Syracuse University

SURFACE

Syracuse University Honors Program Capstone Projects

Spring 5-1-2009

Synthesis Minimizations and Mesh Algorithm Selection: An Extension of the Ultrasonic 3D Camera

Taylor Johnson

Follow this and additional works at: https://surface.syr.edu/honors_capstone

Part of the Digital Circuits Commons, and the Other Computer Engineering Commons

Synthesis Minimizations and
Mesh Algorithm Selection:
An Extension of the
Ultrasonic 3D Camera

A Capstone Project Submitted in Partial Fulfillment of the Requirements
of the Renée Crown University Honors Program at Syracuse University

Taylor Johnson

Candidate for B.S. Degree
and Renée Crown University Honors

May 2009

Honors Capstone Project in Computer Engineering

Capstone Project Advisor: _____
Duane Marcy

Honors Reader: _____
Ehat Ercanli

Honors Director: _____
Samuel Gorovitz

Date: _____

Abstract

Statement of Purpose

The purpose of this Capstone project was to perform synthesis minimizations and optimal mesh algorithm selection for some of the digital components of a prototype Ultrasonic 3D Camera, the subject of my group senior design project for computer engineering. Both of the high-level design tasks that I performed were unnecessary for the scope of the senior design class, whose focus was simply to perform a proof-of-concept or create a basic, functioning prototype. The steps I took in performing synthesis optimizations and mesh algorithm selection went beyond the scope of the senior project, by doing the polishing that would be most suited for a project that was eventually going to be turned into an actual product.

Design Methodology

Synthesis optimizations were performed using the Altera Quartus II software, available to me as a student of the L.C. Smith College of Engineering and Computer Science. With the assistance of the Quartus II sofware, I synthesized the behavioral VHDL code written for the project, and inspected the resulting netlists for places where additional logic was being unnecessarily included. In the places where unnecessary logic was found, I eliminated it, by rewriting behavioral code as structural, or by simply clarifying the definition of components.

Mesh algorithm selection was performed using the Microsoft Visual Studio IDE (Integrated Design Environment) in conjunction with the OGRE 3D graphics engine. Using Visual Studio and OGRE, I was able to experiment with different mesh-forming algorithms and determine which method was best suited for the Camera, given the nature of the incoming data.

Outcome

The VHDL code for the project is now optimized for synthesis, such that if my group were to take steps to have the VHDL code synthesized into a netlist, and then eventually into a mask, and then turned into an actual integrated circuit, that circuit would be almost minimally small, while still performing all its necessary functions. Likewise, the mesh algorithm selected, the naïve method, works perfectly well with the nature of the data that the Camera obtains.

Table of Contents

Introduction

The Ultrasonic 3D Camera is a device that creates a three-dimensional model of an object or surface, by using a phased array of ultrasonic transducers to collect distance data, and then using that distance data to create a three-dimensional mesh representation of the object or surface. The first part of this document describes the design and operation of the Ultrasonic 3D Camera, which was designed as a group project for the computer engineering senior design class, CSE 497. The latter part of this document describes my additional individual contributions to the project, which went above and beyond the scope of the senior design project, and formed the basis of my Capstone project.

Operating Principles

Transducers

The physical components that enable a device like the Ultrasonic 3D Camera to be a feasible are ultrasonic transducers. A transducer, broadly speaking, is "a device for converting energy from one form to another for the purpose of measurement of a physical quantity or for information transfer"[1]. In the case of the Ultrasonic 3D Camera, ultrasonic transducers are used. Transmitting ultrasonic transducers convert energy in the form of an electrical signal (specifically, a 40kHz square wave) into compressional ultrasonic waves that propagate through air. Receiving

ultrasonic transducers do the inverse conversion – from compressional ultrasonic waves back into an electrical signal.

A single ultrasonic transmitter/receiver pair can be used to do simplistic ultrasonic ranging, but for more complicated data collection, the type of which is required for the Ultrasonic 3D Camera, a phased array of ultrasonic transducers is needed. In a phased array arrangement, a collection of transducers is placed together such that they will work cohesively to perform beam forming and beam steering.

Beam Forming

For a phased array of ultrasonic transducers to perform the task of producing directed sound, the individual elements of the phased array must interact in a way that is complementary. Otherwise, the array would simply be a collection of individual elements, and would not be of much use for strategically directing sound.

To understand the way multiple elements can interact, consider the analogy of two identical stones being dropped in a pool of still water at the exact same time, from the exact same height, at some distance apart. Each stone creates an outward ripple, and after some amount of time the ripples of the two stones will encounter one another. In every place where the ripples meet, they will combine to form a ripple whose amplitude is the sum of the amplitude of the individual ripples. In some places, the amplitudes will be equal but opposite and the ripples will

cancel each other out completely. It can be said that be said that the ripples constructively interfere in places where the combined ripple has a non-zero amplitude, and destructively interfere when the resultant amplitude is zero.[2]

Let's step back into the world of transducers. The analogy is structured the following way: the stones are transmitting transducers, the pool of water is the air, the height from which the stones are dropped is the amplitude of the signal exciting the transducers, and the time at which the stones hit the water is the time that the transducers begin transmitting. Constructive and destructive interference also occurs with the sound emitted from ultrasonic transducers, and it is this exact behavior that the phased array of ultrasonic transducers uses to its advantage. In order to focus the sound emitted from transducers into a beam, we want the sound emitted from all 20 transducers to constructively interfere where we want the beam to be formed, and destructively interfere (or constructively interfere as little as possible) everywhere else. This process is commonly referred to as beam forming.

Beam Steering

While beam forming is vital to the operation of any phased array, it would not be of much use beyond precision ranging unless it was paired with beam steering, a common way of referring to the act of changing the direction in which the formed beam is emitted. Consider

again the scenario with the two stones and the pool of water. In the original scenario, the stones were dropped at the exact same time. Now, consider the situation where the stones are dropped at different times. This time, when the ripples from the two stones meet, they will again interfere, but in a different way than in the previous scenario. Specifically, the combined ripple will propagate most strongly in an angled direction angled away from the line that can be drawn between the two points.

Let's move again back to the phased array of ultrasonic transducers. When all of the transmitting transducers in the array are excited at the same time, the resulting beam of sound is directed along the normal to the plane in which the array exists. When the transducers are excited at different times, a beam is still formed, but the direction in which that beam propagates is no longer normal to the array, but some number of degrees off of the normal, in the x-direction, y-direction, or both. This effect is called beam steering, and is the physical basis for the functionality of this project.

Data Collection

Now that we understand how beam forming and beam steering work, we can address exactly how the ultrasonic transducers collect data. The basic narrative of pinging is the following:

1. As soon as a ping is initiated, a timer begins counting and the transducers begin to be excited by the 40kHz square wave.

2. A beam of ultrasound is formed and steered in the direction of the intended sample.

3. The transducers stop being excited as soon as enough time has elapsed for a beam to form.

4. The beam propagates through the air, eventually contacting the object that is the subject of the scan. While most of the sound is absorbed (the amount of absorption depends on the acoustic nature of the material), some of the sound is reflected back towards the array.

5. As soon as the reflection of the ping is detected by one of the receiving transducers, the timer stops counting.

6. The distance of the object that reflected the ping is calculated using the half the timer value (since the sound traveled out and then back) and the speed of sound.

To collect as much data as possible, the Ultrasonic 3D Camera repeats this procedure again and again, iterating through every combination of all possible x-angles and y-angles. With a maximum range of 60 degrees in any direction off of the normal, and a resolution of 0.288 degrees, the array can collect a maximum of (416) * (416) = 173, 056 data points. This is a large amount of data, especially for the size of the objects that the Ultrasonic 3D Camera is meant to scan. One way to speed up the amount of time it takes to perform a full scan is to only iterate through some of the possible angles, which will keep the amount of data down.

System Partitioning

Many things besides transducers are needed to construct a functioning Ultrasonic 3D Camera. The non-transducer elements of the system are partitioned into the following modules:

- Analog Driving Network – the analog components necessary to provide a signal capable of exciting the transmitting transducers

- Analog Receiving Network – the analog components necessary to take the signal detected by the receiving transducers and transform it into a digital representation

- Digital Driving Network – the digital components necessary to perform beam forming and beam steering with the phased array of ultrasonic transducers

- Digital Receiving Network – the digital components necessary to detect when a reflection of a beam has returned to the array

- Modeling Software – the software used to create and model meshes based on the three-dimensional data points collected by the Camera

Module Explanation

In this section, the purpose and operation of each of the individual system modules is explained. The most detailed sections are the digital driving network and digital receiving network, as they are the focus of the synthesis minimization component of this Capstone project.

Analog Driving Network

The analog driving network consists of twenty identical amplifying and conditioning circuits, one for each of the driving transducers. The analog driving network is necessary because the signals that excite the transducers must be appropriately conditioned in order to provide the maximum energy transfer, and also to get as much range out of the transducer array as possible.

The following is a detailed description of the amplification process that each of the transducer excitation signals goes through: A signal is initially emitted from the FPGA as a 3.3v square wave. This 3.3v square wave is given as input to a Schmidt trigger, and is then output from the Schmidt trigger as a stronger, 5v square wave (the Schmidt triggers are powered by a 5v regulator). The signal is then sent to the gate of an nMOS transistor, whose source is tied to ground and whose gate is tied to a 100Ω resistor, the other end of which is tied to the supply voltage of 30v. At this point, the signal at the drain of the transistor is a strong 30v

square wave. A transducer is placed in parallel with the nMOS transistor, and the final result of the configuration is that the transducers are each driven by their very own 30v square wave.

Analog Receiving Network

The analog receiving network consists of four identical amplification circuits, one attached to each of the four receiving transducers. Amplifiers are needed on the receiving end of the system because the reflection of the beam of sound that returns to the array will be extremely weak, on account of the fact that most of the sound is absorbed by the air and the object which reflected the beam. The specific job of the receiving amplifiers is to take a signal that is as low as 50mv and turn it into a 3.3v square wave so that the digital receiving network can detect the arrival of the reflection. To do this, each signal received by a receiving transducer is amplified using an active band-pass filter with the band centered around 40kHz, and then sent to a Schmidt trigger, which digitizes the signal into a variable-duty cycle square wave.

Digital Driving Network

The digital driving network consists of everything necessary to perform intermittent pinging (phase-shifted and non-phase-shifted) of the transmitting ultrasonic transducers. Some components will have 20 instances (one for each transmitting transducer), and other components will only have one instance for the entire digital driving network.

The first component, of which there is only one instance, is a frequency divider. The FPGA supplies a 50MHz clock, but the ultrasonic transducers need to be driven with a 40kHz, and so a frequency divider is used to create a 40kHz signal from a 50MHz signal. To accomplish this, the first thing to do is calculate how many cycles of a 50MHz signal occur during one cycle of a 40kHz signal. 50MHz / 40kHz = 1250. Now, to accomplish the frequency division, it is simply a matter of counting 625 cycles of the 50Mhz signal, then setting the 40kHz signal high, then counting another 625 cycles of the 50Mhz signal, then setting the 40kHz signal low again, and then repeating the process. In other words, simply invert a signal every 625 cycles of the 50MHz clock, and the resulting signal will be a 40kHz square-wave signal, which is exactly what is needed to drive the transmitting ultrasonic transducers. After a 40kHz signal is obtained, it needs to be distributed to the each of the transmitting transducers. However, since the array needs to be pinging the transducers instead of driving them constantly, some additional digital components are needed.

The next component needed is a timer, a component that begins counting at zero whenever the array is going to begin pinging, and increments by one for every rising edge of the 50Mhz clock signal. This specific timer also should stay at its maximum value of 0xFFFFFFFF once reached, instead of looping back around to 0 and counting up again. This would cause the array to ping again prematurely. The timer is another

component that only needs to be included once in the entire driving

network. [I initially thought that a separate timer needed to be included

for each of 20 drivers in the driving network. However, after synthesizing

the driving network for the first time, I noticed the inclusion of all of the

timers, and it occurred to me that only one driver was necessary. I

should've noticed this earlier, but synthesis was necessary for me to

recognize this problem. One of the first optimizations I made after

performing synthesis was correcting this problem.]

The next component needed is a comparator, a component that

outputs a low signal or a high signal, based on the relative values of its

two inputs. In the digital network, two comparators are actually needed,

and instances of both are required for each of the 20 transmitting

transducers. One type of comparator outputs a high signal when input A

is greater or equal to input B, and a low signal otherwise. The other type

of comparator outputs a high signal when input A is less than or equal to

input B, and a low signal otherwise. [It is tempting to think that one type

of comparator could be used for both cases here, by simply flipping the

inputs, but the result of this would change a non-strict inequality into a

strict inequality, which is not the desired functionality for these two

comparisons. A compromising solution could be to create a comparator

component which is strict or non-strict depending on a control input, but

this would require additional controlling logic, not to mention that there

would still be two instances of this component, so no space would be saved by adding this extra functionality.]

The next component is a 6-to-40 decoder, a component that takes a 6-bit input and sets only one of its output bits high, based on the value of the 6-bit input. There is only one instance of the decoder in the digital driving network.

The final component needed is a register, a component with memory, that either samples the data sent to it on the rising edge of a clock, or simply maintains the value it most recently sampled, depending on the state of an enable signal. Registers are often configured to be resettable to some predetermined value, but this functionality is not necessary in this design – the registers can simply be loaded with the desired value at the most convenient time. In the digital driving network, there are two instances of the same 32-bit register for each of the 20 transducers, for a total of 40 instances.

Now that I've described all of the components of the system, I'll describe how using all of these components collectively results in successful pinging, beam forming, and beam steering:

- Two registers are instantiated for each transmitting transducer. The first register is loaded with the delay value, which indicates the number of clock cycles after the beginning of system wide pinging a particular transducer should begin pinging. The second

register is loaded with the delay value plus the duration value, so

that the particular transducer will ping for the desired duration.

- The two comparator make sure that a ping is only let through to a

  transducer when the time is between delay and duration + delay.

- Using different delays for each transducer results in beamsteering.

- The frequency divider turns the 50MHz clock of the FPGA into a

  40kHz signal. This signal is then let through to a transducer

  whenever the registers associated with the transducer indicate

  that the transducer should be pinging.

- The decoder controls which register is written to at any given

  point in time.

Digital Receiving Network

The digital receiving network consists of a 32-bit counter, a D flip

flop, and some additional logic. The purpose of the digital receiving

network is to start timing as soon as a ping is initiated, and detect that a

ping has returned to the array.

To accomplish this task, the receiver ORs together the four inputs

from the analog receiver network, so that whenever any of the receiving

transducers picks up a reflected ping, the output of the OR gate, which is

attached to the input of the D flip flop, will go high. As soon as the D flip

flop reads a '1', the timer will be inhibited, the control system will be

notified that a ping had returned, and the D flip flop will ignore all input

data until the receiving network is reset.

Modeling Software

The modeling software used for the Ultrasonic 3D Camera was

OGRE, an object-oriented 3D graphics engine. OGRE was selected because

of its relative ease of use, and abundance of publicly available code

examples and tutorials.

To get the data to OGRE, it was sent from the FPGA to a computer

via a serial port, and then sent directly to OGRE using UDP packets. Once

in OGRE, the data could be drawn as a point represented in three

dimensions.

Synthesis Minimizations

Synthesis minimization is the first part of my Capstone project. The idea behind synthesis minimization is to perform high-level synthesis on the behavioral VHDL files that I wrote for the digital transmitting and receiving networks and look for excess logic than can be removed, while keeping all functionality intact. The following section describes the synthesis results for each of the components of the digital system.

Comparators

There are two types of comparators included in the digital system, denoted comparator1 and comparator2.

comparator1

Comparators are fairly simple logic components, so when I began the synthesis process with the comparators, I didn't expect to see any places for simplification. And, sure enough, I didn't find anything unnecessary in the synthesized schematic for comparator1. There is a LESS_THAN module, directly given the two inputs, and there is also a CIN of 1, which confirms that the inequality will be of the non-strict variety.

comparator2

This comparator is very similar to the first comparator, so once again I didn't expect to see any places for simplification. Not surprisingly, there was nothing unnecessary in the synthesized schematic for

comparator2. There is again a LESS_THAN module, directly given the two inputs, and there is also a CIN of 1, which confirms that the inequality will be of the non-strict variety.

Timer

When I synthesized the timer (counter) for the digital network, I definitely expected to see some unnecessary logic – perhaps some inferred latches or something of that nature. However, to my great surprise, there was nothing of the sort. The synthesized schematic consists of a 32-bit register whose data input is tied to an adder, which adds the previous value of the register to 1. There is also an enable signal, which enables the counter until the max value of 0xFFFFFFFF is reached, at which point the counter stops counting. Finally, there is also a reset signal that is simply an input to the module. There was nothing in the schematic that could be taken out without changing the desired functionality of the module.

While I did not find any unnecessary logic within the timer module, I did notice that a timer was being instantiated for each of the twenty driving transducers. This, of course, is completely unnecessary, because of the region selection performed by the twenty sets of two comparators in the system. So, essentially, only one instance of the timer is needed for the entire digital driving network, and yet I was instantiating twenty. [Please note that this is not something that

synthesis can uniquely identify – I should have noticed this design flaw while writing the VHDL code. However, synthesis was useful in this circumstance because seeing the actual schematic was what it took to get me to realize that only one counter was necessary, instead of twenty.] As an easy optimization, I removed the 19 superfluous timers from the digital network, and saved a lot of space and hardware.

Decoder

When I synthesized the decoder for the digital driving network, I got what I expected – a long trail of logic. Luckily, all of it was necessary for the decoder to function properly. So, with no work to do here, I moved on to the next synthesis category.

Divider

When I first synthesized the divider for the digital driving network, it seemed to me to be already optimized. It was simply using a counter to count up to 624, then flip its input, and then start counting up from 0 again. However, after careful inspection I noticed that the hardware used to store the counter value was 32-bits wide, which is quite a lot of extra space for a number which only goes as high as 624. Then, I realized that I had specified the use of a regular integer, instead of an integer limited in size (in this case, to 10 bits, which is more than enough to represent 624). So, I changed the VHDL code to incorporate the

minimally sufficient integer size, in the process saving space by eliminating the need for several storage elements.

Register

The 32-bit register synthesized to exactly what I expected – simply an array of 32 D flip flops. This was exactly how a simple register should be represented – no changes were needed to this module.

Counter

This counter is essentially just like the timer mentioned earlier, except that it also has an explicit enable signal accessible to the user. Just like the timer, it synthesizes quite well – simply a 32-bit register with an incrementing input.

D Flip Flop

This D flip flop turned into exactly what I expected – a D flip flop. Flip flops are among the most basic of circuit elements, so it comes as no surprise that my description of a flip flop was turned into a flip flop.

Mesh Algorithm Selection

The second part of my Capstone project focused on the selection of a suitable mesh algorithm to use on the point clouds created by the Ultrasonic 3D Camera. There is a multitude of ways to connect a large collection of data points, but only some of those ways will result in something recognizable, or distinguishable. Mesh algorithms attempt to connect a collection of points in such a way that a "mesh" is created that highly resembles the object from which the data points were originally obtained.

The Naïve Method

The naïve method of mesh creation is fairly straightforward – it assumes that the points are ordered in the way that they were collected, and selects the first three points in the ordering, and connects them via a triangle, which in three dimensions can be seen as a triangular section of the plane in which all three of the points lie. Then, the algorithm iterates through all of the remaining points one at a time, forming a new triangular face between the new point and the two closest points that are already part of a triangle. The following pseudo-algorithm describes this process.

```
1.    Place all collected points in a "collected" linked list.
2.    Take the first three elements of the linked list, create
      a face between them, remove them from the "collected"
      linked list, and place them in the "added" linked list.
3.    While the "collected" linked list is not empty
```

```
4.     Search the "added" linked list for the two points
       closest in distance to the head of the "collected"
       linked list
5.     Create a new face between the resulting three
       points
6.     Remove the head of the "collected" linked list and
       add it to the "added" linked list
```

Configurational Energy Method

The configurational energy method of mesh creation is a great deal more complex than the naïve method. The algorithm sums the square of the edge tensions, which are based on the predetermined optimal edge length. In the case of the Ultrasonic 3D Camera, the optimal edge length is the length of the base of the cone that represents the sound emitted by the Camera. After determining these sums, the algorithm determines tightly packed local areas where the mesh should begin, and then works its way outward.

Marching Cubes Method

The marching cubes method is another complicated mesh creation algorithm that works by drawing a large cube connected to the borders of the point cloud, and then working inward with smaller and smaller cubes. Then, eventually, triangular faces are identified between the points of the inner cubes, and then the next outer cubes, and then finally back out to the outermost cube. So, cubes "march" inward, and then triangular faces force their way out to the borders using the cubes as guides.

Results

I didn't get a chance to fully test these mesh algorithms, due to the fact that the Ultrasonic 3D Camera was never entirely functional, and therefore wasn't generating point clouds with which I could experiment. I was, however, still able to get a decent idea of which mesh algorithm would be most likely to create meshes out of the point clouds created by the Camera.

After much comparison, it turns out that the naïve method will work just fine for our purposes. What I should've realized earlier is that most sophisticated mesh algorithms are meant to deal with situations where the mesh has little to no structural regularity. That is, they are meant to function without any concept of the order in which points were collected. The naïve method, however, works extremely well for particular collection orders. Since we can control the order of data collection simply by controlling the array, it follows that the best approach would be to choose a data collection ordering which aligns nicely with the nature of the naïve method. This approach should yield a mesh algorithm that does a good job of recreating the original subject of the scan, while being quite easy to implement.

Sources Cited and Consulted

Citations:

1. Institute for Telecommunication Sciences. "Transducers."
   http://www.its.bldrdoc.gov/fs-1037/dir-037/_5539.htm

2. Henderson, Tom. "Behavior of Waves."
   glenbrook.k12.il.us/GBSSCI/PHYS/Class/waves/u10l3c.html

Consultants:

Dr. Duane Marcy
Dr. Fred Phelps
Dr. Ehat Ercanli
William Tetley
Charles Slominski

Appendix A: VHDL source files for the digital system

comparator1

```
-- comparator1.vhd
-- a comparator for the count and delay values
library ieee;
use ieee.std_logic_1164.all;

entity comparator1 is
  port (count : in std_logic_vector(31 downto 0);
        delay : in std_logic_vector(31 downto 0);
        comparison : out std_logic);
end comparator1;

architecture behavioral of comparator1 is
begin
  process (count, delay)
  begin
    if count >= delay then
      comparison <= '1';
    else
      comparison <= '0';
    end if;
  end process;
end behavioral;
```

comparator2

```
-- comparator2.vhd
-- a comparator for the count and duration values
library ieee;
use ieee.std_logic_1164.all;

entity comparator2 is
  port (count : in std_logic_vector(31 downto 0);
        duration : in std_logic_vector(31 downto 0);
        comparison : out std_logic);
end comparator2;

architecture behavioral of comparator2 is
begin
  process (count, duration)
  begin
    if count <= duration then
      comparison <= '1';
    else
      comparison <= '0';
    end if;
  end process;
end behavioral;
```

counter

```
-- counter.vhd
-- a timer for keeping track of how long a ping
-- takes to return to the array
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
  port (clk : in std_logic;
        reset : in std_logic;
        count : out std_logic_vector(31 downto 0));
end counter;

architecture behavioral of counter is
  signal count_internal : std_logic_vector(31 downto 0)
    := x"00000000";
begin
  process (clk, reset)
  begin
    if reset = '1' then
      count_internal <= x"00000000";
    elsif clk'event and clk ='1' then
      if count_internal = x"FFFFFFFF" then
        count_internal <= count_internal + x"00000000";
      else
        count_internal <= count_internal + x"00000001";
      end if;
    end if;
  end process;
  count <= count_internal;
end behavioral;
```

divider

```
-- divider.vhd
-- a frequency divider, that divides by 1250
-- 50MHz is turned into 40kHz
library ieee;
use ieee.std_logic_1164.all;

entity divider is
  port (in_clock : in std_logic;
        reset : in std_logic;
        out_clock : out std_logic);
end divider;

architecture behavioral of divider is
  signal cycles : integer range 0 to 624 := 0;
  signal clk : std_logic := '0';
begin
  process (in_clock, reset)
  begin
    if reset = '1' then
      cycles <= 0
```

```
      clk <= '0'
    elsif in_clock'event and in_clock = '1' then
      if cycles = 624 then
        clk <= not clk;
        cycles <= 0;
      else
        cycles <= cycles + 1;
      end if;
    end if;
  end process;
  out_clock <= clk;
end behavioral;
```

register

```
-- reg.vhd
-- a 32-bit register, with enable
library ieee;
use ieee.std_logic_1164.all;

entity reg is
  port (input : in std_logic_vector(31 downto 0);
        clk : in std_logic;
        enable : in std_logic;
        output : out std_logic_vector(31 downto 0));
end reg;

architecture behavioral of reg is
  signal intermediate : std_logic_vector(31 downto 0) :=
x"00000000";
begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if enable = '1' then
        intermediate <= input;
      end if;
    end if;
  end process;
  output <= intermediate;
end behavioral;
```

decoder

```
-- decoder.vhd
-- a 6-to-40 decoder
library ieee;
use ieee.std_logic_1164.all;

entity decoder is
  port (input  : in  std_logic_vector(5 downto 0);
        output : out std_logic_vector(39 downto 0));
end decoder;

architecture behavioral of decoder is
```

```
begin
  process (input)
  begin
    case input is
      when "000000" =>
        output <= "000000000000000000000000000000000001";
      when "000001" =>
        output <= "000000000000000000000000000000000010";
      when "000010" =>
        output <= "000000000000000000000000000000000100";
      when "000011" =>
        output <= "000000000000000000000000000000001000";
      when "000100" =>
        output <= "000000000000000000000000000000010000";
      when "000101" =>
        output <= "000000000000000000000000000000100000";
      when "000110" =>
        output <= "000000000000000000000000000001000000";
      when "000111" =>
        output <= "000000000000000000000000000010000000";
      when "001000" =>
        output <= "000000000000000000000000000100000000";
      when "001001" =>
        output <= "000000000000000000000000001000000000";
      when "001010" =>
        output <= "000000000000000000000000010000000000";
      when "001011" =>
        output <= "000000000000000000000000100000000000";
      when "001100" =>
        output <= "000000000000000000000001000000000000";
      when "001101" =>
        output <= "000000000000000000000010000000000000";
      when "001110" =>
        output <= "000000000000000000000100000000000000";
      when "001111" =>
        output <= "000000000000000000001000000000000000";
      when "010000" =>
        output <= "000000000000000000010000000000000000";
      when "010001" =>
        output <= "000000000000000000100000000000000000";
      when "010010" =>
        output <= "000000000000000001000000000000000000";
      when "010011" =>
        output <= "000000000000000010000000000000000000";
      when "010100" =>
        output <= "000000000000000100000000000000000000";
      when "010101" =>
        output <= "000000000000001000000000000000000000";
      when "010110" =>
        output <= "000000000000010000000000000000000000";
      when "010111" =>
        output <= "000000000000100000000000000000000000";
      when "011000" =>
        output <= "000000000001000000000000000000000000";
      when "011001" =>
        output <= "000000000010000000000000000000000000";
      when "011010" =>
```

```
        output <= "0000000000000100000000000000000000000000";
      when "011011" =>
        output <= "0000000000001000000000000000000000000000";
      when "011100" =>
        output <= "0000000000010000000000000000000000000000";
      when "011101" =>
        output <= "0000000000100000000000000000000000000000";
      when "011110" =>
        output <= "0000000001000000000000000000000000000000";
      when "011111" =>
        output <= "0000000010000000000000000000000000000000";
      when "100000" =>
        output <= "0000000100000000000000000000000000000000";
      when "100001" =>
        output <= "0000001000000000000000000000000000000000";
      when "100010" =>
        output <= "0000010000000000000000000000000000000000";
      when "100011" =>
        output <= "0000100000000000000000000000000000000000";
      when "100100" =>
        output <= "0001000000000000000000000000000000000000";
      when "100101" =>
        output <= "0010000000000000000000000000000000000000";
      when "100110" =>
        output <= "0100000000000000000000000000000000000000";
      when "100111" =>
        output <= "1000000000000000000000000000000000000000";
      when others =>
        output <= "0000000000000000000000000000000000000000";
    end case;
  end process;
end behavioral;
```

flip flop

```
-- flop.vhd
-- a D flip-flop, with reset
library ieee;
use ieee.std_logic_1164.all;

entity flop is
  port (input : in std_logic;
        clk : in std_logic;
        reset : in std_logic;
        output : out std_logic);
end flop;

architecture behavioral of flop is
  signal intermediate : std_logic := '0';
begin
  process (clk, reset)
  begin
    if reset = '1' then
      intermediate <= '0';
    elsif clk'event and clk = '1' then
      intermediate <= input;
```

```
      end if;
    end process;
    output <= intermediate;
end behavioral;
```

driver

```
-- driver.vhd
-- a driver, consisting of two registers and two comparators
library ieee;
use ieee.std_logic_1164.all;

entity driver is
  port (clock : in std_logic;
        reset : in std_logic;
        data : in std_logic_vector(31 downto 0);
        count : in std_logic_vector(31 downto 0);
        ping_in : in std_logic;
        reg_enable : in std_logic;
        delay_enable : in std_logic;
        duration_enable : in std_logic;
        ping_done : out std_logic;
        ping_out : out std_logic);
end driver;

architecture structural of driver is

signal delay_out : std_logic_vector(31 downto 0); -- a
signal duration_out : std_logic_vector(31 downto 0); -- b
signal comparator1_out : std_logic;       -- c
signal comparator2_out : std_logic;       -- d
signal enable1 : std_logic;
signal enable2 : std_logic;

component reg is
  port (input  : in std_logic_vector(31 downto 0);
        clk    : in std_logic;
        enable : in std_logic;
        output : out std_logic_vector(31 downto 0));
end component;

component comparator1 is
  port (count      : in  std_logic_vector(31 downto 0);
        delay      : in  std_logic_vector(31 downto 0);
        comparison : out std_logic);
end component;

component comparator2 is
  port (count      : in  std_logic_vector(31 downto 0);
        duration   : in  std_logic_vector(31 downto 0);
        comparison : out std_logic);
end component;

begin  -- structural

enable1 <= (reg_enable and delay_enable);
```

```
enable2 <= (reg_enable and duration_enable);
C1 : reg port map (data, clock, enable1, delay_out);
C2 : reg port map (data, clock, enable2, duration_out);
C4 : comparator1 port map (count, delay_out, comparator1_out);
C5 : comparator2 port map (count, duration_out,
comparator2_out);

ping_done <= not comparator2_out;
ping_out <= ((comparator1_out and comparator2_out) nand
ping_in);

end structural;
```

pinger

```
-- pinger.vhd
-- the pinger consists of 20 drivers, a decoder, a divider,
-- and a timer
library ieee;
use ieee.std_logic_1164.all;

entity pinger is
  port (clock : in std_logic;
        data : in std_logic_vector(31 downto 0);
        address : in std_logic_vector(5 downto 0);
        reset : in std_logic;
        enable : in std_logic;
        pings : out std_logic_vector(19 downto 0);
        pings_completed : out std_logic);
end pinger;

architecture structural of pinger is
  signal register_enables : std_logic_vector(39 downto 0);
  signal completed_pings : std_logic_vector(19 downto 0);
  signal counter_out : std_logic_vector(31 downto 0);
  signal ping_clock : std_logic;

  component counter is
    port (clk   : in  std_logic;
          reset : in  std_logic;
          count : out std_logic_vector(31 downto 0));
  end component;

  component driver is
    port (clock : in std_logic;
          reset : in std_logic;
          data : in std_logic_vector(31 downto 0);
          count : in std_logic_vector(31 downto 0);
          ping_in : in std_logic;
          reg_enable : in std_logic;
          delay_enable : in std_logic;
          duration_enable : in std_logic;
          ping_done : out std_logic;
          ping_out : out std_logic);
  end component;
```

```vhdl
  component divider is
    port (in_clock : in std_logic;
          reset : in std_logic;
          out_clock : out std_logic);
  end component;

  component decoder is
    port (input : in std_logic_vector(5 downto 0);
          output : out std_logic_vector(39 downto 0));
  end component;

begin  -- structural

  C1 : divider port map (clock, reset, ping_clock);
  C2 : decoder port map (address, register_enables);
  CX : counter port map (clock, reset, counter_out);
  C3 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(0), register_enables(1),
completed_pings(0), pings(0));
  C4 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(2), register_enables(3),
completed_pings(1), pings(1));
  C5 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(4), register_enables(5),
completed_pings(2), pings(2));
  C6 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(6), register_enables(7),
completed_pings(3), pings(3));
  C7 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(8), register_enables(9),
completed_pings(4), pings(4));
  C8 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(10),
register_enables(11), completed_pings(5), pings(5));
  C9 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(12),
register_enables(13), completed_pings(6), pings(6));
  C10 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(14),
register_enables(15), completed_pings(7), pings(7));
  C11 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(16),
register_enables(17), completed_pings(8), pings(8));
  C12 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(18),
register_enables(19), completed_pings(9), pings(9));
  C13 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(20),
register_enables(21), completed_pings(10), pings(10));
  C14 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(22),
register_enables(23), completed_pings(11), pings(11));
  C15 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(24),
register_enables(25), completed_pings(12), pings(12));
```

```vhdl
  C16 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(26),
register_enables(27), completed_pings(13), pings(13));
  C17 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(28),
register_enables(29), completed_pings(14), pings(14));
  C18 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(30),
register_enables(31), completed_pings(15), pings(15));
  C19 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(32),
register_enables(33), completed_pings(16), pings(16));
  C20 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(34),
register_enables(35), completed_pings(17), pings(17));
  C21 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(36),
register_enables(37), completed_pings(18), pings(18));
  C22 : driver port map (clock, reset, data, counter_out,
ping_clock, enable, register_enables(38),
register_enables(39), completed_pings(19), pings(19));

  pings_completed <= (completed_pings(0) and
completed_pings(1) and completed_pings(2) and
completed_pings(3) and completed_pings(4) and
completed_pings(5) and completed_pings(6) and
completed_pings(7) and completed_pings(8) and
completed_pings(9) and completed_pings(10) and
completed_pings(11) and completed_pings(12) and
completed_pings(13) and completed_pings(14) and
completed_pings(15) and completed_pings(16) and
completed_pings(17) and completed_pings(18) and
completed_pings(19));

end structural;
```

counter_en

```vhdl
-- counter_en.vhd
-- a 32-bit counter with reset and enable
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter_en is
  port (clk : in std_logic;
        reset : in std_logic;
        enable : in std_logic;
        count : out std_logic_vector(31 downto 0));
end counter_en;

architecture behavioral of counter_en is
  signal count_internal : std_logic_vector(31 downto 0) :=
x"00000000";
begin
  process (clk, reset, enable)
```

```
  begin
    if reset = '1' then
      count_internal <= x"00000000";
    elsif clk'event and clk ='1' and enable = '1' then
      if count_internal = x"FFFFFFFF" then
        count_internal <= count_internal + x"00000000";
      else
        count_internal <= count_internal + x"00000001";
      end if;
    end if;
  end process;
  count <= count_internal;
end behavioral;
```

receiver

```
-- receiver.vhd
-- the receiver logic, which keeps track of TOF
library ieee;
use ieee.std_logic_1164.all;

entity receiver is
  port (ping1 : in std_logic;
        ping2 : in std_logic;
        ping3 : in std_logic;
        ping4 : in std_logic;
        clock : in std_logic;
        reset : in std_logic;
        found : out std_logic;
        count : out std_logic_vector(31 downto 0));
end receiver;

architecture structural of receiver is

  signal a : std_logic := '0';
  signal b : std_logic := '0';
  signal c : std_logic := '0';
  signal d : std_logic := '0';

  component counter_en is
  port (clk : in std_logic;
        reset : in std_logic;
        enable : in std_logic;
        count : out std_logic_vector(31 downto 0));
  end component;

  component flop is
  port (input : in std_logic;
        clk : in std_logic;
        reset : in std_logic;
        output : out std_logic);
  end component;

begin
  b <= ((not reset) and (clock) and (not a));
  c <= (ping1 or ping2 or ping3 or ping4);
```

```
  d <= not a;
  C1 : flop port map (c, b, reset, a);
  C2 : counter_en port map (clock, reset, d, count);
  found <= a;
end structural;
```
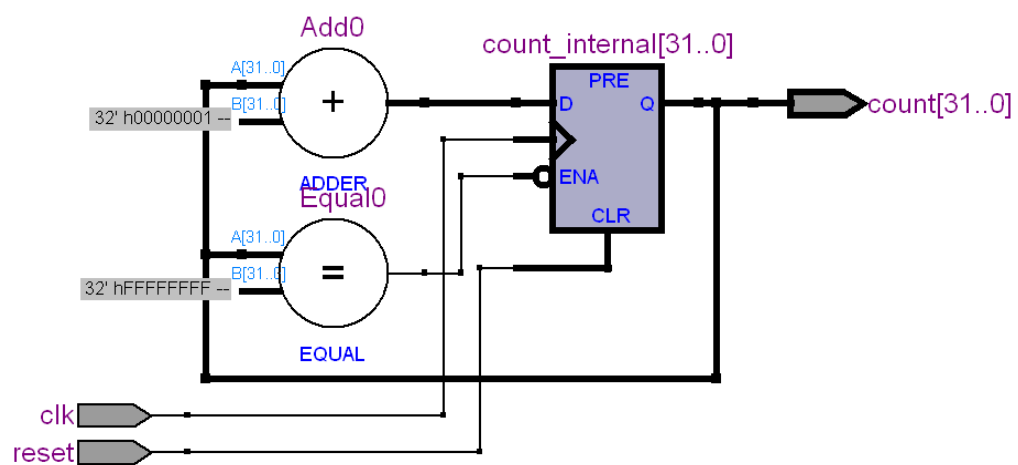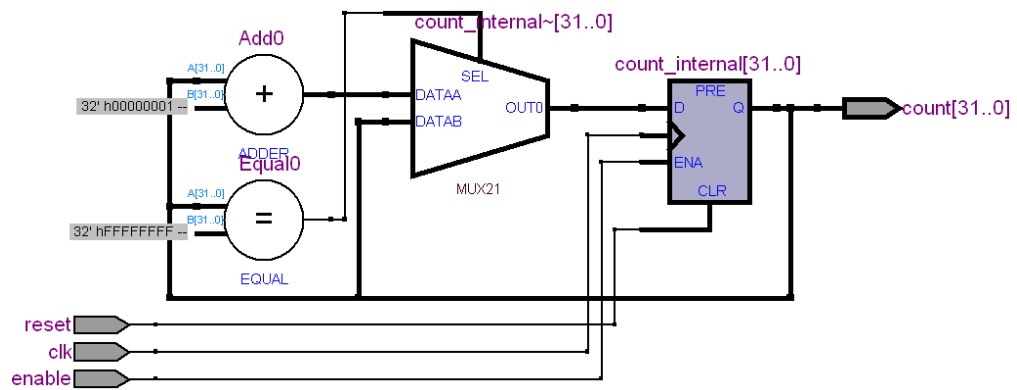
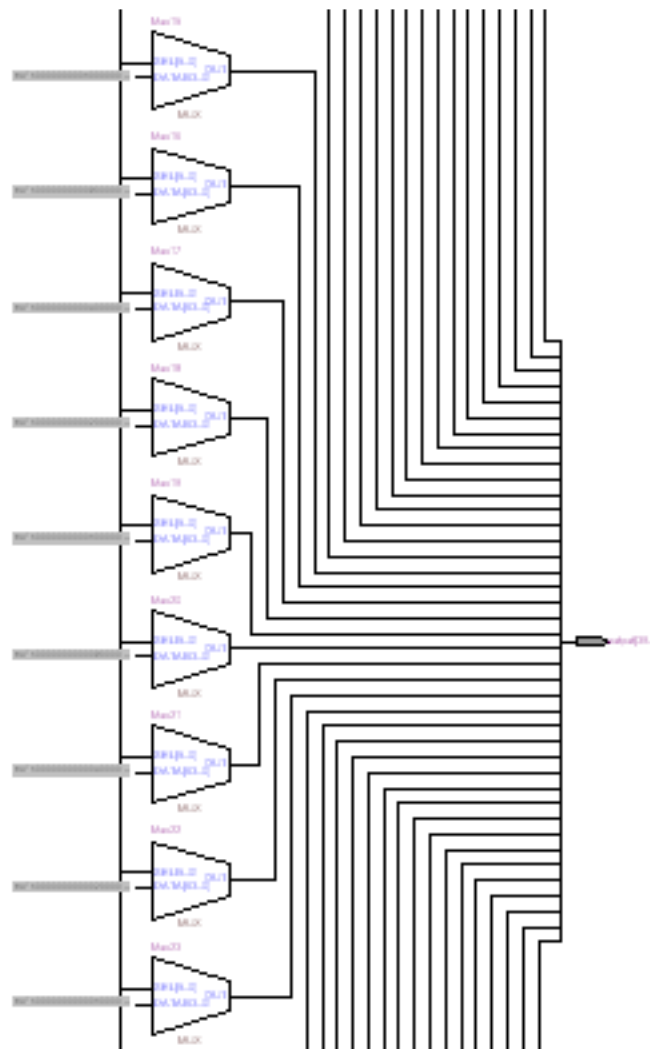# Appendix B: Synthesized RTL schematics
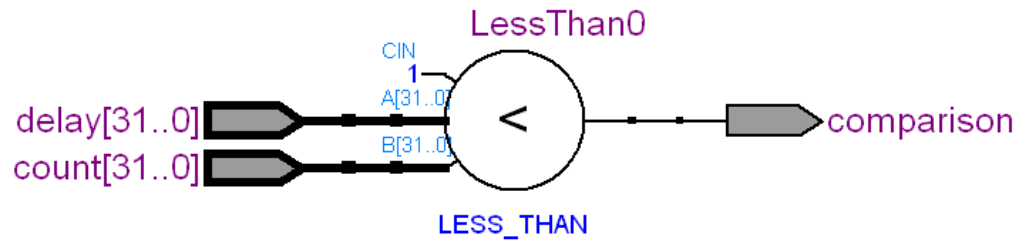
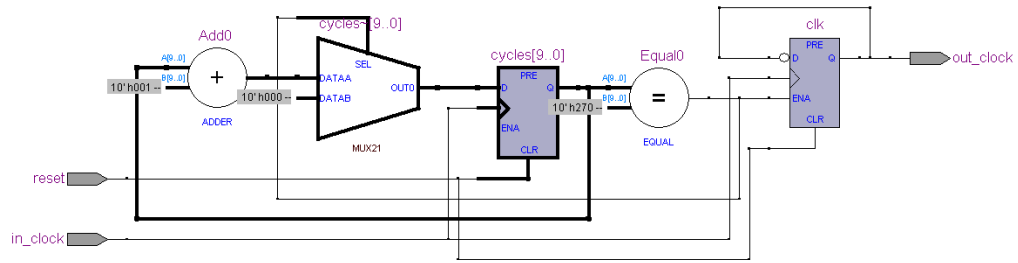comparator1 RTL schematic
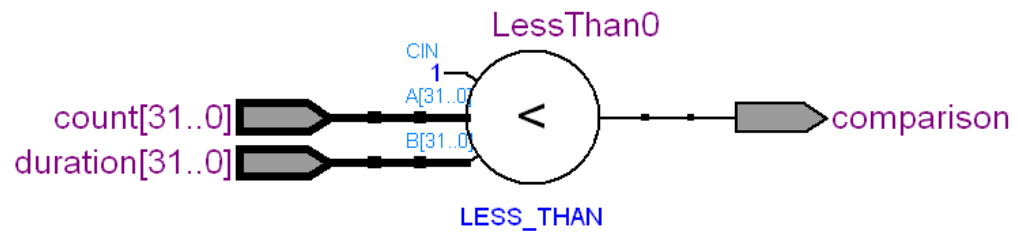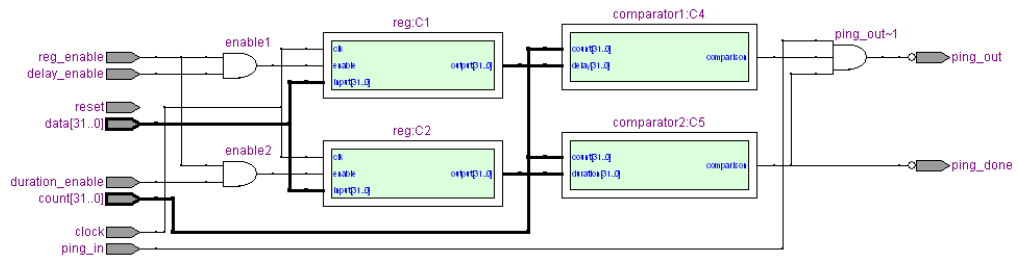

comparator2 RTL schematic

timer RTL schematic
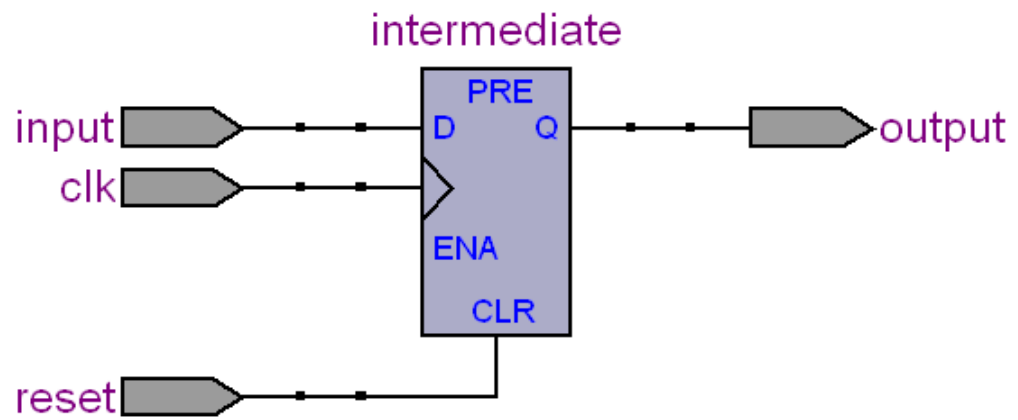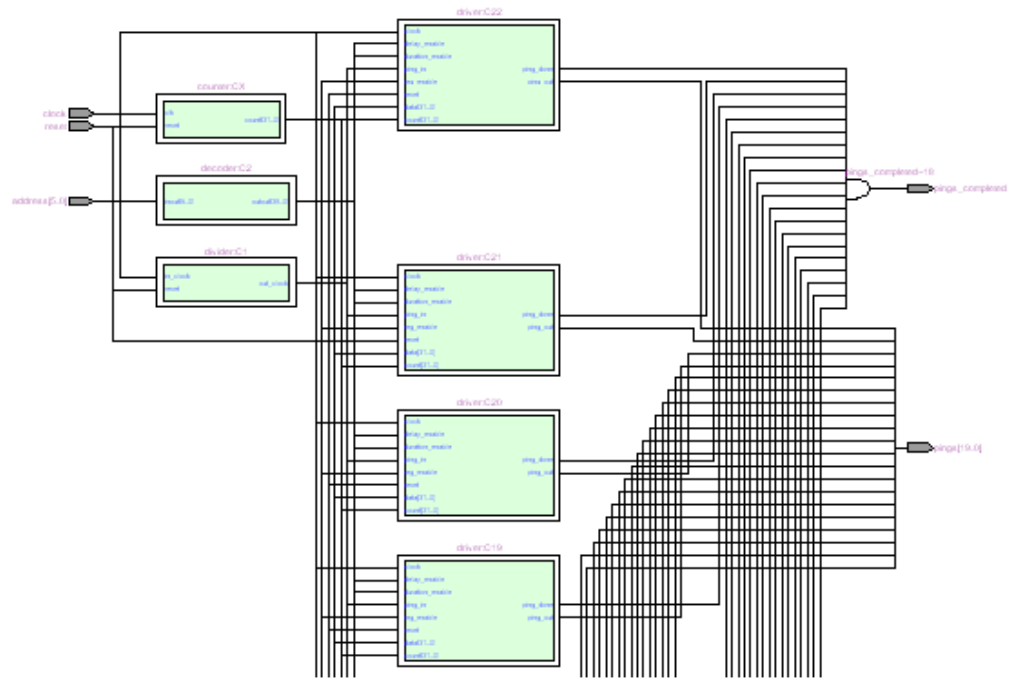
counter_en RTL schematic

decoder partial RTL schematic
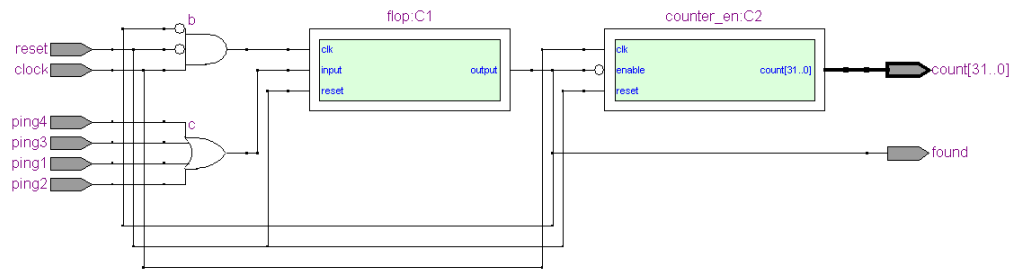
divider RTL schematic
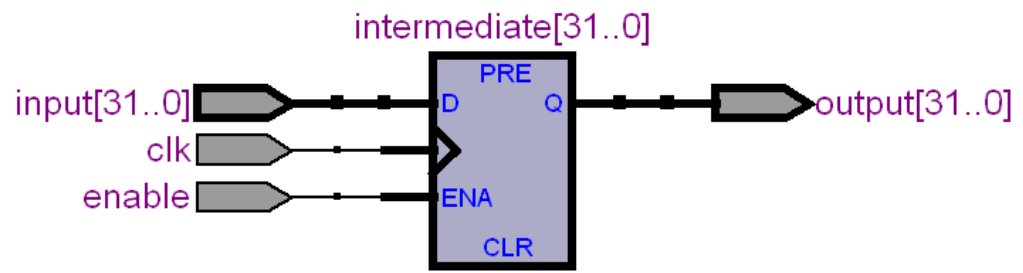

driver RTL schematic

flip flop RTL schematic

pinger partial RTL schematic

receiver RTL schematic

register RTL schematic

Written Capstone Summary

My Capstone project is based on a device called the Ultrasonic 3D Camera, a prototype of which was created to satisfy the requirements of the group project for CSE 497, the computer engineering senior design class. My Capstone project is an extension of the Ultrasonic 3D Camera, so please permit me to first describe the basic concept behind the Ultrasonic 3D Camera.

The Ultrasonic 3D Camera is a device that uses ultrasound, which is simply sound at a frequency higher than humans can hear, to create a three-dimensional model of a surface or object, through the air, from some distance. This is a significant accomplishment, as most ultrasonic technologies do not work through the air – they are generally used through the body or through water, both of which propagate sound much better than air.

To produce and detect ultrasound, the Camera makes use of ultrasonic transducers, which come in two varieties, transmitting and receiving. Transmitting ultrasonic transducers can be considered a type of special speaker that, when turned on, only plays one sound, and at a fixed frequency. Receiving ultrasonic transducers can be considered a type of special microphone that is tuned to only pick up the specific sound emitted by one of the transmitting transducers.

The way ultrasonic transducers collect data is really quite simple to understand – the transmitting transducers produce a directed beam of sound, which leaves the array, hits an object or surface, and then returns to the array, where it is detected by the receiving transducers. By timing how long it takes for sound to leave and return, a calculation can be performed (using the already known speed of sound through air) to see how far away the object that reflected the sound is.

Besides a large collection of ultrasonic transducers, the Ultrasonic 3D Camera has several other parts. There is an analog driving network, which is a collection of circuitry that assures that the sound being transmitted by the transducers is loud and focused. There is an analog receiving network, which is a collection of circuitry devoted to taking the weak signal that the array could potentially pick up and turning it into something usable. There is a digital driving network, which controls the order in which the driving transducers are used, and there is a digital receiving network, which detects exactly when a ping has returned to the camera. Both of the digital networks are described by VHDL code and implemented on an FPGA, a device that allows a user to write software, and then use the software to program the FPGA with hardware functionality. Finally, there is a software component, featuring the 3D modeling software OGRE. The Ultrasonic 3D Camera collects many points of data, but these points need to be connected into a mesh and displayed in a rendering environment – this is where OGRE comes into play.

There are many practical uses for a device like the Ultrasonic 3D Camera. The most obvious choice is on-the-fly modeling. Consider a video game artist who is required to make a realistic model of an object so that it can be used in the video game they are working on. Normally, to be assured that they have an accurate model, they would have to take many measurements of the object they were modeling, and then spend several hours faithfully recreating the object as a model. With the Ultrasonic 3D Camera, an artist could take a quick "snapshot" of the object they are charged with modeling, and then spend their time refining the model generated by the Camera, instead of making the entire model from scratch.

Another exciting application of the Ultrasonic 3D Camera is navigation for robots. Presently, there are many techniques for enabling robots to traverse unknown terrains, but none of them provide the robot with an actual distance map of its surroundings. Integrating a functional Ultrasonic 3D Camera with a robot would have a significant positive impact on the nature of robotic navigation, allowing new algorithms to be written to enable the robot to master its surroundings.

Now that a sufficient understanding of the Ultrasonic 3D Camera has been given, I'd like to describe exactly what I did for my Capstone project. First, I performed synthesis optimizations on the VHDL code written to describe the digital components of the system. VHDL code can be used multiple ways – for the group project, it was simply used for

rapid prototyping with the FPGA. However, VHDL code can also by synthesized and eventually used to produce an actual integrated circuit. For my Capstone I took the VHDL code that I alone wrote for the group project, and conditioned it for synthesis, by removing ambiguities and unnecessary logic.

Second, I selected a mesh algorithm to be used with the array. The point cloud collected by the array needs to be turned into a mesh (a collection of points with triangles whose vertices are the points) so that, when rendered, it looks like an actual object or surface and not simply a floating collection of points. For my Capstone project, I experimented with several mesh algorithms in an attempt to find one that fit well with the Ultrasonic 3D Camera.

The results of my Capstone project were quite good. I successfully minimized all of the digital components of the system, and I determined what I call the "naïve method" of mesh generation to be the best method for the Ultrasonic 3D Camera.