2;
17
18     wire [31:0] C_F2M1, C_F2M2;
19     wire [31:0] C_N,C_1,C_2;
20     wire [31:0] C_N1,C_N2;//controller output
21
22     wire [31:0]C_XR,C_RI;
23     wire F_ring;          //frequency generated by the ring oscillator
24     //module Ring (Clk, F_ring);
25     Ring RingGenerate(F_input, volt_0, volt_1, F_ring);
26     //module Freq_ratio(En,Reset,F_clk,F_ring,C_freq);
27     Freq_ratioA Crystal_Input(En,Reset,F_xtal,F_ring,C_XR);
28
29     Freq_ratioA Ring_Input(En,Reset,F_input,F_ring,C_RI);
30     //assign C_RI = 32'd500000;
31
32     //module Multiple_Calculator (En, Reset,Clk, Ratio_Crystal, Target_Freq, Multiplier);
33
34     //Multiple_Calculator FreqToMulti(En, Reset,F_xtal, C_XI, C_target, C_F2M);
35
36     //module Controller (En, Reset, C_N, C_Freq, C_N1, C_N2);
37     Controller Process(En, Reset, C_target, C_RI, C_N1, C_N2);
38     //module TargetController (En, Reset, C_RX, C_N, C_N1,C_N2);
39     TargetController Process2 (En, Reset, C_XR, C_target, C_F2M1, C_F2M2);
40
41     //assign C_1 = (C_info)? C_N1:C_F2M1;
42     //assign C_2 = (C_info)? C_N2:C_F2M2;
43
44     //This below does not work as a mux, but we have identified that this is
45     //Where the problem is. put Control signals directly into Divider to test
46     //TargetController needs tuned.
47     genvar i;
48     generate
49     for (i = 0; i < 32; i++)
50     begin : assign_C
51           assign C_1[i] = (!C_info & C_N1[i]) | (C_info & C_F2M1[i]);
52           assign C_2[i] = (!C_info & C_N2[i]) | (C_info & C_F2M2[i]);
53
54     end
55     endgenerate
56
57     //module Divider(En,Reset,C_N,F_input,F_output);
58     Divider first (En,Reset,C_N1,F_ring,F_output1);
59
60     Divider second(En,Reset,C_N2,F_ring,F_output2);
61
62
63     endmodule
64
```

User Interface


```
1
2   //=======================================================
3   //  This code is generated by Terasic System Builder
4   //=======================================================
5
6   module User_Input(
7
8      ////////// CLOCK //////////
9      input                       CLOCK_50_B5B,
10
11     ////////// KEY //////////
12     input                       CPU_RESET_n,
13     input            [3:0]      KEY,
14
15     ////////// SW //////////
16     input            [9:0]      SW,
17
18     ////////// SEG7 //////////
19     output           [6:0]      HEX0,
20     output           [6:0]      HEX1,
21     ////////// GPIO GPIO connect to GPIO Default //////////
22     inout            [35:0]     GPIO,
23     output           [31:0]     Control_Signal
24  );
25
26
27  //=======================================================
28  //  REG/WIRE declarations
29  //=======================================================
30     reg [31:0] Control_SignalT;
31     reg [6:0] Dec_1;
32     reg [6:0] Dec_2;
33     reg [8:0] Whole_1;
34     reg [8:0] Whole_2;
35     reg [7:0] Display;
36
37  //=======================================================
38  //  Structural coding
39  //=======================================================
40
41  // assign 7-segment display outputs based on the current value
42  Decoder Unit1 (Display % 10,HEX0);
43  Decoder Unit2 ((Display/10)%10,HEX1);
44  // assign 7-segment display outputs based on the current value
45  assign Control_Signal = Control_SignalT;
46  //Make slow clock
47  reg [31:0] counter = 0;
48  reg slow_clock;
49  localparam CLOCK_DIVIDER = 5000000; // 5 million
50  always @(posedge CLOCK_50_B5B) begin
51      if (counter == CLOCK_DIVIDER) begin
52          slow_clock <= ~slow_clock; // invert the slow clock signal every 50 cycles
53          counter <= 0;
54      end else begin
55          counter <= counter + 1;
56      end
57  end
58
59  assign CLOCK_1MHz = slow_clock;
60    // clocked process to update value based on KEY inputs
61    always @(posedge slow_clock) begin
62     case (SW[9:8])
63        2'b00:begin
64           Display = Whole_1;
65           if (!CPU_RESET_n) begin
66                 // reset value to 0 on CPU reset
67                    Whole_1 <= 8'd0;
68              end else begin
69                  // increment value on KEY[0] press
```

```
70                      if (!KEY[0]) begin
71                      Whole_1 <= (Whole_1 == 99) ? 8'd0 : Whole_1 + 1;
72                      end
73                  // decrement value on KEY[1] press
74                      else if (!KEY[1]) begin
75                         Whole_1 <= (Whole_1 == 0) ? 8'd99 : Whole_1 - 1;
76                      end
77              end
78          end
79          2'b01:begin
80             Display = Dec_1;
81             if (!CPU_RESET_n) begin
82                   // reset value to 0 on CPU reset
83                      Dec_1 <= 7'd0;
84                end else begin
85                    // increment value on KEY[0] press
86                      if (!KEY[0]) begin
87                         Dec_1 <= (Dec_1 == 99) ? 7'd0 : Dec_1 + 1;
88                      end
89                  // decrement value on KEY[1] press
90                      else if (!KEY[1]) begin
91                         Dec_1 <= (Dec_1 == 0) ? 8'd99 : Dec_1 - 1;
92                      end
93              end
94          end
95          2'b10:begin
96              Display = Whole_2;
97             if (!CPU_RESET_n) begin
98                   // reset value to 0 on CPU reset
99                      Whole_2 <= 8'd0;
100               end else begin
101                   // increment value on KEY[0] press
102                     if (!KEY[0]) begin
103                        Whole_2 <= (Whole_2 == 99) ? 8'd0 : Whole_2 + 1;
104                     end
105                 // decrement value on KEY[1] press
106                     else if (!KEY[1]) begin
107                        Whole_2 <= (Whole_2 == 0) ? 8'd99 : Whole_2 - 1;
108                     end
109             end
110
111         end
112         2'b11:begin
113            Display = Dec_2;
114            if (!CPU_RESET_n) begin
115                  // reset value to 0 on CPU reset
116                     Dec_2 <= 7'd0;
117               end else begin
118                   // increment value on KEY[0] press
119                     if (!KEY[0]) begin
120                        Dec_2 <= (Dec_2 == 99) ? 7'd0 : Dec_2 + 1;
121                     end
122                 // decrement value on KEY[1] press
123                     else if (!KEY[1]) begin
124                        Dec_2 <= (Dec_2 == 0) ? 8'd99 : Dec_2 - 1;
125                     end
126             end
127         end
128      endcase
129  //assign output control signal
130  Control_SignalT[6:0] = Dec_1;
131  Control_SignalT[15:7] = Whole_1;
132  Control_SignalT[22:16] = Dec_2;
133  Control_SignalT[31:23] = Whole_2;
134    end
135
136  endmodule
```

7-Segment Decoder

Date: May 21, 2023 Decoder.sv Project: ADPLL_5_3

```
1   /* Zachary Weiss 4/22/2022
2   TCES 330
3   This is the Q1 portion of HW 3
4   Here we describe a 7 seg display for hexidecimal
5   the digits are on the left 0-15
6   corresponding 7 seg assignment is on the right
7   for the 7 seg : 0 is on, 1 is off */
8
9   module Decoder(SW, HEX);
10
11     input [3:0] SW; //4 switchs
12     output logic [0:6] HEX; // 7 seg outputs
13
14        always @(SW) begin
15           case (SW)
16  //input switch is left output is on right
17            4'b0000: HEX = 7'b1000000; //0
18            4'b0001: HEX = 7'b1111001; //1
19            4'b0010: HEX = 7'b0100100; //2
20            4'b0011: HEX = 7'b0110000; //3
21            4'b0100: HEX = 7'b0011001; //4
22            4'b0101: HEX = 7'b0010010; //5
23            4'b0110: HEX = 7'b0000010; //6
24            4'b0111: HEX = 7'b1111000; //7
25            4'b1000: HEX = 7'b0000000; //8
26            4'b1001: HEX = 7'b0010000; //9
27            default: HEX = 7'b1111111; //off
28
29           endcase
30        end
31
32  endmodule
33
```

Controller: ADPLL

Date: May 21, 2023 Controller.sv Project: ADPLL_5_3

```
1   // This module takes a control signal that represents two multipliers.
2   // These Multipliers can range from 00.00 to 511.99.
3   // This module then takes those multipliers and calculates what number we need to divide
    our Ring oscillator by to achieve the expected output in later modules.
4   `timescale 1ps/1ps
5   module Controller (En, Reset, C_N, C_Freq, C_N1, C_N2);
6
7   input En, Reset;
8   input reg[31:0] C_N, C_Freq; //C_N: target multiplier, C_Freq: control signal of frequency
    data.
9   output reg[31:0] C_N2, C_N1;
10  reg [25:0] Whole_1, Whole_2;
11  reg [6:0] Dec_1, Dec_2;
12  reg [31:0] TC1, TC2, T1,T2,T3,Tw1,Tw2, Tw1x,Tw2x; //Temporary intermediate values
13
14  assign Dec_1 = C_N[6:0];
15  assign Whole_1 = C_N[15:7];
16  assign Dec_2 = C_N[22:16];
17  assign Whole_2 = C_N[31:23];
18  assign T3 = C_Freq*100;
19
20  assign TC1 = ((8'd100*Whole_1 + Dec_1));
21  assign TC2 = ((8'd100*Whole_2 + Dec_2));
22
23  Number_Division DUT1(T3,TC1,T1);
24  Number_Division DUT2(T3,TC2,T2);
25
26  Number_Division DUT3(T1,32'd100,Tw1);
27  Number_Division DUT4(T2,32'd100,Tw2);
28
29  assign C_N1[31:7] = Tw1;
30  assign C_N2[31:7] = Tw2;
31
32  assign Tw1x = Tw1*7'd100;
33  assign Tw2x = Tw2*7'd100;
34
35  assign C_N1[6:0] = T1-Tw1x;
36  assign C_N2[6:0] = T2-Tw2x;
37
38     //T2 = C_Freq*100/(100*Whole_2 + Dec_2);
39     //Whole_2 = (T2/100);
40     //Dec_2 = T2-Whole_2*100;
41
42  endmodule
```

Controller: Frequency Targeting

Date: May 21, 2023 TargetController.sv Project: ADPLL_5_3

```
1   //Author: Dylan
2
3
4   //Takes a control signal Target_Freq. This is a 32 bit number representing two 16 bit
    numbers with 7 of those bits representing the decimal places.
5   //Target Freq should be in Mhz so you can currently target anywhere from 511.99 Mhz to 0.01
    Mhz (or 10 kilohertz)
6   //Requires Ratio of how many times faster that the Crystal oscillator is from the input. If
    crystal is 2.5 times faster Ratio_Crystal should equal 250.
7   //Crystal frequecy has been hardcoded in as 50 Mhz on line 18, if needed this value can be
    changed.
8   //Will calculate a multiplier to be sent to Divider module this number again can range from
    511.99 to 0.01
9
10  //modified: Jisu
11  //similar thing to Controller.
12
13  `timescale 1ps/1ps
14  module TargetController (En, Reset, C_RX, C_N, C_N1,C_N2);
15
16     input En, Reset;
17     input reg [31:0] C_RX;//Ratio_Crystal
18     input reg [31:0] C_N;//target frequency
19     output reg [31:0] C_N1,C_N2;
20     reg [31:0] fltDvs1,fltF1;  //floating number divisor, floating number final
21     reg [31:0] fltDvs2,fltF2;  //floating number divisor, floating number final
22     reg [31:0] intDvs1,intF1;  //integer number divisor,integer number final
23     reg [31:0] intDvs2,intF2;  //integer number divisor,integer number final
24
25     reg [31:0] dvs1,dvs2;   //final divisor=(intDvs*100+fltDvs*100)
26     reg [31:0] rmn1,rmn2;// reminder = C_ratio-(intF*Dvs)
27
28     reg[31:0] ringSpd;//ring speed =C_Targ*50, 50: crystal oscillator speed
29
30     parameter xtalSpd = 6'd50; //50 MHz
31     //ringSpd=C_RX*7'd50*100;
32     assign ringSpd = C_RX*xtalSpd;
33     assign intDvs1 = C_N[15:7];
34     assign intDvs2 = C_N[31:23];
35     assign fltDvs1 = C_N[6:0];
36     assign fltDvs2 = C_N[22:16];
37
38     //dvs = intDvs*7'd100+fltDvs;
39     assign dvs1 = intDvs1*7'd100+fltDvs1;
40     assign dvs2 = intDvs2*7'd100+fltDvs2;
41     //ringSpd/dvs=intF;
42     //module Number_Division(Dividend,Divisor,Result);
43     Number_Division intFinalOne (ringSpd,dvs1,intF1);
44     Number_Division intFinalTwo (ringSpd,dvs2,intF2);
45     // rmn = (ringSpd-(intF*dvs))*7'd100;
46     assign rmn1 = (ringSpd - (intF1*dvs1))*7'd100;
47     assign rmn2 = (ringSpd - (intF2*dvs2))*7'd100;
48     //rmn/dvs=fltF;
49     Number_Division fltFinalOne (rmn1,dvs1,fltF1);
50     Number_Division fltFinalTwo (rmn2,dvs2,fltF2);
51
52     assign C_N1[31:7] = intF1[25:0];
53     assign C_N1[6:0] = fltF1[6:0];
54     assign C_N2[31:7] = intF2[25:0];
55     assign C_N2[6:0] = fltF2[6:0];
56
57  endmodule
```

Controller: Ring Oscillator

Date: May 21, 2023 Ring.sv Project: ADPLL_5_3

```
1   //Zachary Weiss
2
3   `timescale 1ps/1ps
4
5   module Ring (Clk, volt_0, volt_1, F_ring);
6
7   input Clk, volt_0, volt_1;
8   output F_ring;
9   logic Q, Q_bar;
10
11  //voltages that enable ring oscillator operation
12
13
14  and (En, volt_0, volt_1);
15
16
17  //oneShot(Clk, Q, Q_bar);
18
19  oneShot DUT0 (Clk, Q, Q_bar);
20
21  //ringOscillator(En, one_shot, F_ring);
22
23  //Sending Q to the Ring Oscillator means the one-shot buffer period is specifiying the
    amount of time the ring oscillaotr is off
24  //Sending Q_bar means you are specifiying the length it is on.
25  ringOscillator DUT1 (En, Q, F_ring);
26
27  endmodule
28
```

One Shot

Date: May 21, 2023 oneShot.sv Project: ADPLL_5_3

```
1   //Zachary Weiss
2
3   `timescale 1ps/1ps
4
5   module oneShot(Clk, Q, Q_bar);
6
7   input Clk;
8   output Q, Q_bar;
9   logic X, Reset  /* synthesis keep */;
10
11  parameter propogation_delay = 60ps; //Smallest possible from expected board
12
13  not #(propogation_delay)   (X, Q);
14  not #(propogation_delay)   (Reset, X);
15
16  //instantiation of flip flop
17  //DFF1 (Clk, Reset, D, Q, Q_bar);
18  DFF1 unit0 (Clk, Reset, 1'b1, Q, Q_bar);
19
20
21  endmodule
```

Asynchronous DFF

Date: May 21, 2023 DFF1.sv Project: ADPLL_5_3

```
1   //Zachary Weiss
2   //asynchronous DFF
3
4   `timescale 1ps/1ps
5
6   module DFF1 (Clk, Reset, D, Q, Q_bar);
7
8   input D, Clk, Reset;
9   output logic Q;
10  output Q_bar;
11
12  always_ff @(negedge Reset, posedge Clk)
13
14  begin
15
16  if (!Reset)
17
18  Q <= D;
19
20  else
21
22  Q <= 0;
23
24  end
25
26  assign Q_bar = ~Q;
27
28  endmodule
29
```

Ring Oscillator

Date: May 21, 2023 ringOscillator.sv Project: ADPLL_5_3

```
1   //Zachary Weiss
2
3   `timescale 1ps/1ps
4
5   /*first gate is a NAND w/ Enable
6   all consecutive NOT gates in series
7   total number of gates must be odd therefore we generate an even number of NOT gate
8   We must manually add time delay since simulated inverter has propogation delay of 0*/
9
10  //NOTE: RTL shows 3 inverters because synthesis converters NAND into AND in series with
    inverter
11
12
13  module ringOscillator(En, one_shot, F_ring);
14
15     input one_shot, En;
16     output F_ring;
17     //number of inverters minimum possible 2
18     logic [2:0] inverter /* synthesis keep */;
19     parameter propogation_delay = 30ps;
20
21     nand #(propogation_delay)  (inverter[0], one_shot, inverter[2], En);
22     not #(propogation_delay)   (inverter[1], inverter[0]);
23     not #(propogation_delay)   (inverter[2], inverter[1]);
24
25     assign F_ring = inverter[2];
26
27  endmodule
28
```

Calculators: Divider

Date: May 21, 2023 Divider.sv Project: ADPLL_5_3

```
1   `timescale 1ps/1ps
2   module Divider(En,Reset,C_N,F_input,F_output);
3
4   input En, Reset;
5   input [31:0] C_N;
6   input F_input;
7   output reg F_output;
8
9   reg [31:0] Natural, Decimal, Half_point1, Half_point2;
10  reg[15:0] count_1 = 16'd0;
11  reg [15:0] count_p = 16'd0;
12  reg [15:0] Divisor;
13
14  always@(posedge F_input)begin
15
16  //calculate Natural Decimal and Halfway
17     Natural = C_N[31:7];
18     Decimal = C_N[6:0];
19     Half_point1 = (Natural + 16'd1) >> 1; //Greater half point
20     Half_point2 = Natural >> 1; //Lesser half point
21     count_1 <= count_1 + 16'd1;
22  //if count_p < Decimal than Natural+1 else Natural
23  //This if statement is for first x/100 cycles where x is the decimal number. The frequency
    of these cycles will
24  //be 1 greater than the whole number.
25     if(Decimal>count_p)begin
26        Divisor = Natural + 16'd1; //Set our Divisor to one greater than natural.
27        if(Half_point1 >= count_1)begin
28           F_output = 1;
29        end
30
31        else if(count_1 > (Half_point1))begin
32           F_output = 0;
33        end
34
35        if(count_1 >= Divisor)begin //We have gone through one full cycle of Clk_1
36           count_p <= count_p + 16'd1;
37           count_1 <= 16'd1;
38        end
39     end
40  //This statement is for the last (100-x)/100 cycles with a frequency == to the whole
    number. if number is whole, this is the
41  //only cycle that will run.
42     else begin
43        Divisor = Natural;
44        if(Half_point2 >= count_1)begin
45           F_output = 1;
46        end
47
48        else if(count_1 > (Half_point2))begin
49           F_output = 0;
50        end
51
52        if(count_1 >= (Divisor))begin
53           count_p <= count_p + 16'd1;
54           count_1 <= 16'd1;
55        end
56        if(count_p >= 16'd100)begin
57           count_p <= 16'd0;
58        end
59     end
60  end
61
62  endmodule
63
```

Calculators: Ratio


```
1
2
3   // This unit measure the frequency of the input and return as a positive integer.
4   //reference clock(refClk) is the input frequency, which is slower than the ring oscillator
    generated clock(clk)
5   //use average to get more accurate difference.
6   `timescale 1ps/1ps
7   module Freq_ratioA(En,Reset,F_clk,F_ring,C_freq);
8      input F_clk,F_ring;//F_clk: frequency of input clock; F_ring: frequency of a ring
    oscillator.
9      input En, Reset; // Enable signal, Reset signal.
10     logic combReset;//high reset
11     output logic [31:0]C_freq=32'b0;       //this is the output connected to the logic module
12     logic [6:0]clkCounter=7'b0;   // this is connected to an output for debug purpose. We
    will change this to a internal reg later.
13     logic [31:0]ringCounter=32'b0;   //this is connected to an output for debug purpose. We
    will change this to a internal reg later.
14
15
16     parameter phaseAvg=7'd100; // this is controlling condition to measure certein numbers
    of slower clock. For other modules' design it should be 100.
17              //if clkCounter = phaseAvg; C_freq = ringcounter;
18
19
20
21     always_ff @(posedge F_clk or posedge combReset or negedge En) begin
22        //(neg) hold when enable goes 0
23        if(!En) begin
24           clkCounter <= clkCounter;
25        end
26        //(pos) reset the counter
27        else if(combReset) begin
28           clkCounter <= 7'b0;
29        end
30        //add clkCounter 1'b1.
31        else begin
32           if(clkCounter == 7'b1111111) begin
33              clkCounter <= 7'b0;
34           end
35           else begin
36              clkCounter <= clkCounter +7'b1;
37           end
38        end
39
40
41     end
42
43
44
45     always_ff @(posedge F_ring or posedge combReset or negedge En) begin
46        if(!En) begin
47           ringCounter <= ringCounter;
48        end
49
50        else if(combReset) begin
51           ringCounter <= 9'b0;
52        end
53
54        else begin
55           if(ringCounter == 32'hFFFFFFFF) begin
56              ringCounter <= 32'b0;
57           end
58           else if (ringCounter == 32'd100)begin
59              ringCounter[31:7] <= ringCounter[31:7]+25'd1;
60              ringCounter[6:0] <= 7'd0;
61
62           end
63           else begin
64              ringCounter <= ringCounter +9'b1;
```

```
65            end
66
67         end
68
69      end
70
71      always @(posedge F_clk or posedge Reset or negedge En) begin
72         if(!En) begin
73            C_freq      <= C_freq;
74            combReset   <=combReset;
75         end
76
77         else if(Reset) begin
78            C_freq      <= 1'b0;
79            combReset   <=1'b1;
80         end
81
82         else begin
83            if ( clkCounter == phaseAvg-1) begin
84               C_freq   <= ringCounter;      //save ringCounter to C_freq now
85               combReset   <= 1'b1;       //set reset to 1 now
86            end
87            else begin
88               C_freq   <= C_freq;
89               combReset   <= 1'b0;
90            end
91         end
92      end
93
94   endmodule
```

Calculators: Multiplier

Date: May 21, 2023 Multiple_Calculator.sv Project: ADPLL_5_3

```
1   //Author: Dylan
2
3
4   //Takes a control signal Target_Freq. This is a 32 bit number representing two 16 bit
    numbers with 7 of those bits representing the decimal places.
5   //Target Freq should be in Mhz so you can currently target anywhere from 511.99 Mhz to 0.01
    Mhz (or 10 kilohertz)
6   //Requires Ratio of how many times faster that the Crystal oscillator is from the input. If
    crystal is 2.5 times faster Ratio_Crystal should equal 250.
7   //Crystal frequecy has been hardcoded in as 50 Mhz on line 18, if needed this value can be
    changed.
8   //Will calculate a multiplier to be sent to Divider module this number again can range from
    511.99 to 0.01
9
10  //modified: Jisu
11  //similar thing to Controller.
12
13  `timescale 1ps/1ps
14  module Multiple_Calculator (En, Reset,Clk, Ratio_Crystal, Target_Freq, Multiplier);
15
16  input En, Reset,Clk;
17  input reg [31:0] Ratio_Crystal;
18  input reg [31:0] Target_Freq;
19  output reg [31:0] Multiplier;
20  logic [31:0] T_Multiplier, Temp_1, Temp_2, Temp_3, Temp_4, Temp_5, Temp_6, Temp_7, Temp_8;
21  reg [31:0] T1,T4,T2;
22  reg [8:0] Whole_1, Whole_2, Whole_1_T, Whole_2_T;
23  reg [6:0] Dec_1, Dec_2, Dec_1_T, Dec_2_T;
24
25  parameter C_Crystal = 50; //50 MHz
26
27  Number_Division DUT1 ((Whole_1_T*Ratio_Crystal),C_Crystal,Temp_1);
28
29  Number_Division DUT2 (Temp_1,100,T1);
30  Number_Division DUT3 ((Dec_1_T*Ratio_Crystal),(100*C_Crystal),Temp_3);
31  Number_Division DUT4 (Temp_1,100,T_Multiplier[15:7]);
32  Number_Division DUT5 ((Whole_2_T*Ratio_Crystal),C_Crystal,Temp_4);
33  Number_Division DUT6 (Temp_4,100,T4);
34  Number_Division DUT7 ((Dec_2_T*Ratio_Crystal),(100*C_Crystal),Temp_6);
35  Number_Division DUT8 (Temp_4,100,T_Multiplier[31:23]);
36
37  always @(posedge Clk or posedge Reset or negedge En)begin
38
39     if(!En) begin
40        /*Dec_1  <= Dec_1;
41        Whole_1  <= Whole_1;
42        Dec_2    <= Dec_2;
43        Whole_2 <= Whole_2;
44        Temp_1   <= Temp_1;
45        Temp_2   <= Temp_2;
46        Temp_3   <= Temp_3;
47        Temp_4   <= Temp_4;
48        Temp_5   <= Temp_5;
49        Temp_6   <= Temp_6;
50        T_Multiplier <= T_Multiplier;
51        Multiplier  <= Multiplier;
52        Whole_1_T = Whole_1_T;
53        Dec_1_T = Dec_1_T;
54        Whole_2_T = Whole_2_T;
55        Dec_2_T = Dec_2_T;
56        */
57     end
58
59     else if(Reset)begin
60        /*Dec_1  <= 1'b0;
61        Whole_1  <= 1'b0;
62        Dec_2    <= 1'b0;
63        Whole_2 <= 1'b0;
64        */
```

```
65          /*Temp_1    <= 1'b0;
66          Temp_2   <= 1'b0;
67          Temp_3   <= 1'b0;
68          Temp_4   <= 1'b0;
69          Temp_5   <= 1'b0;
70          Temp_6   <= 1'b0;
71          T_Multiplier<= 32'b0;
72          Multiplier  <= 32'b0;
73          Whole_1_T <= 9'b0;
74          Dec_1_T <= 7'b0;
75          Whole_2_T <= 9'b0;
76          Dec_2_T <= 7'b0;
77          */
78       end
79
80       else begin
81          Dec_1 = Target_Freq[6:0];
82          Whole_1 = Target_Freq[15:7];
83          Dec_2 = Target_Freq[22:16];
84          Whole_2 =Target_Freq[31:23];
85
86          /*Whole_1_T = Whole_1;
87          Dec_1_T = Dec_1;
88          Whole_2_T = Whole_2;
89          Dec_2_T = Dec_2;
90          */
91
92          //First Multiplier
93          //Temp_1 = (Whole_1*Ratio_Crystal)/(C_Crystal); //85
94          //Temp_2 = Temp_1-(Temp_1/100)*100;
95          Temp_2 = Temp_1-((T1)*100); //85
96
97          //Temp_3 = (Dec_1*Ratio_Crystal)/(100*C_Crystal);//2
98
99          Temp_7 = Temp_3 + Temp_2;
100         if(Temp_7 >= 100)begin
101            Temp_7 = Temp_7-100;
102            Temp_1 = Temp_1 + 100;
103
104         end
105         T_Multiplier[6:0] = Temp_7; //87
106         //T_Multiplier[15:7] = Temp_1/100; //0
107
108   //Same thing for second number
109         //Temp_4 = (Whole_2*Ratio_Crystal)/(C_Crystal);//125
110         //Temp_5 = Temp_4 -(Temp_4/100)*100;
111         Temp_5 = Temp_4 -(T4*100);//25
112         //Temp_6 = (Dec_2*Ratio_Crystal)/(100*C_Crystal);//3
113
114         Temp_8 = Temp_6 + Temp_5; //28
115         if(Temp_8 >= 100)begin
116            Temp_8 = Temp_6 - 100;
117            Temp_4 = Temp_4 + 100;
118
119         end
120         T_Multiplier[22:16] = Temp_8;
121         //T_Multiplier[31:23] = Temp_4/100;
122
123         Multiplier = T_Multiplier;
124      end
125
126   end
127
128   endmodule
129
```

Calculators: Long Division

Date: May 21, 2023 Number_Division.sv Project: ADPLL_5_3

```
1
2   // Code created by vipin on
    https://verilogcodes.blogspot.com/2015/11/synthesisable-verilog-code-for-division.html
3
4   module Number_Division(A,B,Res);
5
6       //the size of input and output ports of the division module is generic.
7       parameter WIDTH = 32;
8       //input and output ports.
9       input [WIDTH-1:0] A;
10      input [WIDTH-1:0] B;
11      output [WIDTH-1:0] Res;
12      //internal variables
13      reg [WIDTH-1:0] Res = 0;
14      reg [WIDTH-1:0] a1,b1;
15      reg [WIDTH:0] p1;
16      integer i;
17
18      always@ (A or B)
19      begin
20          //initialize the variables.
21          a1 = A;
22          b1 = B;
23          p1= 0;
24          for(i=0;i < WIDTH;i=i+1)    begin //start the for loop
25              p1 = {p1[WIDTH-2:0],a1[WIDTH-1]};
26              a1[WIDTH-1:1] = a1[WIDTH-2:0];
27              p1 = p1-b1;
28              if(p1[WIDTH-1] == 1)    begin
29                  a1[0] = 0;
30                  p1 = p1 + b1;   end
31              else
32                  a1[0] = 1;
33          end
34          Res = a1;
35      end
36
37  endmodule
```