TUTORIAL ON DESIGNING AND SIMULATING A TRUNCATION SPURS-FREE DIRECT DIGITAL SYNTHESIZER (DDS) ON A FIELD-PROGRAMMABLE GATE ARRAY (FPGA)

BY

SHUO LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor José E. Schutt-Ainé

Abstract

Direct digital synthesis is a technique for using digital data processing blocks as a means to generate a frequency and phase tunable output signal referenced to a fixed-frequency precision clock source. Many telecommunication applications require such a high-speed switching, fine-tuning and superior quality signal source for their components. This thesis will introduce the direct digital synthesizer (DDS) and investigate the signal integrity issues associated with the DDS design.

In order to minimize the size of the lookup table to save hardware and lower the power consumption, we normally truncate the phase word output from the phase accumulator in the standard approach of designing DDS. However, this process will generate spurious frequencies (spurs), which degrade the quality of the output signals. It is considered one of the main signal integrity issues in the DDS design.

Previous research introduces a novel spurs-free truncation method for compressing the lookup table to avoid using phase truncation without significant hardware change. This thesis aims to implement this DDS with novel truncation spurs-free structure and test it in a practical environment. It does so by providing a tutorial on designing, implementing and simulating the DDS on an Altera DE2-115 FPGA using Altera Quartus II FPGA design software and ModelSim Simulator. The Verilog hardware description language is used as the development language. This thesis will describe entire designs of both DDS with traditional structure and DDS with novel truncation spurs-free structure. By comparing the outputs, it also examines the corresponding simulation results and verifies the improvement of the signal quality.

To my family, professors and friends for their love and support.

Acknowledgments

First of all, I would like to give my sincere gratitude to my graduate thesis advisor, Professor José E. Schutt-Ainé, for all his kind help, support, encouragement and guidance on my graduate study. He is a very kind and knowledgeable professor who is always willing to help the students when they are in need. I feel blessed that I have had the opportunity to learn from and work with my brilliant advisor and his research group in the past year.

Second, I would like to thank all my colleagues in Professor José E. Schutt-Ainé's research group. They are not only excellent researchers, but also great friends who are always very generous and helpful. The happy hours spent with them are really enjoyable. I would like to give my special thanks to my good friends in graduate school: Mujing Wang, Tao Yang, Xinying Wang, Jin Lei, Xu Chen, Xiao Ma, Albert Zaichen Chen, Mao Li, Linyang Zhang, Xinyang Song and Liang Zhang for all the inspirational talks and sweet memories we have together.

Thanks to Mr. Karan Bhagat and Mr. Yuanwang Yang who started this project and provided a detailed tutorial for me to carry on. Thanks to all the ECE staff, especially for Ms. Jen Carlson and Ms. Laurie Fisher who offered me great help during my graduate study at the University of Illinois at Urbana-Champaign.

Last but not least, I would like to give my sincere thanks to my parents, grandparents, girlfriend and sister for their continuous support, trust, understanding and love.

Contents

Chapter 1. INTRODUCTION

1.1 Motivation

A direct digital synthesizer (DDS) is a type of frequency synthesizer used for generating arbitrary waveforms referenced to a single, fixed-frequency clock. Nowadays, the cost-competitive, high-performance, functionality-integrated and small package-sized DDS products are widely used in the field of telecommunications and are becoming an alternative to some traditional analog synthesizer solutions. The applications of DDS include signal generation, local oscillators, function generators, mixers, modulators, and sound synthesizers.

The advantages of DDS are the following [1]:

1. Precise tuning resolution in micro-hertz and sub-degree phase tuning capability.

   Application example: Because the DDS is able to generate signals at very precise frequencies, it is useful for the applications that require phase continuous frequency sweeping such as filter characterization [2].

2. High speed tuning while keeping the phase continuous with no overshoots or undershoots.

   Application example: Function generators.

3. Digital architecture ensures no need for manual tuning or tweaking associated with component aging and temperature drift in an analog synthesizer solution.

   Application examples: Local oscillator for digital phase-locked loop (PLL)

4. Digital control interface enables the system to be remotely controlled, minutely optimized and under processor control.

With the above advantages over the traditional analog frequency synthesis technologies, it seems obvious that direct digital synthesis technology should be able to dominate the frequency synthesis area. However, the signal integrity problems mainly caused by the spurs have limited its usage in many high-demand applications. Among them, the truncation spurs generated from the phase truncation in the standard DDS design process are the primary problem and have become a major signal integrity issue in the DDS design.

By investigating and understanding the sources of the spurs, previous research has come up with a truncation spurs-free method for compressing the look-up table to avoid phase truncation without significant increase of hardware usage [3]. The motivation of this thesis is to design such a DDS with truncation spurs-free structure and verify the improvement of the signal quality. Since it is a preliminary research work, the DDS will be designed and implemented in Verilog codes that are synthesizable on an FPGA. FPGA is widely used in digital circuit design for its flexibility, accuracy and effectiveness. The most significant advantage of using an FPGA is that designs can be created and changed in a very short period. Instead, with application specific integrated circuits (ASIC), the designers will have to wait months for the circuits to be fabricated [4]. In this thesis, we aim to investigate the truncation spurs-free method; therefore, we will adjust the design and parameters along the way, so it is more feasible for us to use the FPGA design approach. However, on the other hand, the FPGA will limit the highest frequency we can reach. In this thesis work, the fixed reference clock frequency is 50MHz, which is the maximum FPGA on-board clock.

1.2 Outline

This thesis will serve as a complete tutorial on the background knowledge of DDS with traditional structure and DDS with truncation spurs-free structure as well as how to design and implement them in Verilog that are synthesizable on FPGA and simulate them with ModelSim.

1. Chapter 2 provides an overview of the structure and operation of the DDS with both traditional and truncation spurs-free structures. Additionally, it introduces which types of spurs that may exist, the reason why they exist and the method for eliminating the truncation spurs.

2. Chapter 3 serves as an introduction to Verilog HDL. It addresses the necessary knowledge that we should have about Verilog for completing this project by providing examples of some major blocks in digital circuit design.

3. Chapter 4 introduces the background knowledge of FPGA and the FPGA design flow

4. Chapter 5 presents a detailed and complete step-by-step tutorial on designing, implementing and simulating both DDS with traditional structure and DDS with truncation spurs-free structure on FPGA in Altera Quartus FPGA development software.

5. Chapter 6 presents the simulation results from the different structures of DDS and some theoretical analysis of the results.

6. Chapter 7 concludes this thesis with a summary and potential topics for further research.

Chapter 2. THEORETICAL ANALYSIS OF DDS

2.1 Theoretical Analysis of DDS with Traditional Structure

The traditional direct digital synthesizer (DDS) mainly consists of four primary components. The first one is the phase accumulator (PA), which determines the frequency range and accuracy of the output signal. The second component is the lookup table (LUT), which is used to store the amplitude information of the quantized and discrete sine wave. The third component is the digital to analog converter (DAC), which generates analog signal. The fourth component is the low pass filter (LPF), which is used to smoothen the output signal [5],[6]. The traditional structure of DDS is shown in Figure 2.1 [4].

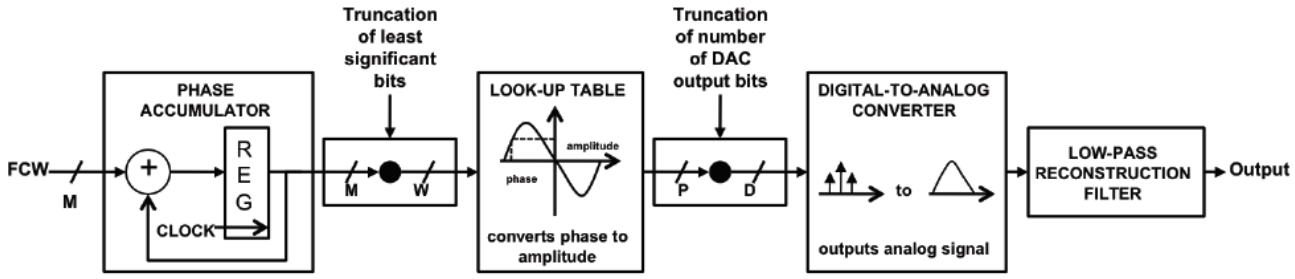

Figure 2.1: Traditional Structure of DDS

The operation of the DDS starts with applying a frequency control/tuning word (FCW/FTW) to the PA. With referencing a fixed input clock, the PA will increment by the M-bit FCW (M = 32 in our design) once in each clock cycle and the result value is stored in an inbuilt register. The output of the register will loop back to be accumulated with input FCW in the next clock cycle. The output of the PA is then truncated from M-bit to W-bit (W<M) and fed into the input of LUT. The process of truncation is simply elimination of the lower order bits. The LUT will take the W-bit word (W = 8 in our design) as the phase of the sine wave and generate the amplitude. Therefore, the LUT is also called a phase to amplitude converter. The quantized version of the sine wave is then fed into the DAC, which generates the analog output signal. Generally, the bit width of the DAC input is always limited. Therefore, the output of the LUT is truncated again before being fed into the DAC. After converting the digital signal to analog signal, the output of DAC is then fed into the LPF, which reduces the noise and eliminates the spikes of the signal [1],[4]. The main consideration is that PA and DAC should operate on the same reference clock. Even though the PA is clocked, it still operates very fast; however, the speed of DAC and LPF is relatively low due to their design architecture [7]. Now, we will look into the details in each component and understand the background theory on how exactly the DDS works.

Phase Accumulator (PA):

The PA is constructed by an adder and a built-in register. In each clock cycle, we can realize the accumulation by adding the output of the register from last clock cycle back to the input of the adder. Then, the M-bit output of PA will increment by FCW. The structure of the PA is shown in Figure 2.2.

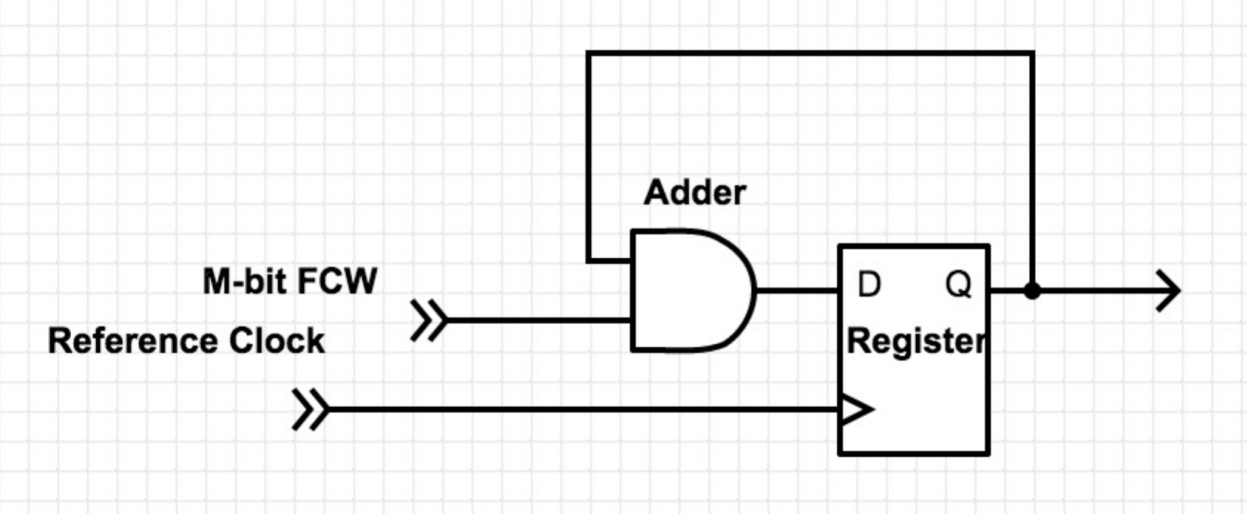

Figure 2.2: Structure of Phase Accumulator

The output of PA forms a quantized saw-tooth waveform as shown in Figure 2.3 [1],[4]. Each dot on the saw-tooth waveform is the actual value of PA output.

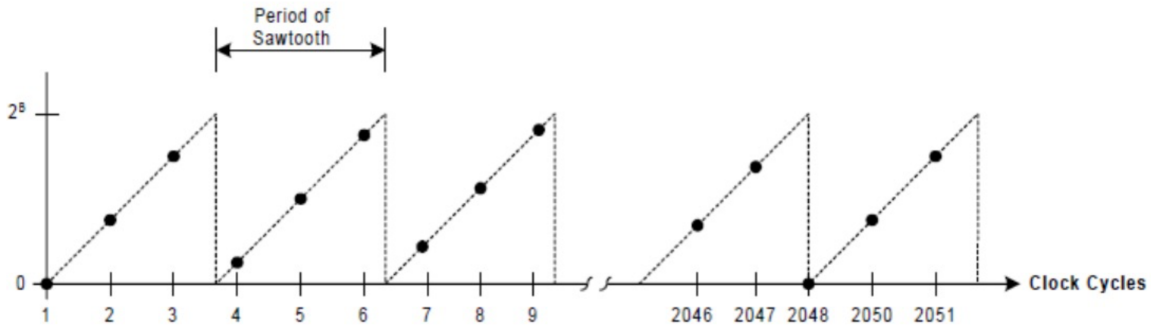

Figure 2.3: Behavior of Truncation Words

In Figure 2.2, we name the reference clock frequency of DDS system as f_{clock}. DDS conducts frequency accumulation from the "Phase" concept. As we mentioned above, the PA is constructed by an adder and a register. For each coming clock impulse, the adder will add FCW with the output of the PA from the last clock cycle, then send the sum back to the output of the PA to realize accumulation. In this way, PA will increment by FCW once at each coming clock impulse. The output data is the phase of the synthesized signal. The overflow frequency of the PA is the frequency of the DDS output signal.

PA is the core of DDS system that generates phase information of the signal increment. For the sine wave, instant amplitude completely depends on instant phase according to Equation 2.1

$$\omega = \frac{d\phi(t)}{dt} \tag{2.1}$$

Therefore, the faster the phase change, the higher the signal frequency.

The PA applies the overflow feature of M-bit binary adder to simulate 2π phase cycle of the ideal sine wave. The output of the PA can be considered as the phase signal of the ideal sine wave while the output of LUT can be considered as the clock sampling of the time-domain waveform.

Let M be the word length of the PA and f_{clock} as the reference clock frequency; then the clock cycle is shown as Equation 2.2.

$$T_c = \frac{1}{f_{clock}} \tag{2.2}$$

Then, the PA has 2^M possible values. FCW is the frequency control word. During the working process of the system, the increment of the PA in each clock cycle is

$$\Delta\phi = FCW \times \frac{2\pi}{2^M} \tag{2.3}$$

The corresponding angular frequency would be

$$\omega = \frac{\Delta\Phi}{\Delta t} = \frac{\Delta\Phi}{T_C} = FCW \times \frac{2\pi f_{clock}}{2^M} \tag{2.4}$$

Therefore, the output frequency of the DDS is

$$f_{DDS} = \frac{\omega}{2\pi} = FCW \times \frac{f_{clock}}{2^M} \tag{2.5}$$

From Equation 2.5, we know that the larger the FCW, the faster the PA jumps, which leads to a higher frequency at the output. The resolution of the DDS output signal or the step interval of the f_{DDS} is

$$\Delta f_{DDS} = \frac{f_{clock}}{2^M} \tag{2.6}$$

Since the output signal of the DDS is the sampling synthesis of the sine wave, it is very important to fulfill the Nyquist theorem requirement. The Nyquist theorem states: "If a function x(t) contains no frequencies higher than B Hz, it is completely determined by giving its ordinates at a series of points spaced 1/(2B) seconds apart," [4],[8],[9]. Equation 2.5 is conditional, given that Equation 2.7 is true [9].

$$f_{DDS} = \frac{f_{clock}}{2} \tag{2.7}$$

Thus,

$$FCW \leq 2^{M-1} \tag{2.8}$$

According to the characteristic requirement of the spectrum, we normally choose [1]

$$f_{DDS} \leq 0.4 f_{clock} \tag{2.9}$$

For a more straightforward explanation, the function of PA performs as a “phase wheel.” Shown as Figure 2.4 , the sine wave oscillation is considered as a vector rotation around a phase circle. Each designated point on the phase wheel corresponds to an equivalent point on a full cycle of a sine wave. As the vector rotates, the corresponding output sine wave is being generated. When the vector finishes rotating the whole phase wheel at a constant speed, it means that a complete cycle of sine wave is outputted. The contents of the PA correspond to the points on the cycle of the sine wave. The number of discrete phase points on the phase wheel depends on the resolution of the PA, which is $f_{clock}/2^M$ in this thesis design. The larger the M, the larger number of discrete phase points we have on the phase wheel (the number is 2^M). However, the output of the PA is linear, so it cannot be used to generate a sine wave directly; it is the reason that the DDS system includes a LUT [1].

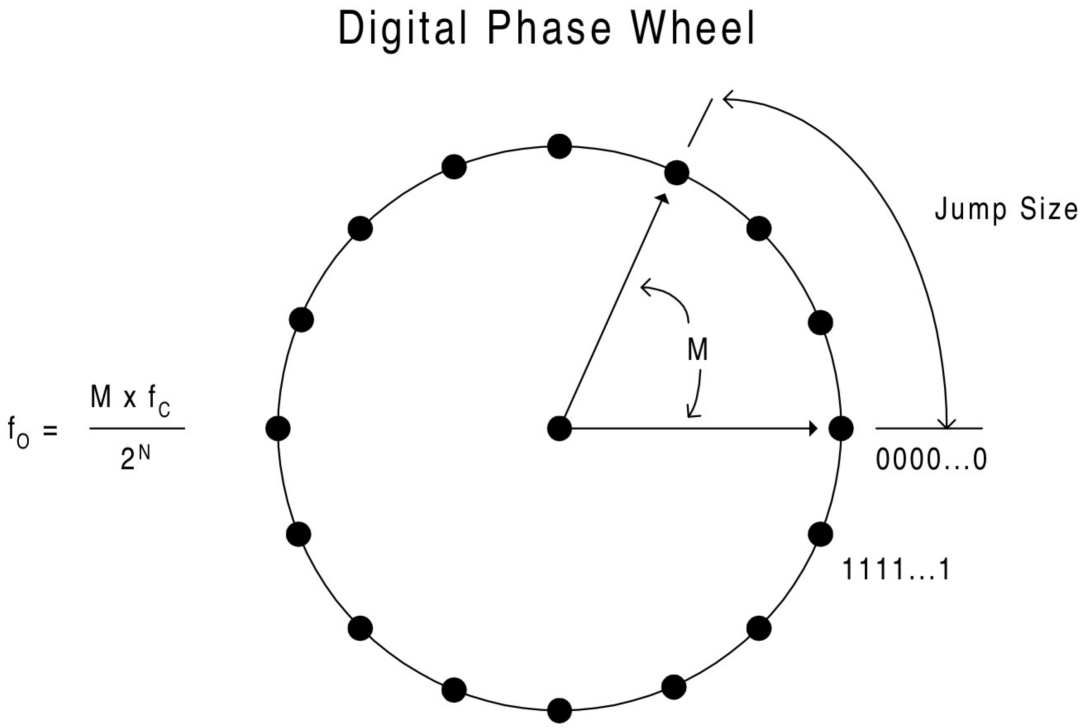

Figure 2.4: Digital Phase Wheel

Lookup Table (LUT):

The lookup table (LUT) serves as a phase-to-amplitude converter, which is used to convert a truncated version of the PA's instantaneous output value into the discrete sine wave amplitude information that is presented to the D/A converter [1]. It consists of a read only memory (ROM). By using the output data of the PA as the phase sampling address, we can get waveform sampling data (binary code) stored in the LUT, then complete the phase-to-amplitude conversion. To keep the LUT reasonably sized, we truncate the bits from the PA and only feed the higher order bits to the input of LUT to save the hardware resources and power.

The LUT contains unique values of a sine wave over one period; however, most DDS architectures will exploit the symmetrical nature of a sine wave and utilize mapping logic to synthesize a complete sine wave cycle from a quarter cycle of data from the PA. The LUT will generate the all the necessary data by reading forward and backward through the LUT [1],[10].

In the FPGA design, one will need a Memory Initialization File [.mif] containing the values of the LUT. We use MATLAB to generate this file with the file extension as ".mif". This file will be added to the ROM [4]. For details, please refer to **Section 5.2-6**.

Digital to Analog Converter (DAC)

The digital to analog converter (DAC) is applied to create an analog waveform from the digital discretized sine wave. Since the bit width of the DAC is generally limited, we will need to apply a second truncation process to the output of the LUT to get a word with appropriate number of bits and then feed it to the DAC input. An important fact is that the DAC plays a big role for limiting the design's maximum attainable frequency because the PA and DAC need to work with same reference clock. In this thesis, we design and implement the DDS completely in HDL, and perform the behavioral and post map & route simulations with ModelSim, so we don't need an actual DAC. Also, DAC will bring other sources of spurs, which may influence the results. Further potential research directions in this topic may need to have an actual DAC or even involve designing DAC. Please refer to **Section 7.2** for details.

Low Pass Filter (LPF):

In the DDS design, the LPF performs as a reconstruction filter, which reduces the noise and eliminates the spikes of the input signal that come from the DAC. Since we do not want any aliases of the fundamental frequency, the LPF also performs as an antialiasing filter; therefore, it limits us to the Nyquist frequency. Due to the sharp frequency response characteristics, a Chebyshev filter is typically used on this stage [1]. Same with the DAC, we don't need an actual LPF in this thesis work.

2.2 Types of Spurs in DDS

The direct digital synthesis (DDS) technology has several advantages over the traditional analog frequency synthesis technologies in terms of high-frequency resolution, high-speed frequency tuning, continuous phase and so on. However, the signal quality issues primarily caused by the spurs limit the usage of DDS technology in many high-demand applications. The three major sources of spurs are shown as Figure 2.5:

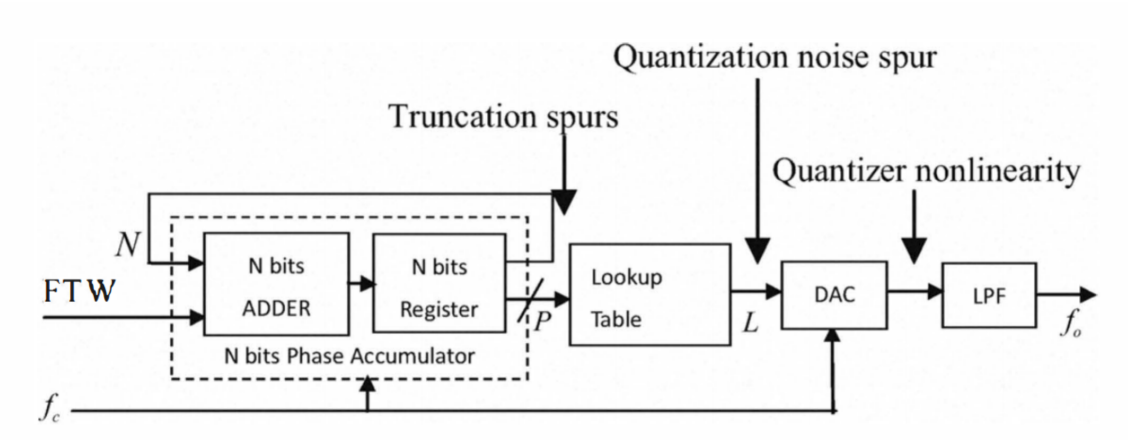

Figure 2.5: Three Major Sources of Spurs

Phase Truncation Spurs:

In order to obtain high-frequency resolution at the DDS output, the bit width of the phase accumulator must be sufficiently wide, typically 24-48 bits [1],[3]. However, in order to design a smaller sized lookup table (LUT) that only needs a reasonable amount of hardware and consumes less power, we eliminate some of the least significant bits (LSBs) of the 32-bit word from the phase accumulator output because it is relatively easy to reduce the size of the LUT and truncate the phase word at its input [1]. Unfortunately, this truncation of bits will lead to the spectral impurity to the output signal known as the phase truncation spurs and it is the biggest source of noise and spikes in the DDS system. We will provide detailed further explanation in **Section 2.3**. Many algorithms can be implemented and added into this digital design to reduce the phase truncation spurs. In this thesis project, we will design and implement a DDS with novel truncation spurs-free structure on FPGA to eliminate the truncation spurs.

Quantization Noise Spurs:

The bit width of a digital to analog converter (DAC) is always limited. Generally, the bit width of the DAC is even narrower than that of the LUT [1]. In this project, we truncate the output of the LUT even further and then give it to the DAC. The discrete amplitude values of the sine signal are quantized and stored in the LUT. The DAC will accept signed binary number with a certain precision. To achieve this, the input bits are further rounded. This quantization will generate spurs at the output frequency spectrum of the DDS.

Quantization Nonlinearity Spurs:

Due to the DAC's inherent design and non-ideal transfer function behavior, it is impossible to design a perfect DAC. Every input will have few errors associated with it, so we cannot get an ideal output. These errors, caused by nonlinear behavior of the DAC, lead to the quantization nonlinearity spurs. This type of spur can further exacerbate the truncation and quantization spurs noise spurs and is very hard to evaluate [11]. The spurs caused by the nonlinear behavior of DAC can only be reduced by increasing the precision of the DAC; however, the elimination will improve the quality of output signal significantly [3].

2.3 Theoretical Analysis of DDS with a Novel Truncation Spurs-Free Structure

In the previous sections, we mentioned that the truncation spurs caused by truncating the lower order bits of PA output are the primary signal integrity issue of DDS system, which limits the usage of DDS in many high-demand applications. In this section, we will introduce a truncation spurs-free method to compress the LUT size without phase truncation and a significant change of hardware usage [3].

First of all, we need to understand exactly what happens behind the phase truncation that leads to the truncation spurs. In Figure 2.6, the red line represents the output of the PA; the green line represents the output of the truncator; the y-axis represents the phase address in LUT and the x-axis represents the time. The input of the LUT is actually the output of the truncator, which serves as the phase address. In the truncating process, the irregular sampling of the LUT sine wave may occur as the red arrow pointed out in Figure 2.6.

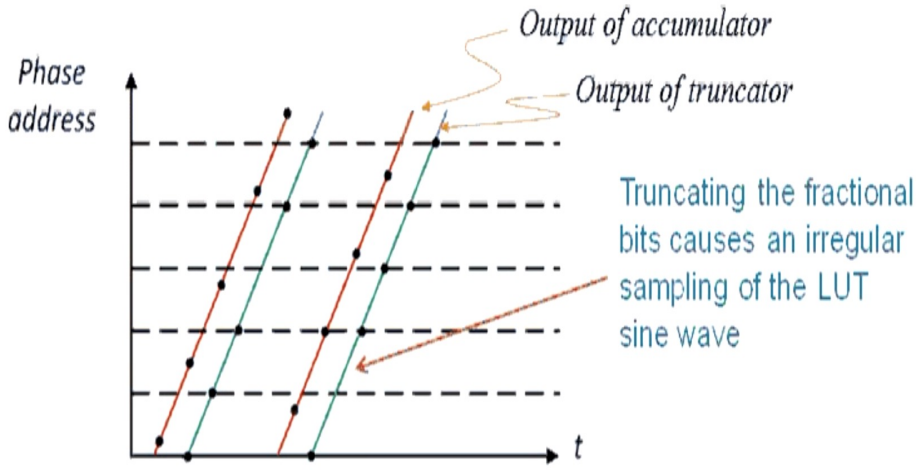

Figure 2.6: Effects on Phase Truncation

In Figure 2.5, N is the bit-width of PA input and P is the bit-width of the LUT input. We know that if N = P, then there is no phase truncation. The truncation spurs existed because N-P bits are discarded. We call these N-P bits fractional bits. As we mentioned in Section 2.1, there is a fact that the bit-width of DAC input is always limited, generally much smaller than that of PA. Thus, we will have to perform truncation on the LUT output before feeding it to DAC anyway.

L is the bit-width of DAC input (L<<N); thus there are only 2^L possible data values at the DAC input. Consequently, there are only 2^L corresponding phase points referred to as the key phase points stored in LUT. These key phase points will divide one complete cycle of sine wave into 2^L-1 key spans. If we can find which key span corresponds to which PA outputs, an accurate amplitude value stored in LUT can be determined and sent to the DAC. In this way, we can eliminate the truncation spurs. Now, the question is how can we actually determine which key span corresponds to which PA outputs. To solve this problem, we come up with a truncation spurs-free structure of DDS by introducing a comparator and an adder to the traditional structure of DDS. The truncation spurs-free structure of DDS is shown in Figure 2.7.

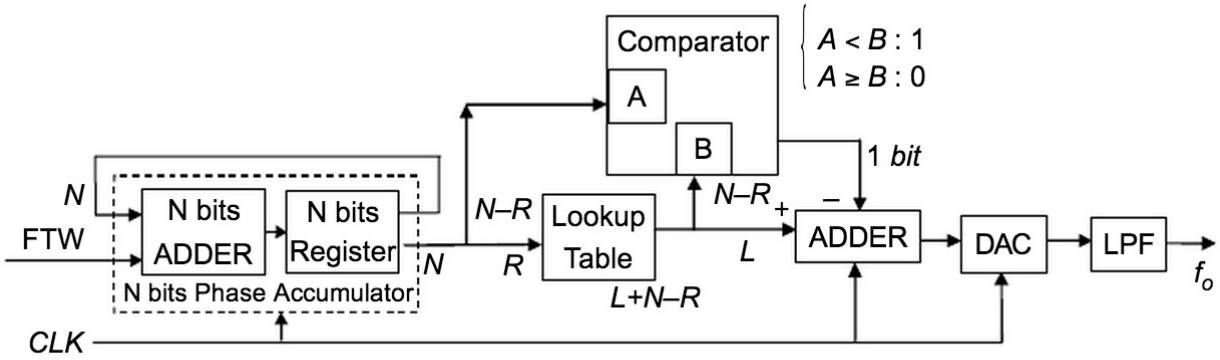

Figure 2.7: Truncation Spurs-Free Structure of DDS

As we mentioned above, there are 2^L key phase points stored in the LUT. The comparator will compare the outputs of PA to each key phase point and generate an accurate amplitude value in one clock cycle. The main difference is that the lower (N-R) bits from output of PA are not simply discarded. Instead, they will be compared with the lower (N-R) bits of the key phase points stored in the LUT. If the lower (N-R) bits of PA output correspond to a larger value, then the amplitude value will be sent to DAC directly. Otherwise, the amplitude value is sent after subtracting 1. The output of LUT is adjusted by an adder. The irregular sampling of the sine wave will not happen again as we shown in Figure 2.6.

The size of the new LUT would be $(2^R \times (L+N-R))$ instead of $2^R \times L$. We can see that the size of ROM increases linearly instead of exponentially.

Chapter 3. INTRODUCTION TO VERILOG

Verilog is a hardware description language (HDL) standardized as IEEE 1364-1995. It can be used to model digital systems at algorithm level, gate level and transistor level. It is most commonly used in the design and verification of digital circuits at the register-transistor level (RTL) of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits.

Verilog is very simple, straightforward and efficient. Since Verilog is not only a machine-readable language, but also a human readable language, it can support hardware design verification, synthesis, and testing. Nowadays, Verilog has become the top choice in digital systems design and the foundations for synthesis, verification, and layout technology.

Verilog includes plenty of built-in primitives, including logic gates, user-defined primitives, transistors and line logic. It also has the function to check the timing related problems between device pins. Generally speaking, Verilog has two data types to support its mixed abstraction levels. These two types are net and variable. For continuous assignment, variable and net are able to assign the data to net continuously. Verilog provides a basic structural modeling method. For procedural assignment, the calculation results of net and variable can be stored in variable. Verilog provides a behavioral modeling method.

A project developed in Verilog consists of several modules. Each module will include an I/O and a function description. The function description of a module can be structural level, behavioral level or mixed level. Then we connect these modules together with nets. A complete Verilog design module includes four main parts: port definition, I/O statement, signal type statement, and function description.

Compared with another common hardware description language, VHDL (very high speed integrated circuit HDL), Verilog is a weakly typed HDL, is easier to learn and more concise with efficient notations. The syntax is more C-like. On the other hand, VHDL as a strong typed HDL is more verbose than Verilog. Consequently, designs in VHDL are considered self-documenting. Engineers working with VHDL need to do extra coding; however, they often catch errors missed by Verilog. Verilog is good at hardware modeling but lacks higher level constructs, while VHDL has many programming constructs but lacks the low level modeling capabilities. Although Verilog is more popular in industry today, VHDL is still being used in some top companies like National Instruments due to its features. In this thesis design, we will choose Verilog as the hardware design language.

3.1 Resources

1. From my own learning experience, I would say that the "World of ASIC" website (www.asic-world.com) is one of the best resources to learn digital design in Verilog [12]. It provides detailed tutorials, design examples, and suggestions on tools and reference books for Verilog beginner. It also provides instructions for other HDLs as well as a scripting language used in digital design and verification such as VHDL, SystemVerilog, SystemC, and Perl.

2. Besides the references books recommended on the "World of ASIC" website, two junior-level undergraduate courses at UIUC are good resources on learning digital design, especially the skills that you will use in this tutorial [13],[14]. One is ECE 385 Digital Systems Laboratory, which gives you hands-on experiences on designing complex digital systems from scratch. You will learn how to implement circuits on a breadboard, design digital systems in HDL and synthesize your circuits on an FPGA for testing and verification. Another is CS 233 Computer Architecture, which teaches fundamentals on computer architecture in a practical approach by providing excellent machine problems. You will need to use Verilog and C++ to finish these assignments.

3.2 Verilog Design Examples

Some design examples that will be necessary to use when designing the DDS are given below:
Note: These are not actual codes for this project. For major modules, please check **Appendix B**.

First of all, we will need to define a module in Verilog, which includes defining a module name, ports as well as vector ports and ports directions. The design example is shown in Figure 3.1.

```
1  module dds(
2      clk,
3      reset,
4      FCW,
5      dataout);
6
7  input clk;
8  input reset;
9  input [31:0] FCW;
10 output  data2dac;
11
12 //Add entire design and instantiations.
13
14 endmodule
```

Figure 3.1: Example for Defining a Module in Verilog

Hardware has two kinds of drivers (data type which can drive a load). The first one is called reg in Verilog while the other is called wire. The data type example is shown in Figure 3.2.

```
reg register_name; //a single bit register
wire wire_name; //a single wire
reg [31:0] register_name; //a 32-bit register
wire [31:0] wire_name; //a 32-bit wide bus
```

Figure 3.2: Example for Data Type Assignment

For the operators, they are almost the same thing in the other programming languages such as C programming language. Now, we will introduce some useful control statements and variable assignments by providing design examples for some required sub-modules in DDS design.

Register:

```
module register (
in,
clk,
reset,
out
);

input in, clk, reset;
output out;

reg out; //Internal variables

always @ (posedge clk)
    if (~reset) begin
        out <= 1'b0;
        end
    else begin
        out <= in;
        end

endmodule
```

Figure 3.3: Design Example of a Register

In the Figure 3.3, the design example of the register includes an if-else statement, which is to check a condition to decide whether or not to execute a portion of code. If the condition is satisfied, the code is executed. If not, it runs the other portion of code. In the above example, it checks if reset is 1. If this condition is satisfied, then it will output 0; if not, it will output the input.

Counter:

```
module counter (
clk,
reset,
enable,
count
);

input clk, reset, enable;
output [3:0] count;

reg [3:0] count;

always @ (posedge clk or posedge reset)
    if (reset) begin
    count <= 0;
    end
    else begin
        while (enable) begin
            count <= count + 1;
        end
    end

endmodule
```

Figure 3.4: Design Example of a Counter

In the Figure 3.4, the design example of the counter includes a while statement, which executes the code within it repeatedly if the condition returns true. In the above example, the count keeps increment by 1 if the enable is checked to be true.

n-bit Full Adder:

```
module n_adder (
    in_a,
    in_b,
    cin,
    cout,
    sum
);

parameter size = n;

input [size-1:0] in_a;
input [size-1:0] in_b;
input cin;
output [size-1:0] sum;
output cout;

assign {cout,sum} = in_a + in_b + cin;

endmodule
```

Figure 3.5: Design example of n-bit full adder

In digital design, there are two types of elements, combinational and sequential. In Verilog, there are two approaches to model combinational elements, one is using an "assign" statement; the other one is using an "always" statement. However, there is only one way to model sequential elements, which is using "always" statement. In the Figure 3.5, we use "assign" statement to output cout and sum because it is a combinational logic. Besides "assign" and "always", there is a third statement called initial statement, which is only used in designing test benches in Verilog. It is executed at the beginning of simulation. An example is shown in Figure 3.6

```
initial begin
    clk = 0;
    enable = 1;
    FCW = 1048576; // 2^20 = 1048576

end
```

Figure 3.6: Design Example of Initial Statement

Comparator:

```
module comparator (
    a,
    b,
    G,
    E,
    L
);
input [7:0] a, b;
output G, E, L;

wire G, E, L;

    assign G = (a>b);
    assign E = (a==b);
    assign L = (a<b);

endmodule
```

Figure 3.7: Design Example of an 8-bit Comparator

```
module comparator (
    a,
    b,
    G,
    E,
    L
);

input a0,a1,b0,b1;
output G,E,L;

reg G,E,L;

always@(a0 or a1 or b0 or b1)
    begin
     G <= {a1,a0}>{b1,b0};
     E <= {a1,a0}=={b1,b0};
     L <= {a1,a0}<{b1,b0};
    end

endmodule
```

Figure 3.8: Design Example of a 2-bit Comparator

In Figure 3.7 and Figure 3.8, we use two different approaches to design the comparator. One uses an “assign” statement; the other one uses an “always” statement. In this thesis work, we will use an “assign” statement to design the comparator because it is relatively simpler.

Finite State Machine (FSM):

```
module FSM (
    clock,
    reset,
    sig_1,
    sig_2,
    a,
    b
);

input clock, reset;
output a, b;

wire clock, reset, sig_1, sig_2;
reg a, b;

parameter size = 2;
parameter IDLE = 2'b00, A = 2'b01, B = 2'b10;

reg [size-1:0] state;
reg [size-1:0] next_state;
```

```
//combinational logic
always @ (clock or sig_1 or sig_2)
    begin : FSM_COMBO
        next_state = 2'b00;
        case(state)
        IDLE : if (sig_1 == 1'b1) begin
                    next_state = A;
                    end else if (sig_2 == 1'b1) begin
                    next_state = B;
                    end else begin
                    next_state = IDLE;
                    end

        A : if (sig_2 == 1'b1) begin
                next_state = B;
                end else begin
                next_state = A;
                end

        B:  next_state = IDLE;

        default : next_state = IDLE;
        endcase
    end
```

Figure 3.9: Design Example of the FSM

```
45  //sequential logic
46  always @ (posedge clock)
47      begin : FSM_SEQ
48          if (reset == 1'b1) begin
49          state <= IDLE;
50          end else begin
51          state <= next_state;
52          end
53      end

54  //output logic
55  always @ (posedge clock)
56      begin : FSM_OUTPUT
57          if (reset == 1'b1) begin
58              a <= 1'b0;
59              b <= 1'b0;
60              end else begin
61              case(state)
62              IDLE : begin
63                      a <= 1'b0;
64                      b <= 1'b0;
65                      end
66
67              A : begin
68                  a <= 1'b0;
69                  b <= 1'b1;
70                  end
71
72              B : begin
73                  a <= 1'b1;
74                  b <= 1'b1;
75                  end

77               default : begin
78                           a <= 1'b0;
79                           b <= 1'b0;
80                           end
81              endcase
82              end
83          end
84  endmodule
```

Figure 3.9: Continued

In the Figure 3.9, the design example of the FSM includes the case statements, which are used when we have one variable that needs to be checked for multiple values. In the above example, we use case statement to check the state status. It is important to note that that it is better to include a default case with a return to idle every time we use case statement to make the code safe because if the Verilog machine enters to a non-covered state, it will hang there. Also, when you use the case statement, if you don't cover all the cases and you are trying to write a combinational logic, the synthesis tool will infer latch.

From all the examples and descriptions above, you may see the "always" statement several times. In the following, I will introduce this important block in Verilog design.

From the name we can imagine that the "always" statement executes always instead of executing once like the initial statement. It includes a sensitive list or a delay associated with it. There are two types of sensitive lists, one is level sensitive for combinational logic; one is edge sensitive for flip-flops. In the FSM example, "sig_1" and "sig_2" included in "always" in the combinational logic block are level sensitive lists. The "clock" included in "always" in the sequential logic block is an edge sensitive list. If a change happens in any of the sensitive lists, the always statement will be triggered.

We can have an "always" statement without a sensitive list in the case where we have a delay, as shown in Figure 3.10.

```
21  always begin
22      #1 clk = ~clk
23      end
```

Figure 3.10: Example of "Always" Statement Without a Sensitive List

Finally, if you look into the codes above, you will see two different types of assignment operators. One is "=", which is used in the combinational logic; the other one is "<=", which is used in the sequential logic. "=" is called blocking assignment and "<=" is called non-blocking assignment. The blocking assignment executes codes sequentially while the non-blocking one executes codes in parallel. This is very important because misuse of these two assignments will totally disrupt the codes. Also, begin and end constructs are only necessary when multiple operations.

Hardware coding is different from software coding. Sometimes you should think like real hardware and solve issues associated with real hardware to make a good design.

Chapter 4. FPGA DESIGN FLOW

4.1 Introduction to FPGA

FPGA is an integrated circuit designed to be configured by a customer or a designer after manufacturing. The FPGA configuration is generally specified using HDL. From a research standpoint, we will use the FPGA instead of the ASIC approach to build the DDS in this thesis, because the designs on FPGA can be easily modified and tested on board.

As shown in Figure 4.1, the FPGA we choose is Altera DE2-115 development and education board, which is an ideal teaching platform [15].

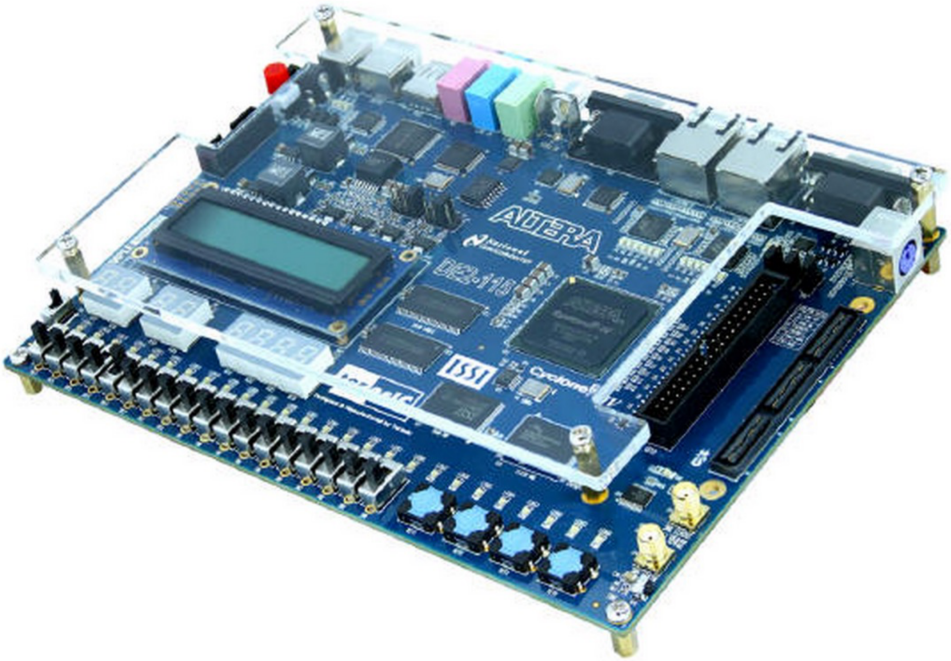

Figure 4.1: DE2-115 Development and Education Board

There are plenty of switches and ports, 128M on-board memory and enough hardware resources for us to use. The on board clock frequency is 50MHz.

4.2 Design Flow

The FPGA design flow is shown as Figure 4.2 [16],[17].

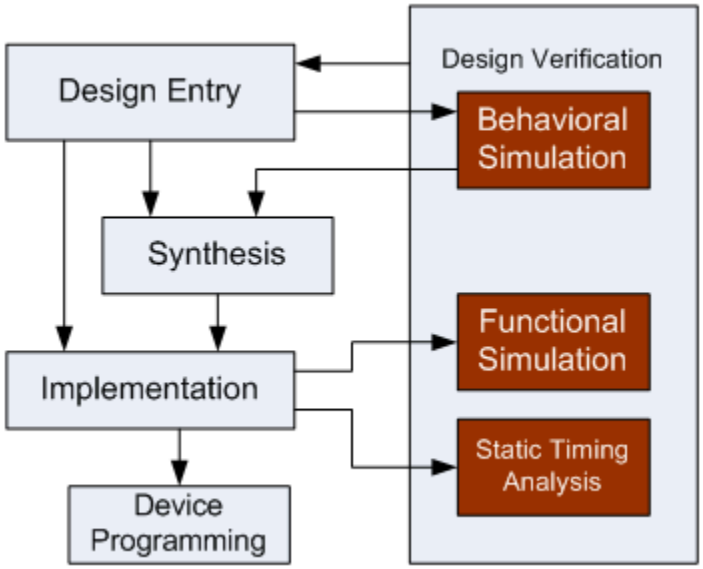

Figure 4.2: FPGA Design Flow

Note: Design Entry includes **Functional/Device Specification** and **HDL Coding** and Implementation includes **Mapping** and **Placement and Route**.

Functional/Device Specification:

On this stage, the designers need to set up the configuration (make/model/speed/class/device family) of the FPGA. After that, the designing software will conduct some preliminary setup for the particular FPGA device's intellectual properties (IPs), designs and components [4],[17],[18].

HDL Coding:

On this stage, the designers write HDL codes to model the entire digital designs. The designers can also use a schematic approach to model the entire circuits. We will show both approaches in **Chapter 5.2**; however, since we need to implement an algorithm in this thesis, the HDL coding approach is preferred [4],[17],[18].

Logic Synthesis:

Synthesis is a process that converts the HDL codes to the gate-level netlist, which describes the different types of components, elements, and interconnections between the components and other details like area occupied and temperature of operation, etc. Also, synthesis will help check the syntax of the HDL codes and map the design to a particular FPGA family [4],[17],[18].

Mapping:

In this stage, the software maps the generic logic design to the logic technology contained in the selected FPGA device [4],[17],[18].

Placement and Route (PAR):

This stage is one of the most important steps in the entire implementation. Placement decides where the components should be placed on the FPGA while routing is responsible for the connections between different components. PAR is crucial because it is related with the timing and area constraints of the design. A bad placement will result in problematic routing, which leads to design violations [4],[19].

FPGA Configuration and Programming:

The last step of the FPGA design flow is to program the designs on the FPGA and test the circuit. On this stage, the software converts the entire design to a "bitstream" file, which is loaded on the FPGA board. After that, the FPGA is ready to run the digital design [4],[19].

Design Verification:

It is extremely important for every design to meet certain standards and satisfy certain conditions. After each design step, the designers need to check if their circuits meet various constraints such as timing, area and functional logic [19].

In this thesis, we will use ModelSim as the simulation tool for design verification. The descriptions of each testing stage are given below:

1. **Behavioral Simulation:** This task is to verify the functionality of the HDL codes.

2. **Gate-Level Simulation:** The gate-level netlist is generated after the completion of synthesis. This task is to test the timing and gate-level functionality of the design.

3. **Static Timing Analysis (STA):** STA comes out after the completion of PAR. The designer will analyze some important issues related with the design of the circuit such as setup and hold times, critical path and clock skews. The STA will examine every possible path in the circuit and help debugging glitches and slow paths [4].

4. **Post-PAR Timing Simulation:** This task provides a comprehensive timing summary of the circuit.

Chapter 5. FPGA DESIGN TUTORIAL IN QUARTUS II

In this thesis project, we will use Quartus II Web Edition FPGA design software. It is a software tool produced by Altera for analysis and synthesis of HDL designs, which enables the developers to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

First of all, we should download the software from Altera's official website, http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html. We may just select to download the Quartus II Web Edition **v14.0**, which is free and should be good enough for our project. The Quartus II FPGA design software is only workable on **Windows XP, 7 or 8 or Linux** so far. For Mac users, it is not very convenient, so they may need to install a Windows based operating system (Window7 for this project) by installing **Parallels** first on their Macs. There may be some other better ways to do this; however, from my own working experience, I highly recommend you just choose a workstation with Windows based OS to avoid any further problems in your design and simulation.

After installing your Altera Quartus II, you can open your software by clicking the shortcut icon on the desktop or **Start Manu ➔ All Programs ➔ Altera 14.0 directory ➔ Quartus II 14.0 directory ➔ Quartus II 14.0**. After opening Altera Quartus II software, Figure 5.1 will display.

Figure 5.1: Opening Display

Then the design console will display as in Figure 5.2 automatically.

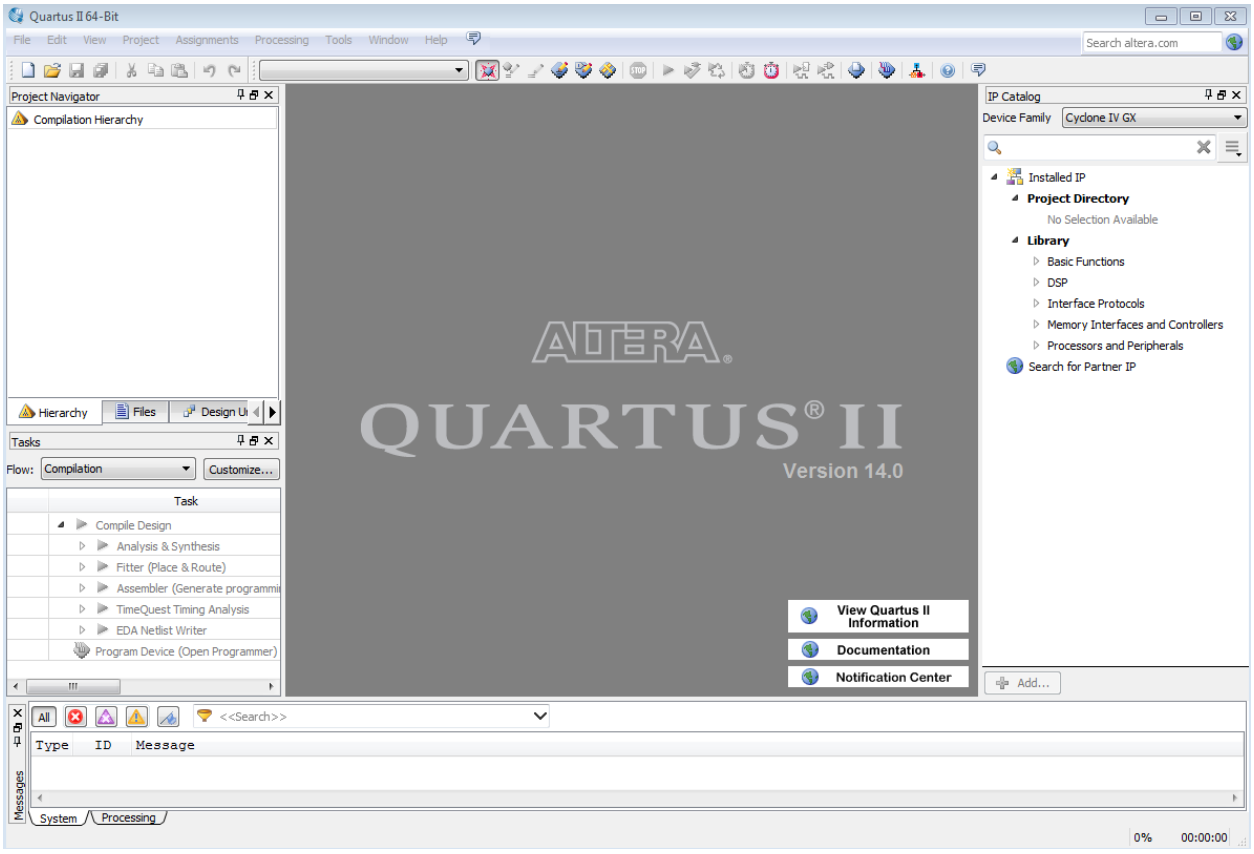

Figure 5.2: Design Console

5.1 Project Setup

1. To create a new project, from the **File** menu or opening display in Figure 5.1, select **New Project Wizard**. Click **Next** to go through the introduction screen if it appears. Then, the Figure 5.3 will appear. Fill in the fields in Figure 5.3 (make sure there are no spaces in any of your entries). The program will ask you if it should create the specified directory if it does not exist; choose **yes**.

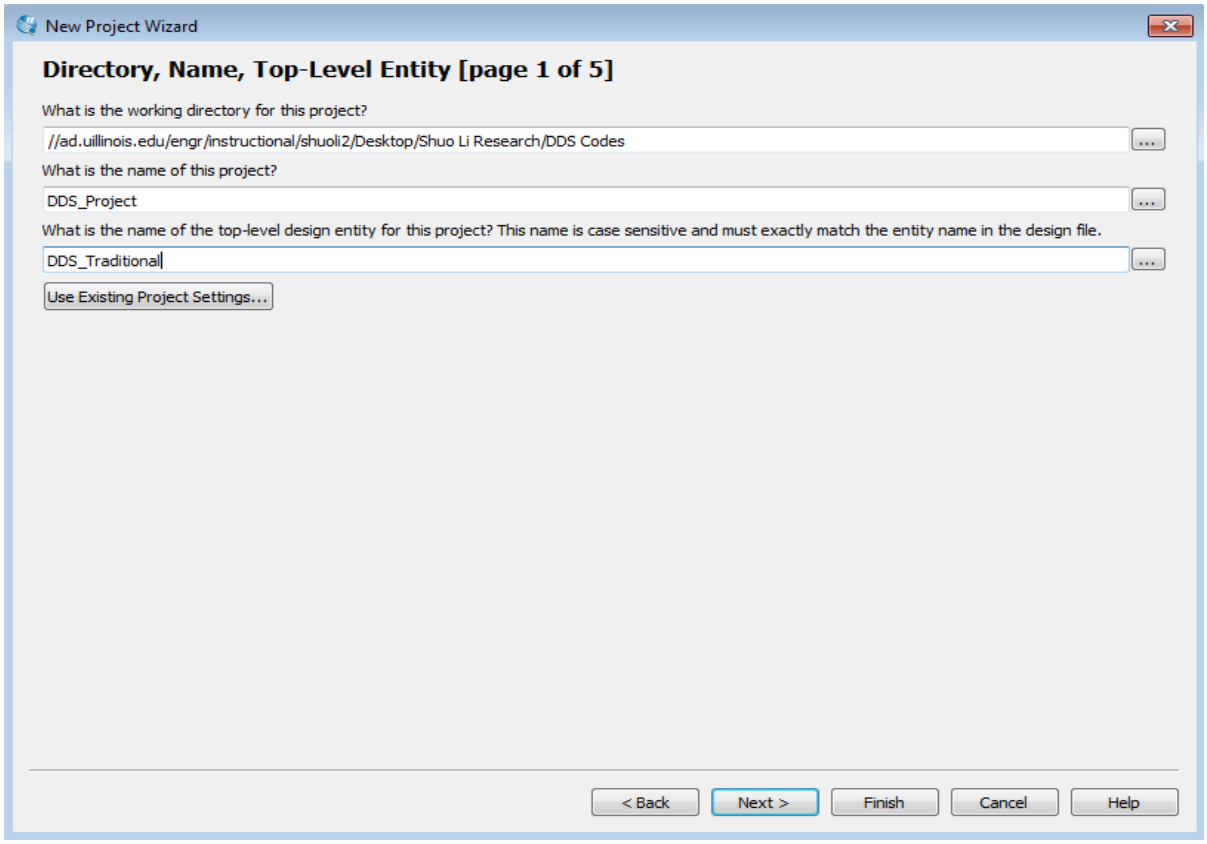

Figure 5.3: New Project Wizard

2. Select **Next** on page 2 without adding any files. On page 3, select **Cyclone IV** for the device family, make sure the second option under target device is selected, and choose **EP4CE115F29C7** in the available devices list according to Figure 5.4. Then click **Next** to page 4.

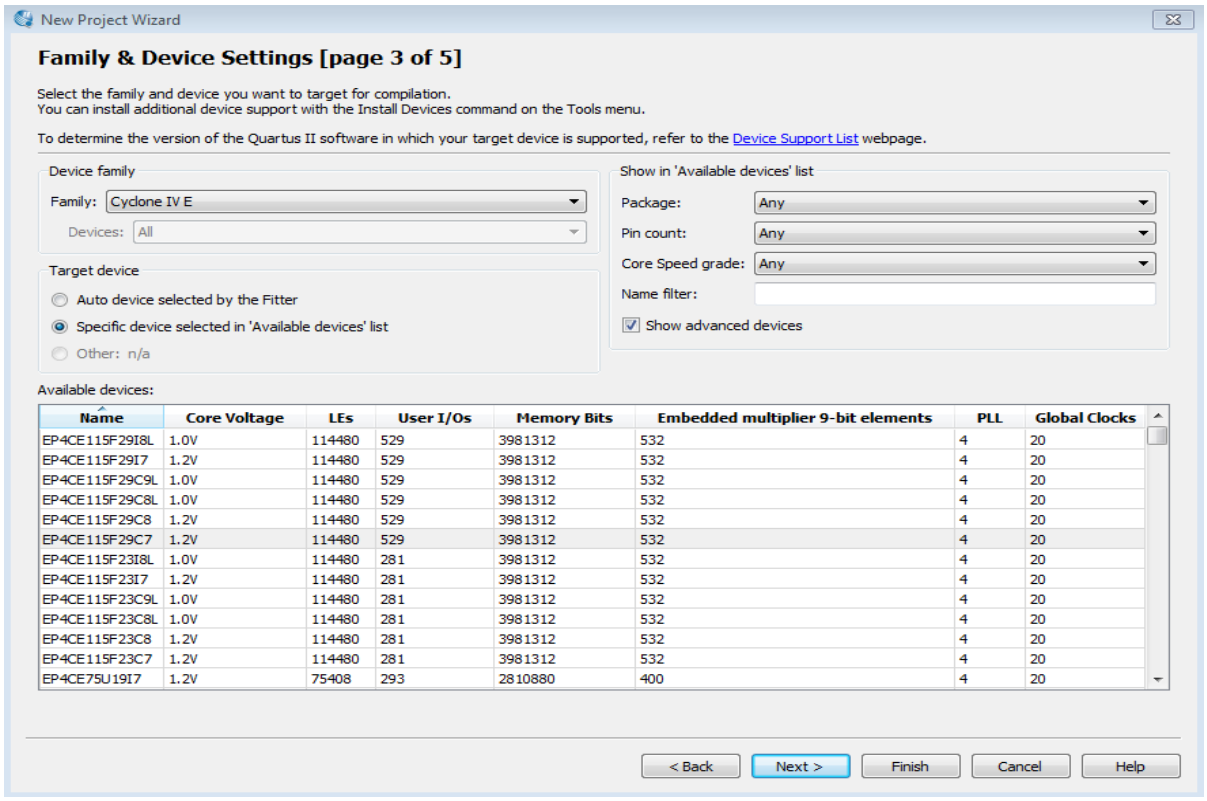

Figure 5.4: Device Specifications

3. Select **ModelSim-Altera** as the simulation tool name, and **Verilog HDL** as the simulation format, then click **Next**. See Figure 5.5.

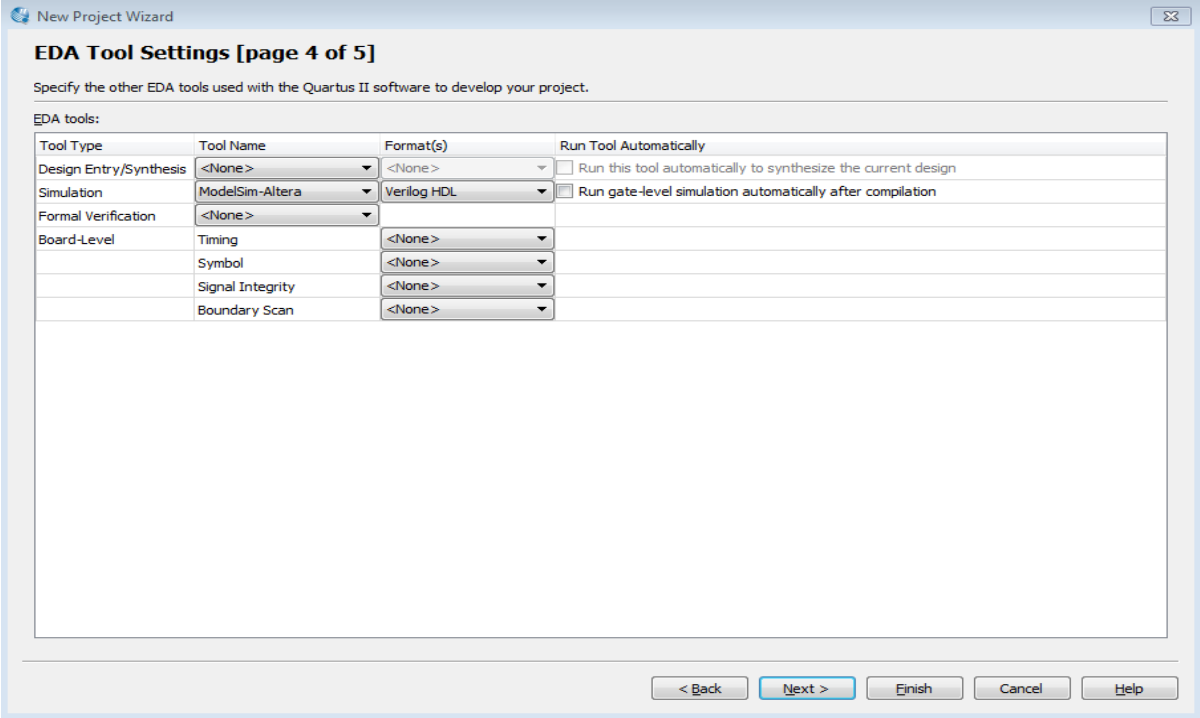

Figure 5.5: EDA Tool Settings

4. Click **Finish** on page 5. Now you should be able to have an entry for the project in the Project Navigator window. It should display as in Figure 5.6.

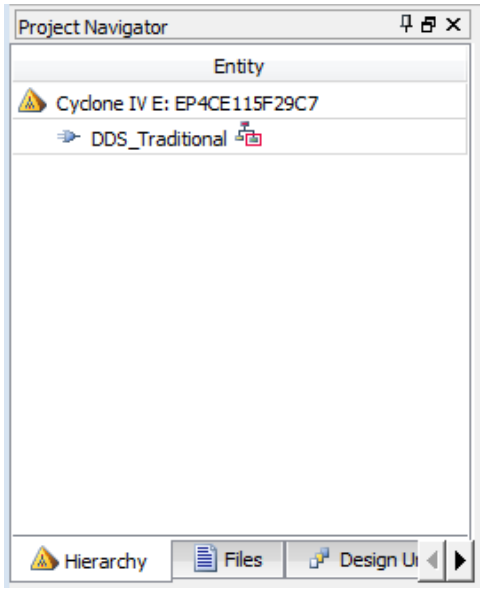

Figure 5.6: Project Navigator

5.2 HDL Coding

1. To create the top-level module (DDS_Traditional.v), go to **File** ➔ **New** or click on the icon in the top left corner. A panel will display as in Figure 5.7. Under the Design Files, select **Verilog HDL File**. Click **OK**. Now we have a blank Verilog file shown as in Figure 5.8.

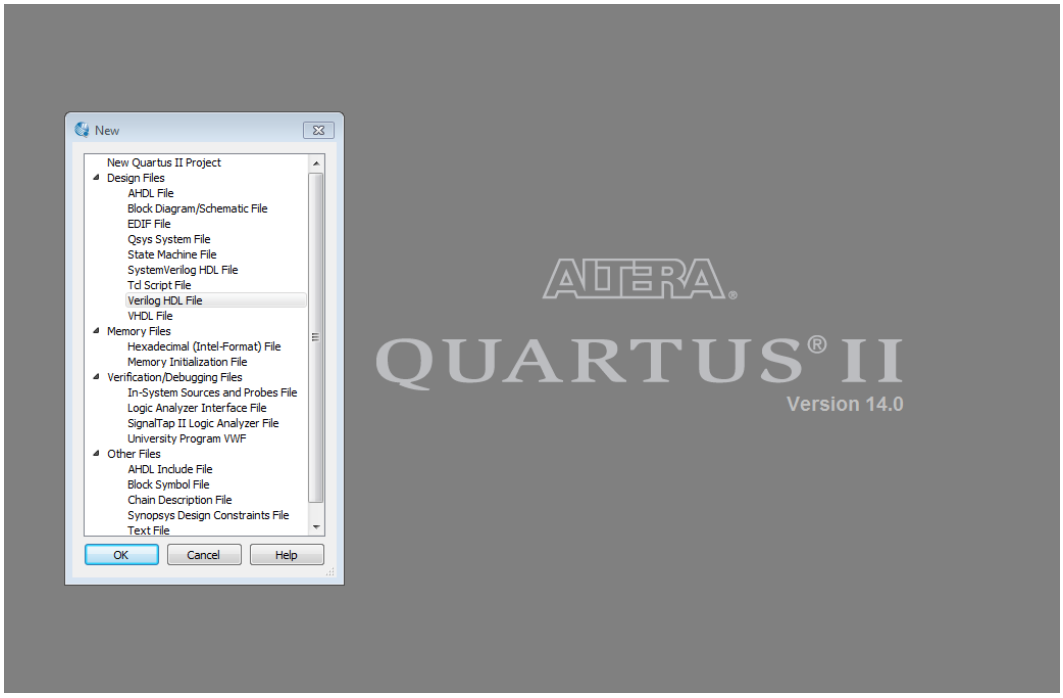

Figure 5.7: New Panel

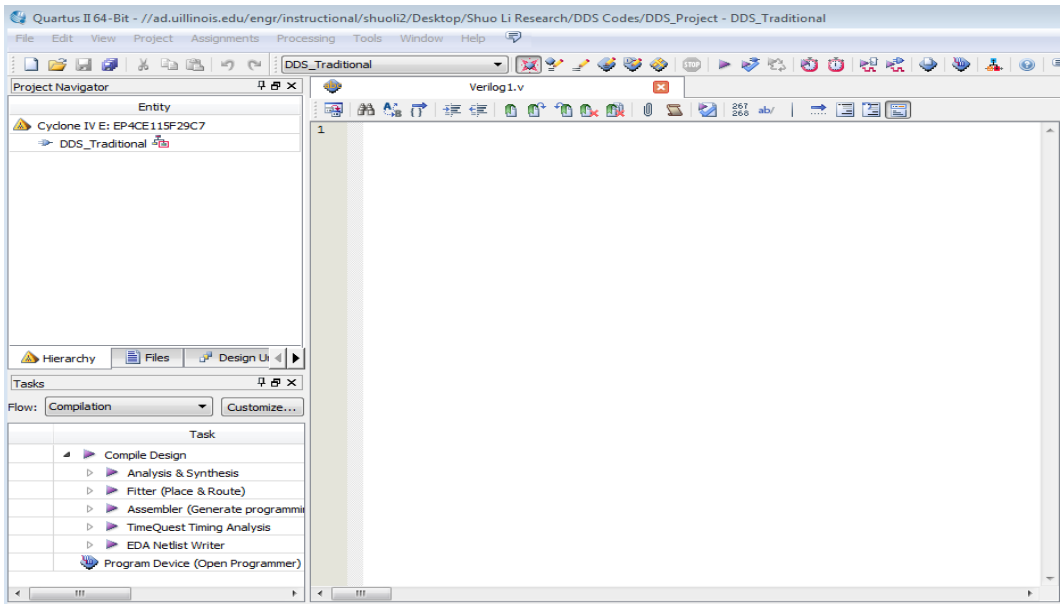

Figure 5.8: Blank Verilog File

2. After generating several blank Verilog files, we will need to name them, include them into the project and set up a top level file, which has to be the **DDS_Traditional.v**. As shown in Figure 5.9, we name one of the blank files DDS_Traditional.v, then click **Save**. Remember to check the **Add file to current project**.

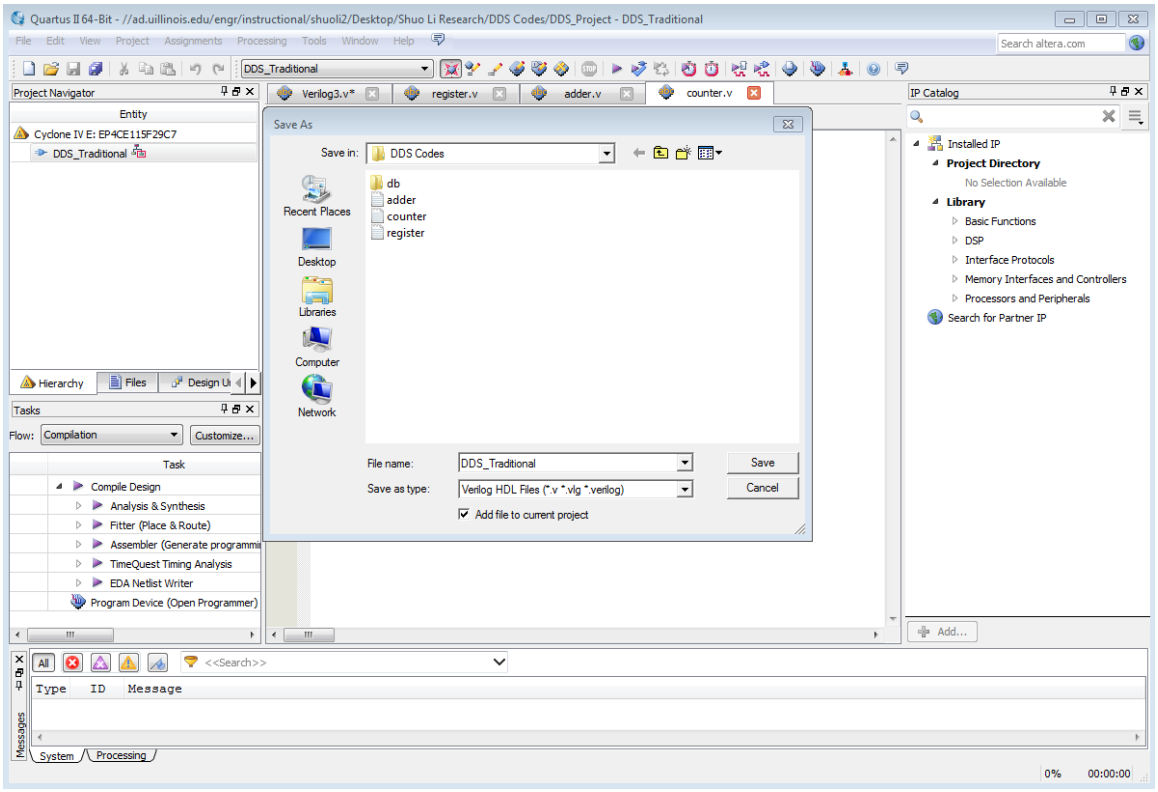

Figure 5.9: Naming the Blank Verilog File

Click **Files** in the Project Navigator. Among several files, choose **DDS_Traditional.v**; right click it and select **Set as Top-Level Entity** as shown in Figure 5.10.

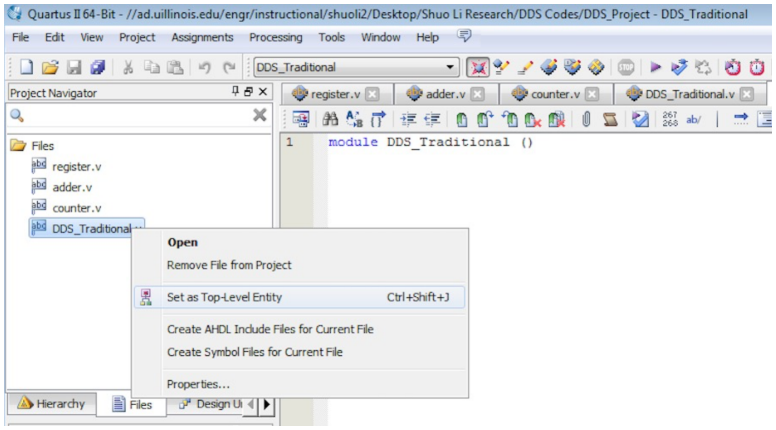

Figure 5.10: Set Up the Top Level File

3. Start coding both structures of DDS. First, design a DDS with traditional structure in Verilog. The code for top-level module is shown in Figure 5.11.

```
module DDS_Traditional (clk, reset, out);

input clk, reset;
output [7:0] out;

reg [31:0] fcw;
reg [7:0] out;

wire clk, reset;
wire [31:0] q1, q2, sum;
wire [7:0] LUT_out, address;

register reg1(.clk(clk), .reset(reset), .d(fcw), .q(q1));
register reg2(.clk(clk), .reset(reset), .d(sum), .q(q2));
adder add(.a(q1), .b(q2), .sum(sum));
phase_truncator trun(.clk(clk), .reset(reset), .pa_out(q2), .address(address));
look_up_table lut(.clk(clk), .reset(reset), .address(address), .LUT_out(LUT_out));

always @(posedge clk or posedge reset)
begin
    if (reset == 1'b1)
            fcw <= 32'd429496730; //5 MHz output
    else
            out <= LUT_out;
end
endmodule
```

Figure 5.11: Top-Level Verilog Codes for DDS with Traditional Structure

4. Second, design a DDS with truncation spurs-free structure in Verilog. The Verilog codes for the additional substractor (adder) and the comparator as well as the top-level module are shown in Figure 5.12.

```
module comparator(a, b, G);

input [23:0] a, b;
output G;

reg G;

always @ *
    if (a > b) begin
        G <= 1'b0;
    end else if (a == b) begin
        G <= 1'b0;
    end else begin
        G <= 1'b1;
    end
endmodule
```

```
module substractor (a, b, clk, result);

input [7:0] a;
input clk, b;
output [7:0] result;

reg [7:0] result;

always @ (posedge clk)
begin
    if (a == 8'b10000000)
        result = 8'b10000001; //handle the overflow
    else
        result = a - b;
end
endmodule
```

Figure 5.12: Verilog Codes for DDS with Truncation Spurs-Free Structure

```
module DDS_Free (clk, reset, out);

input clk, reset;
output [7:0] out;

reg [31:0] fcw;
reg [7:0] out;

wire clk, reset, G;
wire [31:0] q1, q2, sum;
wire [7:0] LUT_out, address, dds_output;
wire [23:0] address_fractional, LUT_fractional;

register reg1(.clk(clk), .reset(reset), .d(fcw), .q(q1));
register reg2(.clk(clk), .reset(reset), .d(sum), .q(q2));
adder add(.a(q1), .b(q2), .sum(sum));
phase_truncator trun(.clk(clk), .reset(reset), .pa_out(q2), .address(address),
.address_fractional(address_fractional));
look_up_table lut(.clk(clk), .reset(reset), .address(address), .LUT_out(LUT_out),
.LUT_fractional(LUT_fractional));
comparator comp(.a(address_fractional), .b(LUT_fractional), .G(G));
substractor sub(.clk(clk), .a(LUT_out), .b(G), .result(dds_output));

always @(posedge clk or posedge reset)
begin
    if (reset == 1'b1)
            fcw <= 32'd429496730;
    else
            out <= dds_output;
end
endmodule
```

Figure 5.12: Continued

5. In Figure 5.12, we can see that the DDS with Truncation Spurs-Free Structure actually adds a comparator block and an adder block on the top of the traditional DDS design. We use HDL to link all these modules together in this thesis. Now, we will also show a schematic design entry approach to link the modules together. This is to create the top-level entity as a schematic diagram, rather than as a Verilog module. In some cases, this approach is easier, more straightforward and less time-consuming than wiring the circuits up directly in codes.

   Note: This part is to introduce another approach for designing the digital circuits; however, it is not the approach we used in this thesis project. We use **10** as the ROM width to avoid confusion. However, **Step 6** on how to program the ROM is very important. It is applicable to both HDL and Schematic design approaches.

   To start, we need to create a blank schematic file. Go to **File ➔New** and select **Block Diagram/Schematic File** and click **OK**. It will generate a **[.bdf] file**. The Figure 5.13 will pop up. Then we should give this blank schematic file an appropriate name, which

has to be **same** as what we typed in the top-level file name in Figure 5.3. In this case, it should be "**DDS_Free**".

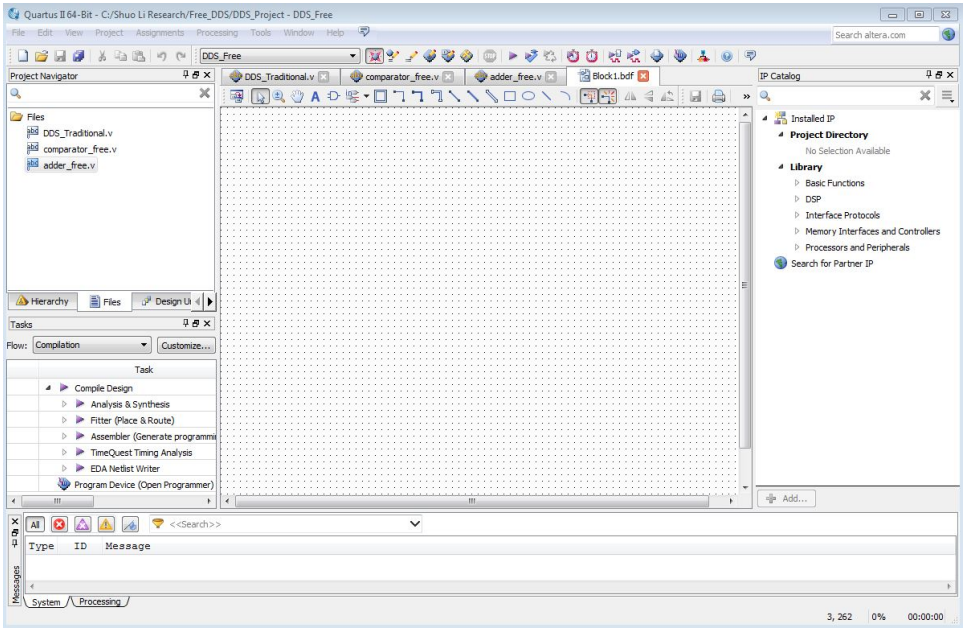

Figure 5.13: Blank Schematic File

Then we should generate the block diagrams from the Verilog codes. In the **Project Navigator**, select **Files**, then right click the Verilog file you want to generate a block diagram from and click **Create Symbol Files for Current File** as shown in Figure 5.14.

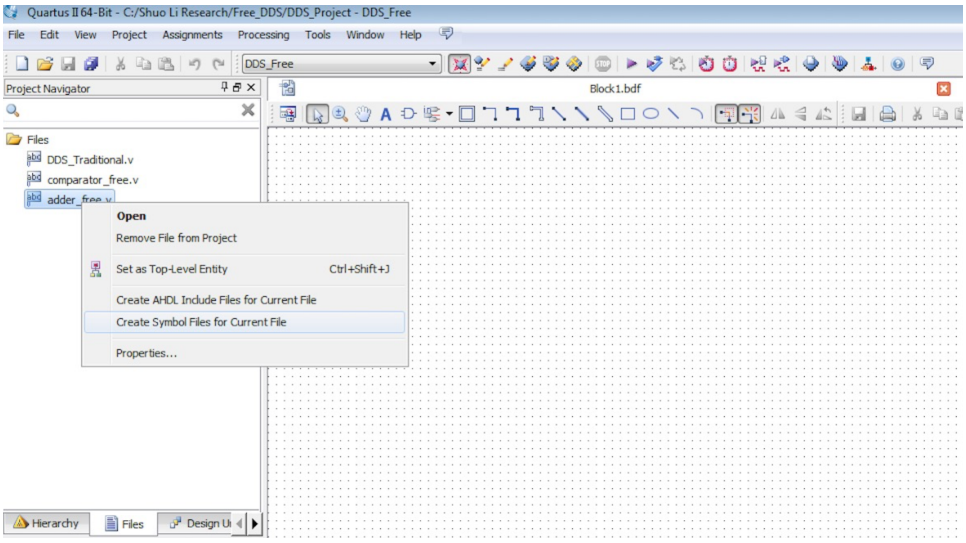

Figure 5.14: Generating a Block Diagram from Verilog Codes

The software will create the symbol file after compiling this single Verilog file. If the creation is successful, then the **Message console** should show no errors as in Figure 5.15.

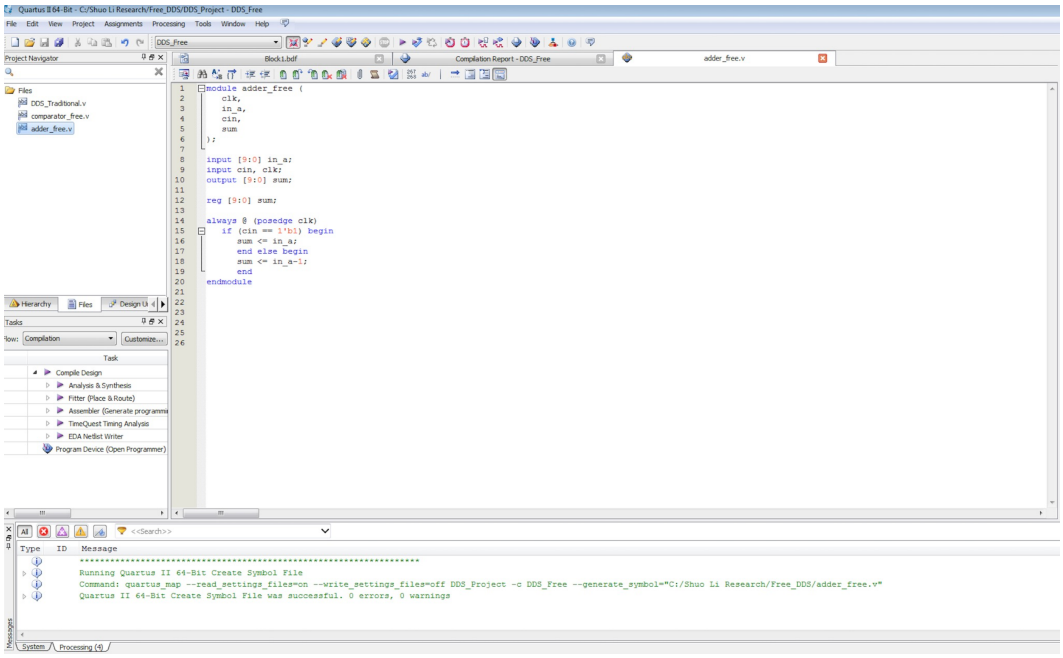

Figure 5.15: Successful Creation

After that, you can add the symbol into the schematic file by switching back to the blank schematic file. Right click on the schematic and select **Insert ➔Symbol**, then the symbol window will display as Figure 5.16. In Libraries, select **Project**. The available symbols will be listed. Select **adder_free** and click **OK**.

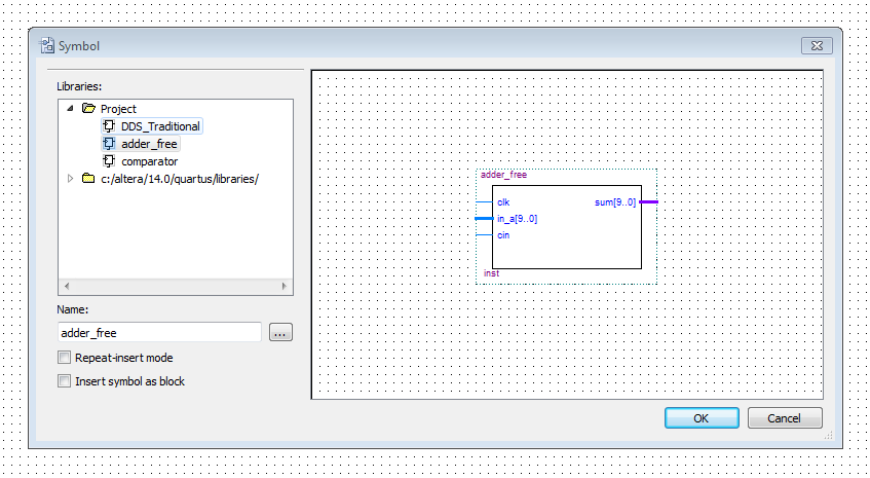

Figure 5.16: Symbol Window

Then we can place all the symbols in the schematic as shown in Figure 5.17.

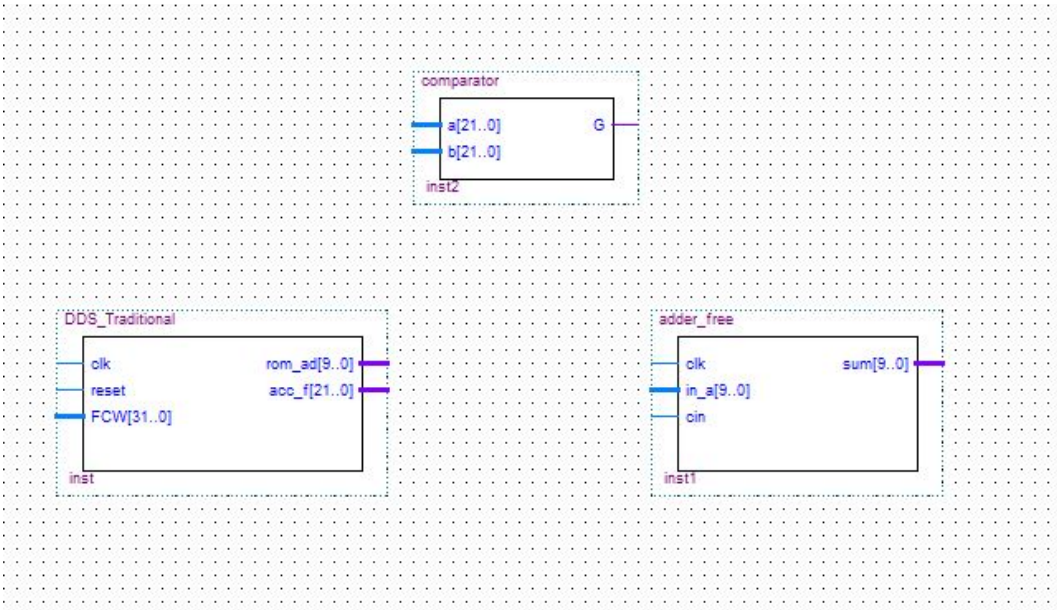

Figure 5.17: Symbol Placement

6. So far, we have most of the required modules to build the DDS except the LUT. To create the LUT, we need a [.mif] file containing the ROM data first. We can generate this file using MATLAB. The MATLAB code to generate this [.mif] file is shown in Figure 5.18. This "sin.mif" file is for designing the traditional DDS. The size of this ROM is $2^8 \times 8$. Instead, the width of ROM for DDS with truncation spurs-free structure is 32. We can do this simply by adjusting the width parameter in the code while keeping depth = 2^8.

rom.m — Edited

rom.m 〉 No Selection

```
1  width=8; //bit width
2  depth=2^8; //data depth
3  t=linspace(0,6.28,depth);
4  sin_val=sin(t);
5  sin_val=fix(sin_val*(2^width-1)/2+0.5); //result after rounding
6
7  addr=[0:255];
8
9  file=fopen('f:/My Works/FPGA Examples/sin.mif','wt');
10 fprintf(file,'WIDTH=%d;\n',width);
11 fprintf(file,'DEPTH=%d;\n',depth);
12 fprintf(file,'ADDRESS_RADIX=UNS;\n');
13 fprintf(file,'DATA_RADIX=DEC;\n');
14 fprintf(file,'CONTENT BEGIN\n');
15 for i=1:depth
16 fprintf(file,'%d:%d;\n',addr(i),sin_val(i));
17 end
18 fprintf(file,'END;\n');
19 fclose(file);
```

Figure 5.18: MATLAB Codes for Generating [.mif] File

After creating the [.mif] files, we can start to create the ROM module with Quartus II following the below steps.

First, select **Tools ➔IP Catalog**, then an IP Catalog will pop up on the right. Under **Library ➔Basic Functions ➔On Chip Memory**, select **ROM: 1-PORT** and click **Add**. The process is shown as in Figure 5.19.

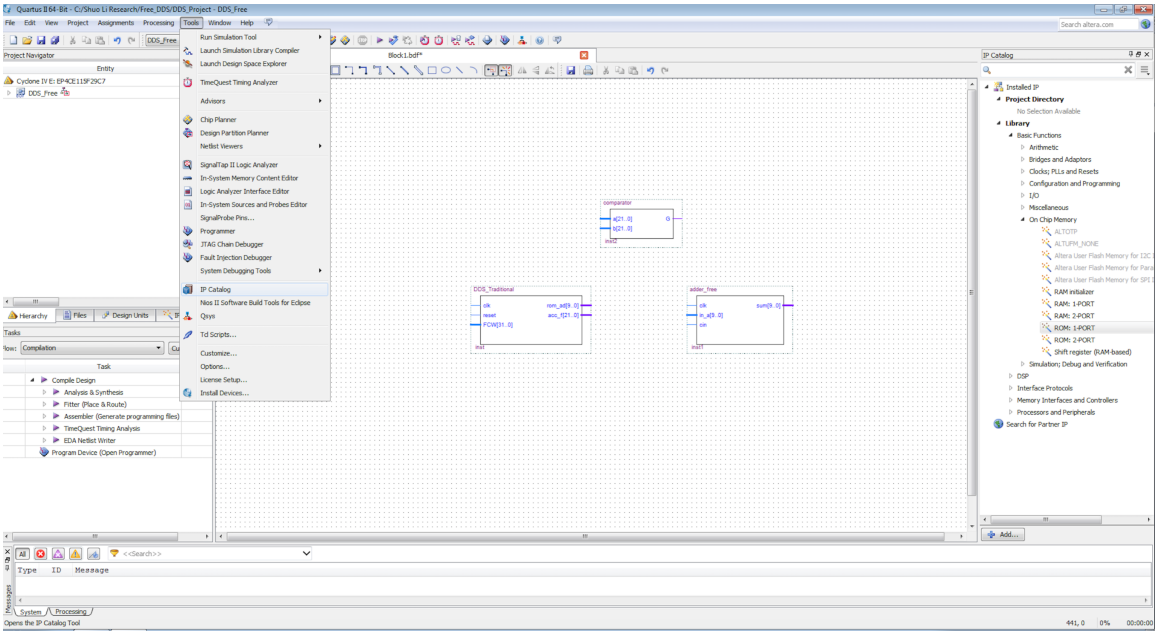

Figure 5.19: Create a 1-Port ROM

After that, a window as shown in Figure 5.20 will pop up to ask a file name and file type. Fill in the name with **ROM_Free** and select **Verilog** as the file type. Then, click **OK**.

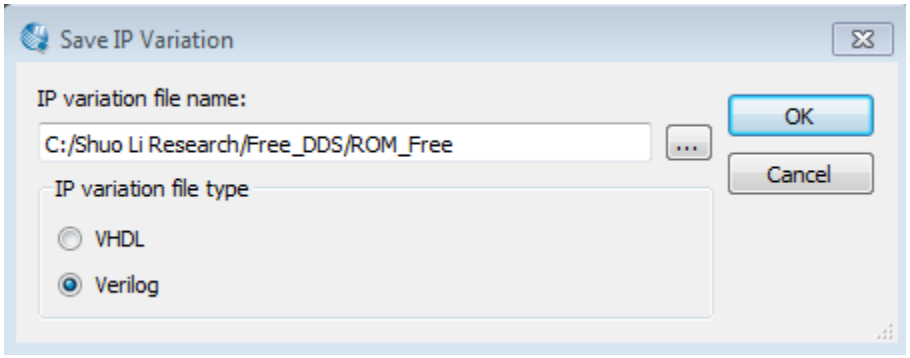

Figure 5.20: Save IP Variation

As shown in Figure 5.21, the MegaWizard Plug-In Manager window will pop up. On page 1, fill in the width (32 bits) and depth (2^{10} = 1024 words) of the ROM and leave the others as default. Click **Next**.

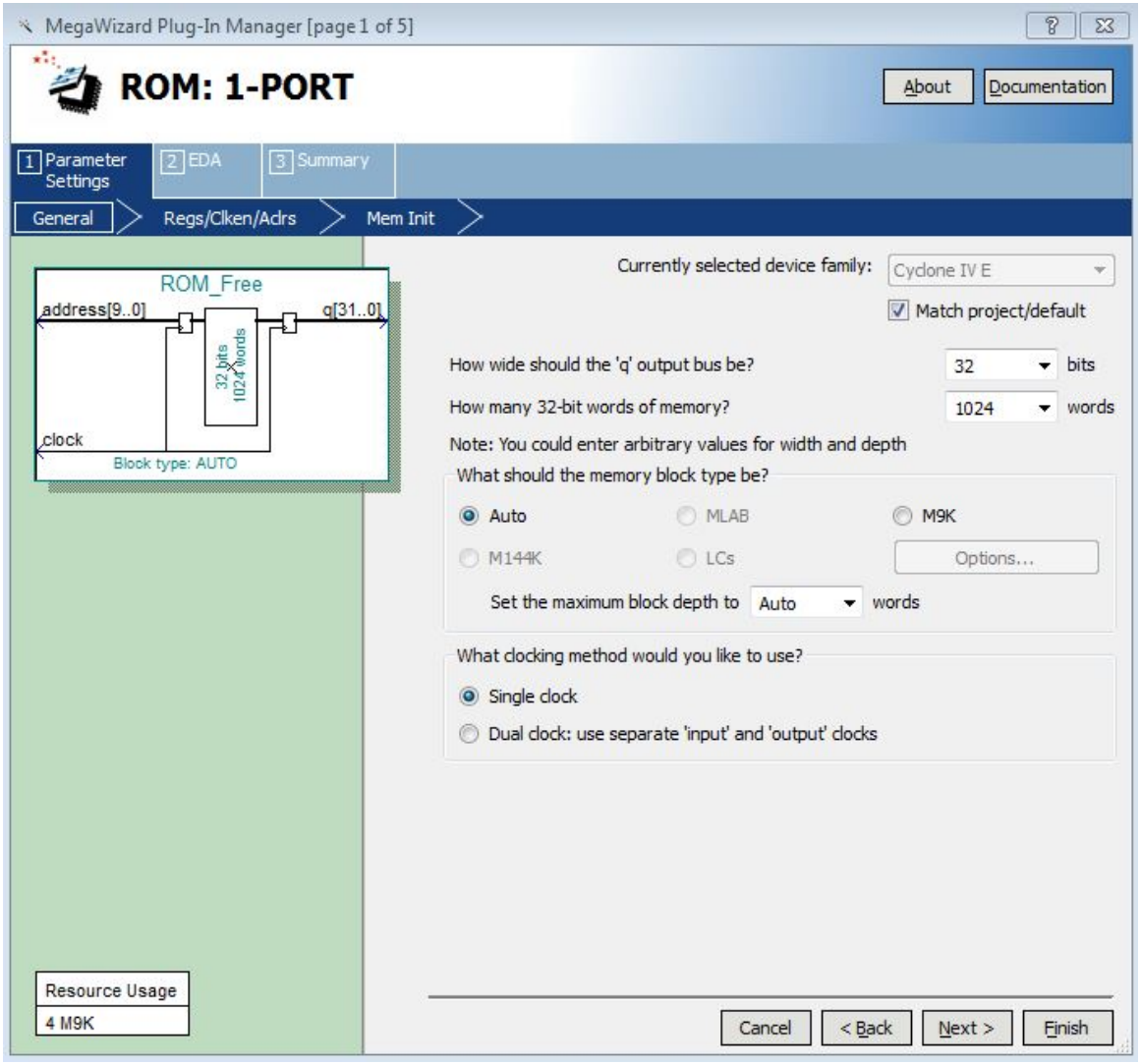

Figure 5.21: Parameter Setting

Leave the page 2 as default but make sure the 'q' output port is **registered**. Click **Next**. On page 3, we will need to do the memory initialization. As we mentioned above, we will need to use the **[.mif]** file generated by MATLAB to initialize the ROM. As shown in Figure 5.22, browse to the file location and select the "**sin_free.mif**" file, then click **Open**. Leave the others as default. Click **Next**.

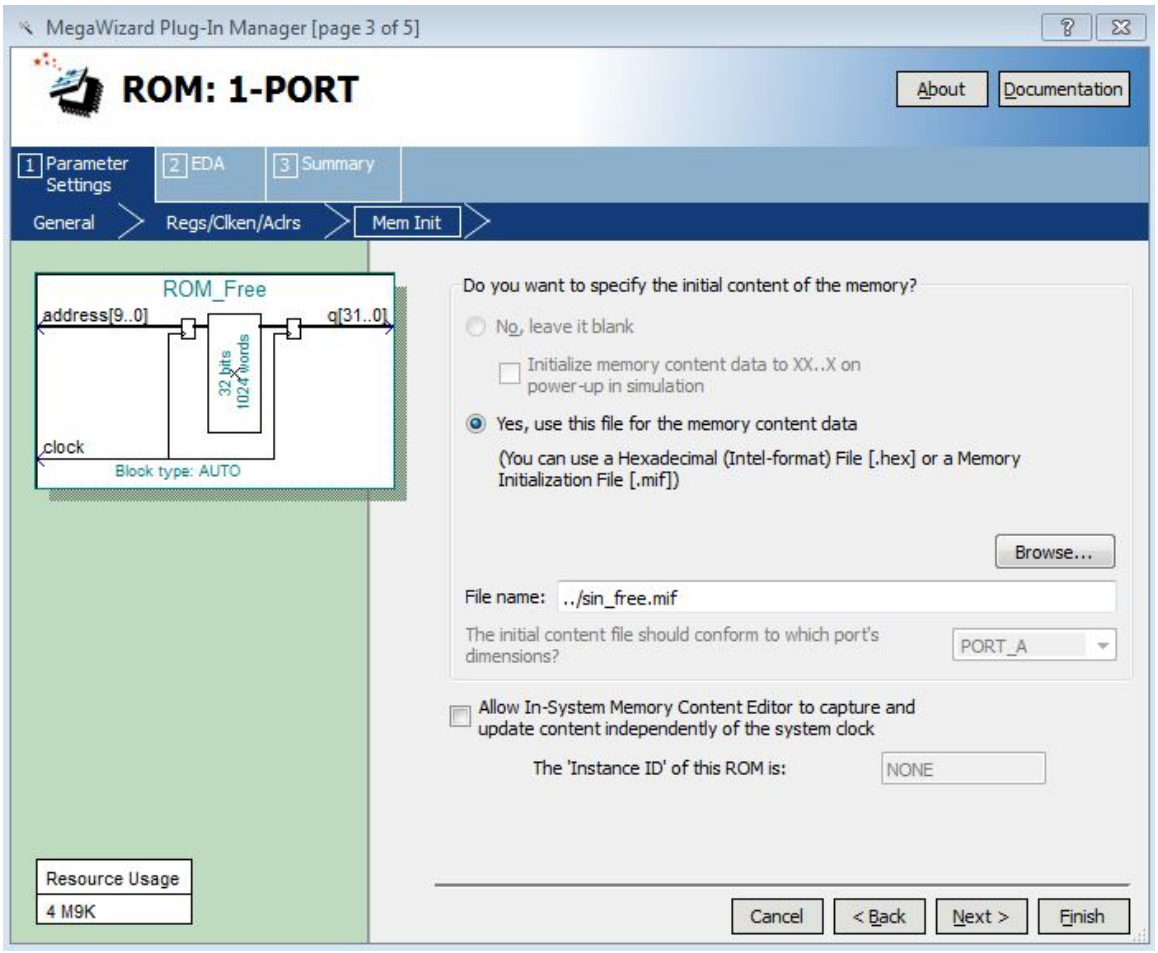

Figure 5.22: Memory Initialization

On page 4, it is optional to select generate netlist for timing and resource. On page 5, check the box before "**ROM_Free.bsf**". Then, this symbol file of ROM will be generated. After all the above steps, click **Finish**. The process is as shown in Figure 5.23.

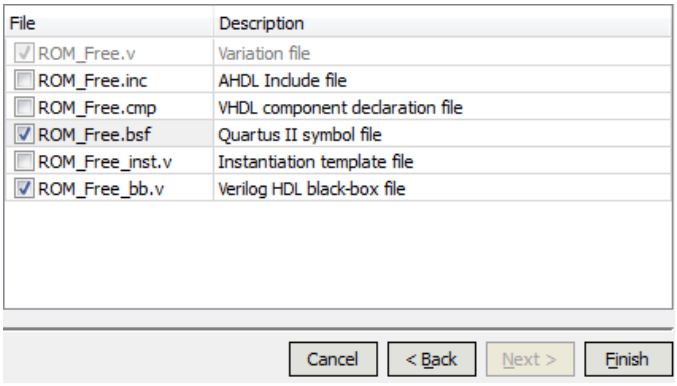

Figure 5.23: Generating Various Files

Now, we will be able to select the symbol file for ROM follow the same step as shown in Figure 5.16. We have all the required modules for designing DDS with truncation spurs-free structure so far. The schematics are shown in Figure 5.24.

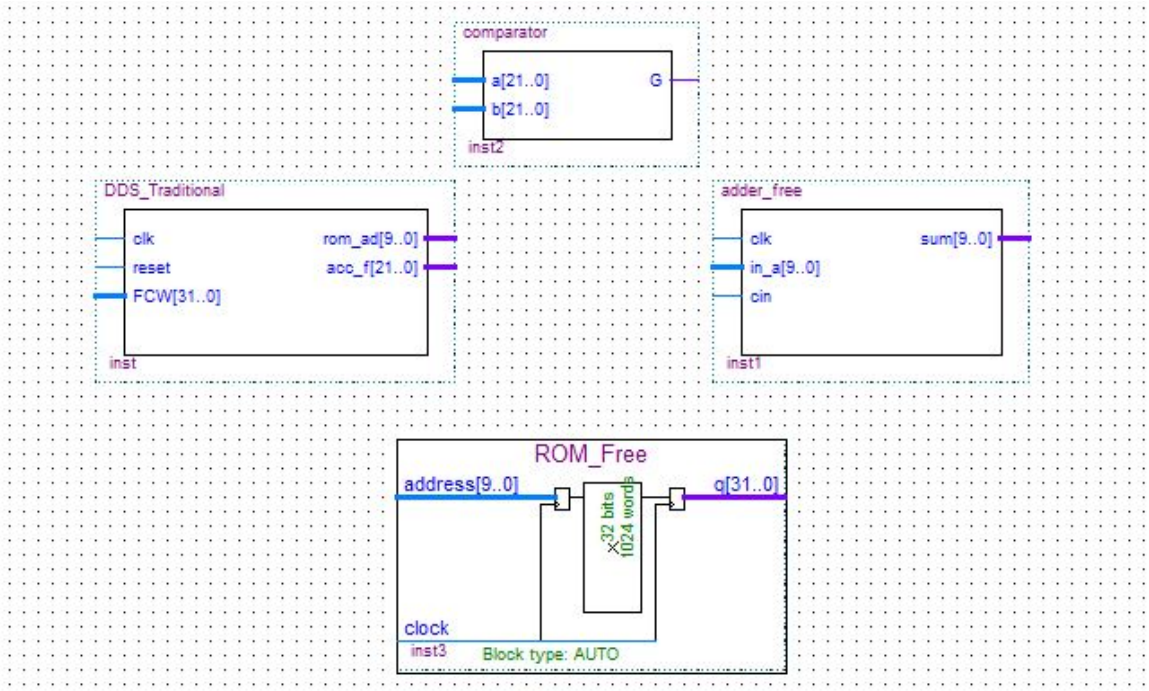

Figure 5.24: Updated Schematic

7. Now, we will need to connect all these blocks together with single wire or bus according to the design in Figure 2.6.

    a) Add the input and output ports by selecting them from tools bar shown in Figure 5.25. Make sure input and output ports are not misused.

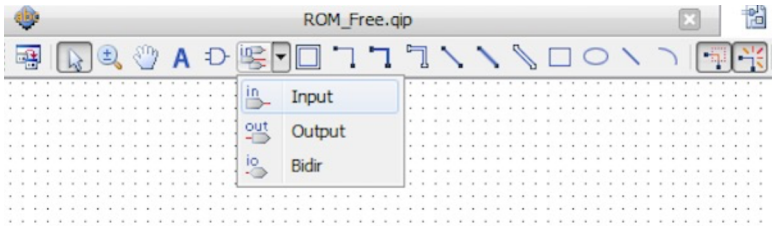

Figure 5.25: Tools Bar for Ports and Wires

b) Add wires and buses to connect the separate modules. Make sure the bit-widths of two ports matched. Adjust the name and width of ports and wires by right clicking on them and selecting **Properties**. Name as shown in Figure 5.26 and click **OK**.

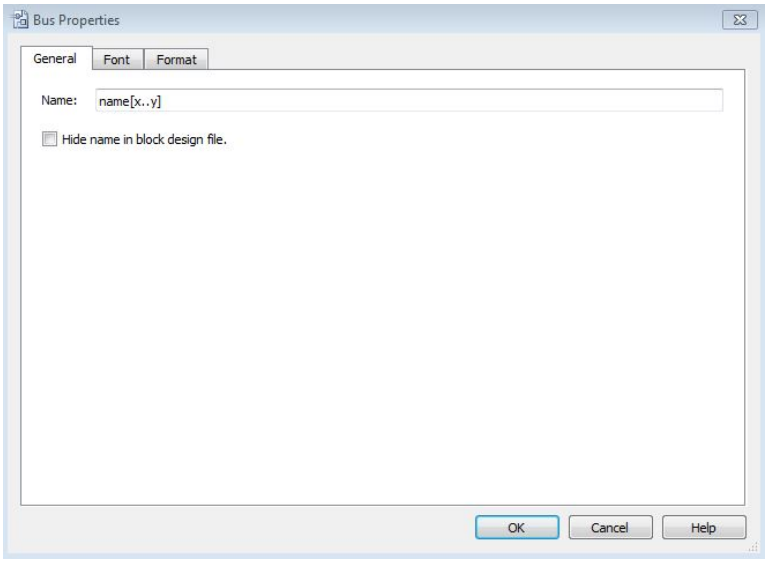

Figure 5.26: Bus Properties

c) After wiring the circuits up, the final top-level schematic is shown as in Figure 5.27.

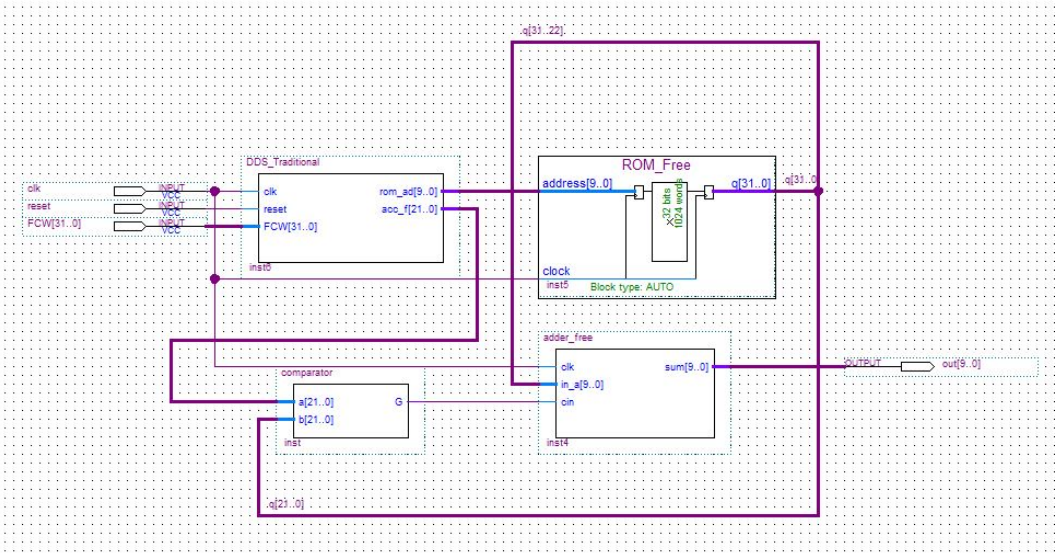

Figure 5.27: Top Level Schematics of DDS with Spurs Free Structure

5.3 Compilation

After finishing the HDL coding for both the traditional and truncation spurs-free structures of DDS design, we will perform compilation with the Quartus II software. The tasks of compilation include logic analysis and synthesis, placement and route (PAR), generation of programming files, timing analysis and EDA netlist.

Before conducting compilation, we can run logic synthesis first. In this phase, Quartus II will check the codes to correct syntax, and generate errors or warnings. If there is an error, we will need to correct the error first, then run **Analysis & Synthesis** again. Quartus II will also build hierarchy in the **Project Navigator** if needed. To start analysis & synthesis, simply click on **Start Analysis & Synthesis** button  in the tool bar [13].

Now, we can start to do the compilation. Click the Start Compilation button  in the tool bar or select **Processing ➔Start Compilation.** We may get a few warnings about timing characteristics and load capacitances this time. Most of the time, we can just ignore them; however errors need to be corrected before continuing. The design console after a successful compilation should be similar to that shown in Figure 5.28.

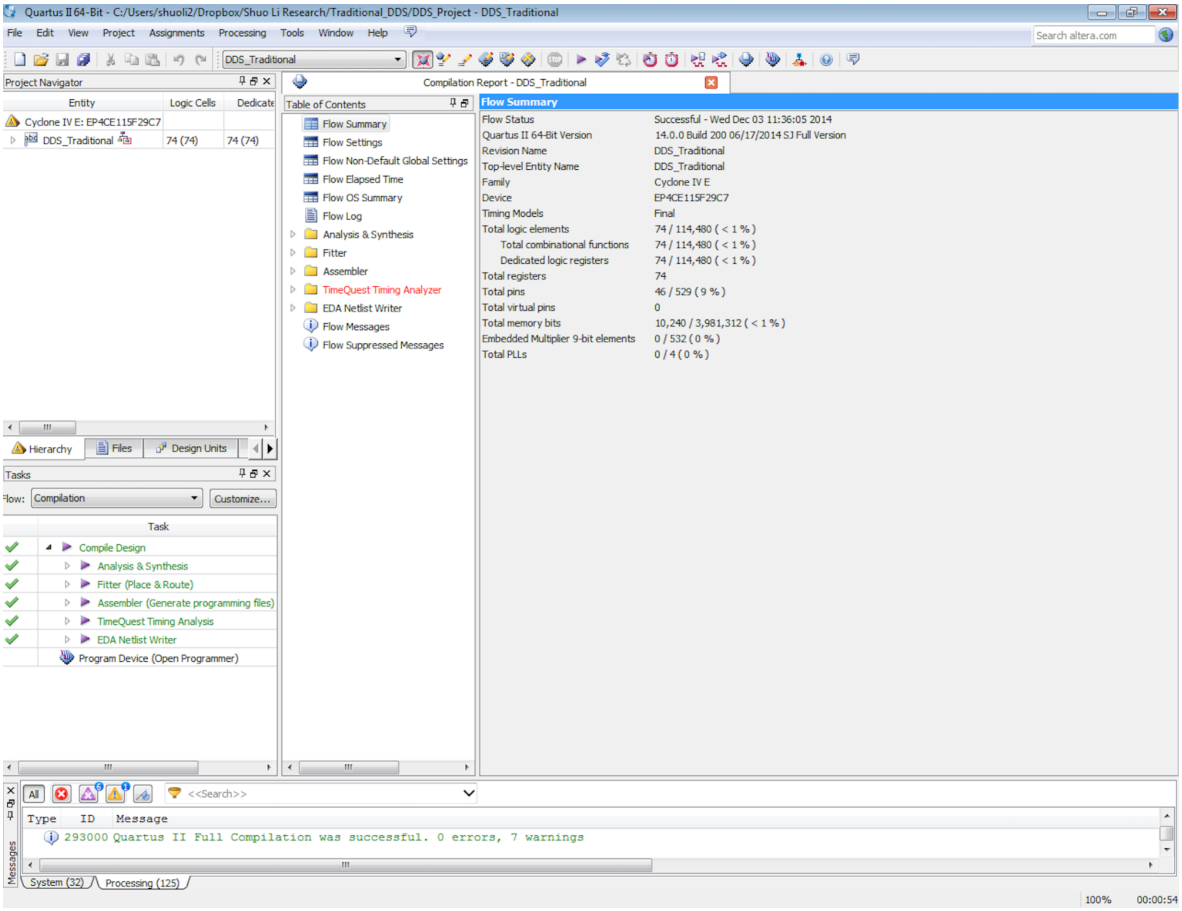

Figure 5.28: Successful Compilation

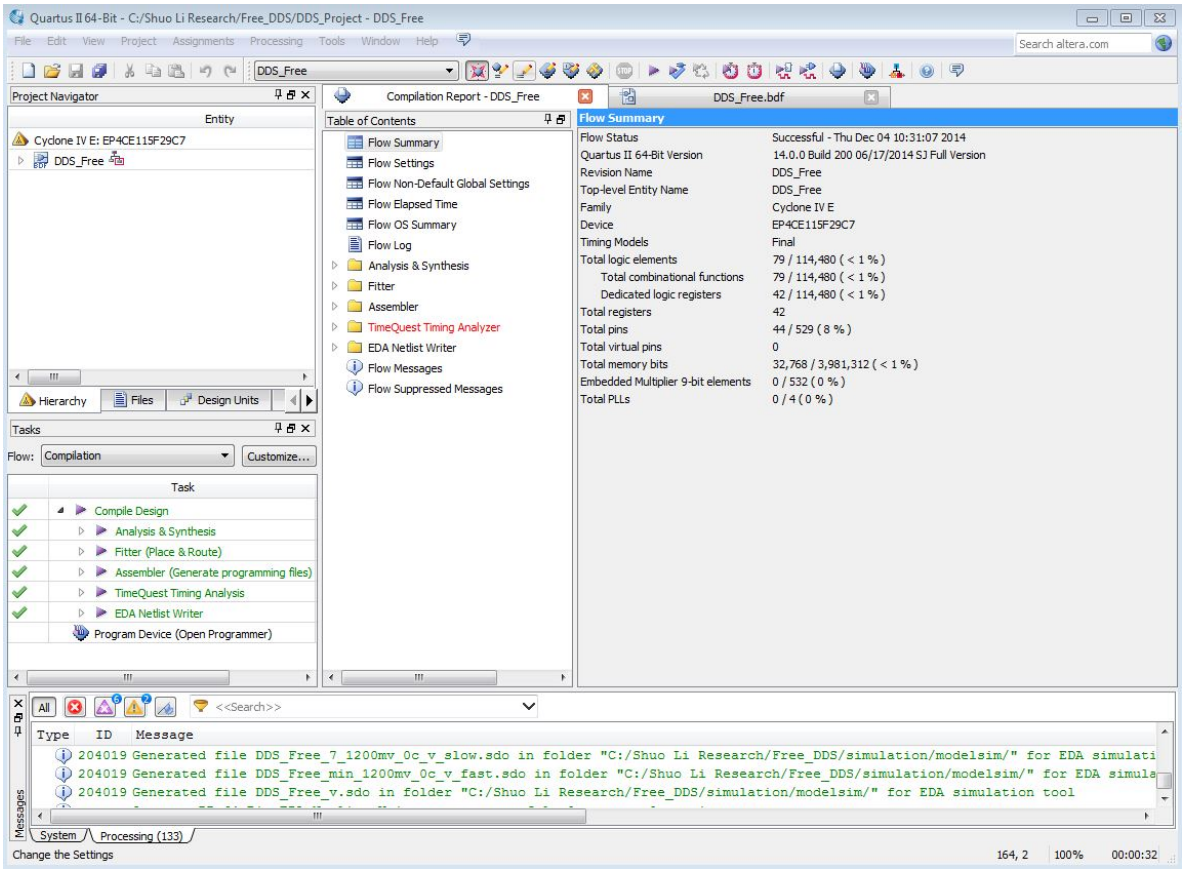

Figure 5.28: Continued

The compilation process also provides us lots of summary reports on resources usage, timing, power consumption and so on. We will give details in **Section 5.5**.

5.4 FPGA Configuration and Programming

Pin Assignments:

To enter the pin assignment, click the Pin Planar button  , or select **Assignments ➔Pin Planar**. Then the pin planar window will automatically pop up as shown in Figure 5.29. The pin layout of the Cyclone IV chip is on the right of the window and the pin assignment table is at the bottom. Enter the pin assignments for **every** pin displayed according to Altera FPGA DE2-115 User Manual [20]. Then, save the pin assignments and **recompile** the entire design.

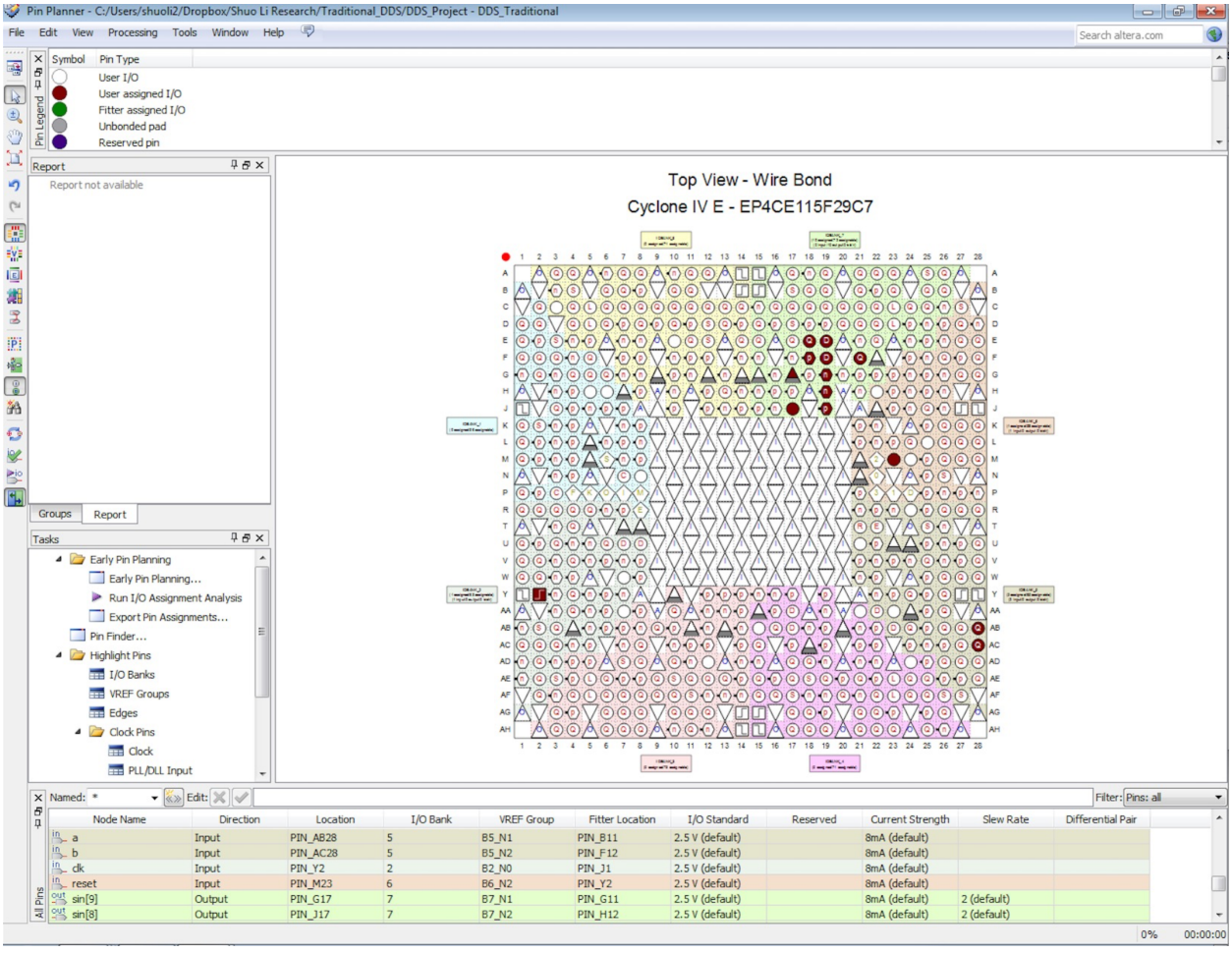

Figure 5.29: Pin Planar

Programming the FPGA:

Now, we are ready to program our design on the FPGA board.

1. Plug in the **power** for the FPGA and connect the **FPGA Blaster Port** to the computer with the included USB cable. The Hex Displays on FPGA should be flashing and the screen should show "Welcome to the Altera DE2-115" on it. The setup should be the same as that shown in Figure 5.30. Note that in the left corner of the FPGA there is a small switch. It should be switched to **RUN** instead of PROG, which sometimes confuses the designers.

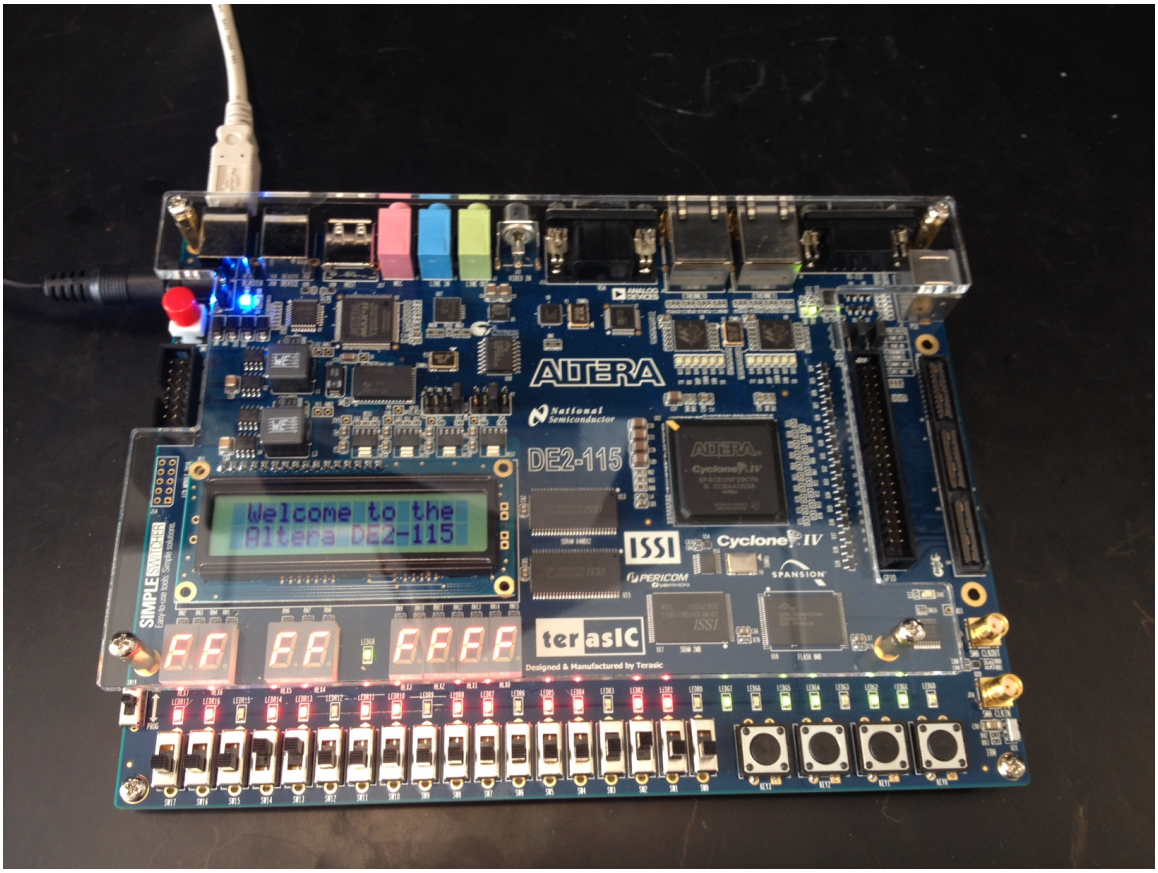

Figure 5.30: FPGA Setup

2. Click the **Programmer button**  or select **Tools ➔Programmer**. The programmer window should pop up as shown in Figure 5.31. Then, on the left of the programmer window, select **Add File**. The Select Programming File window will display. From a list of directories, select **output_files directory ➔ DDS_Traditional.sof**, click **Open**. This process is shown in Figure 5.32 and Figure 5.33.

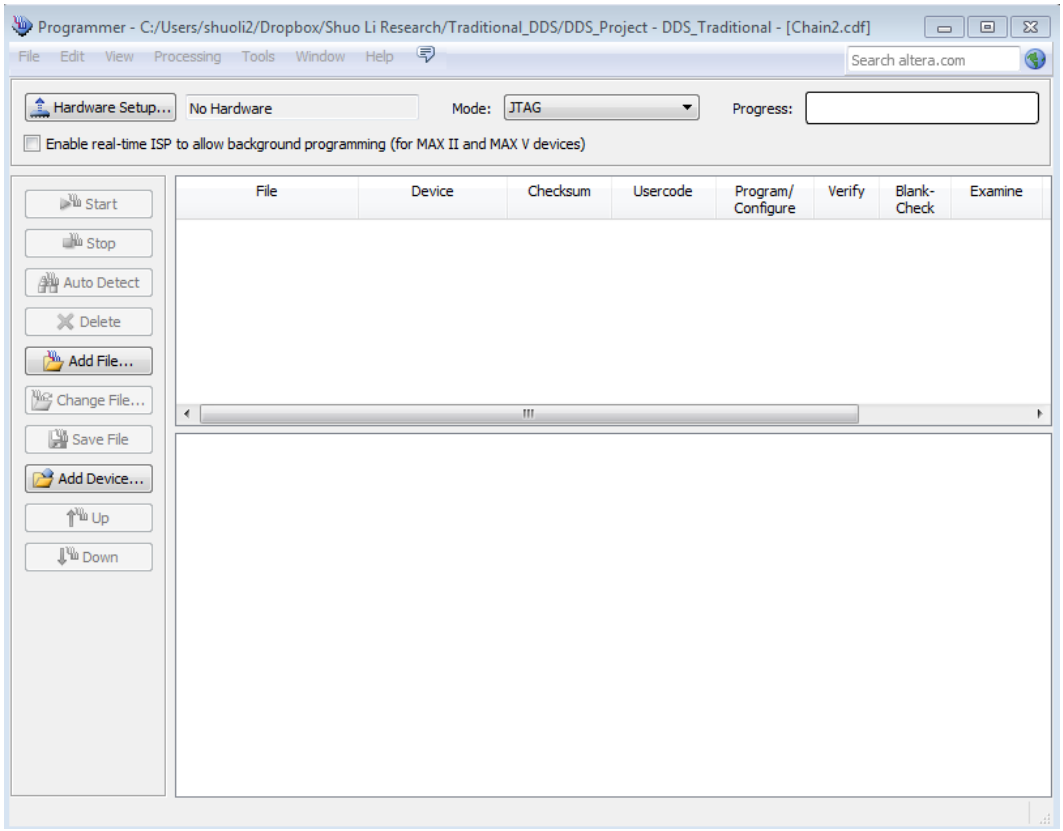

Figure 5.31: Programmer Window

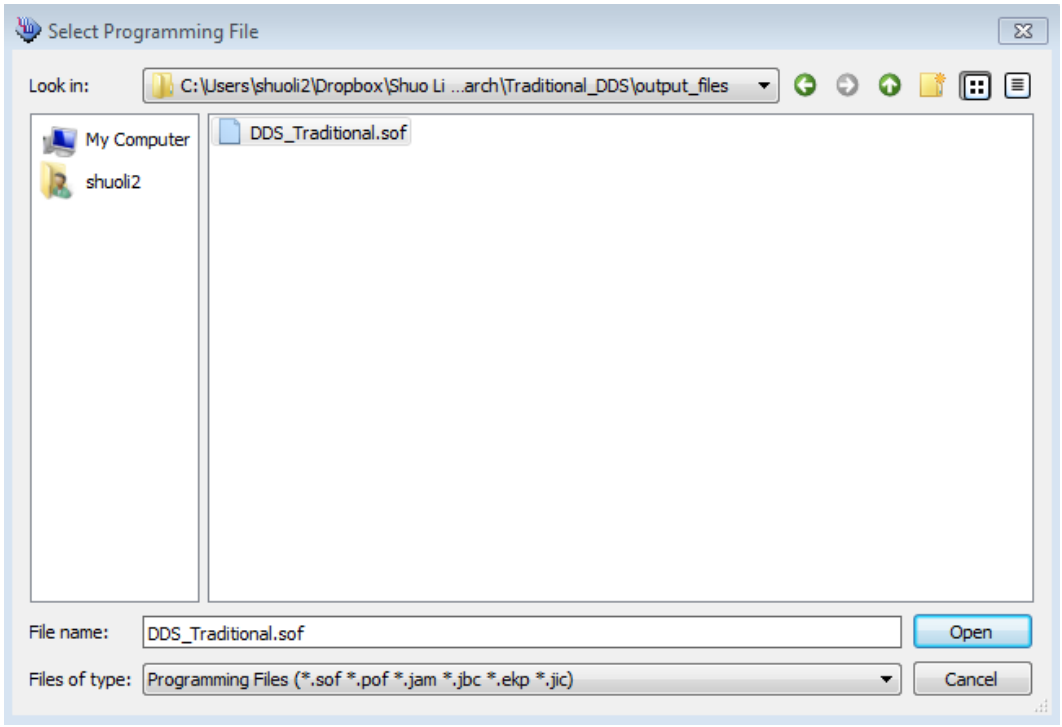

Figure 5.32: Adding [.sof] File

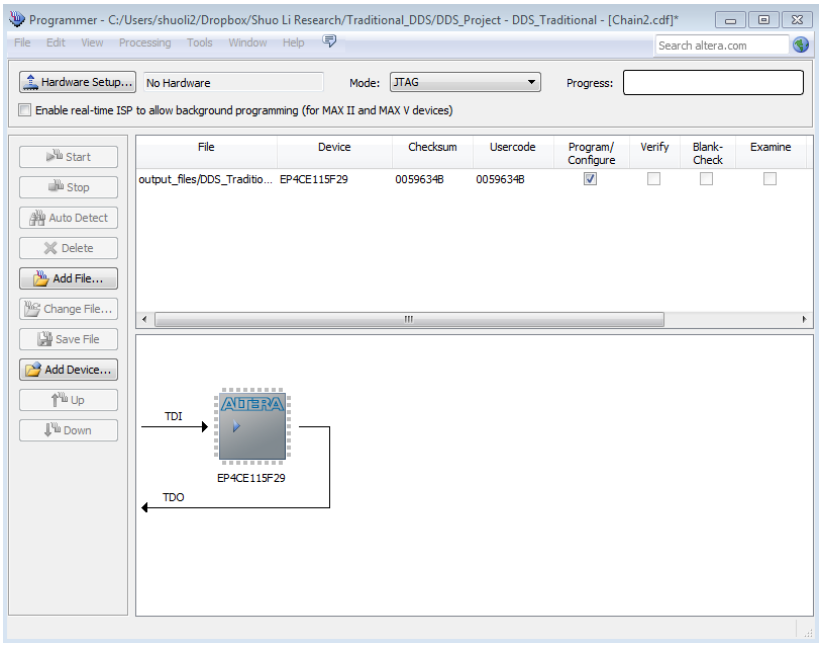

Figure 5.33: After Adding [.sof] File

Then, on the left corner of the programmer window, select **Hardware Setup**. In the hardware setup window, select **Hardware Settings**, then select **USB-Blaster [USB-0]** from the drop down list of **Currently selected hardware**. After that, click **Close**. The process is shown in Figure 5.34.

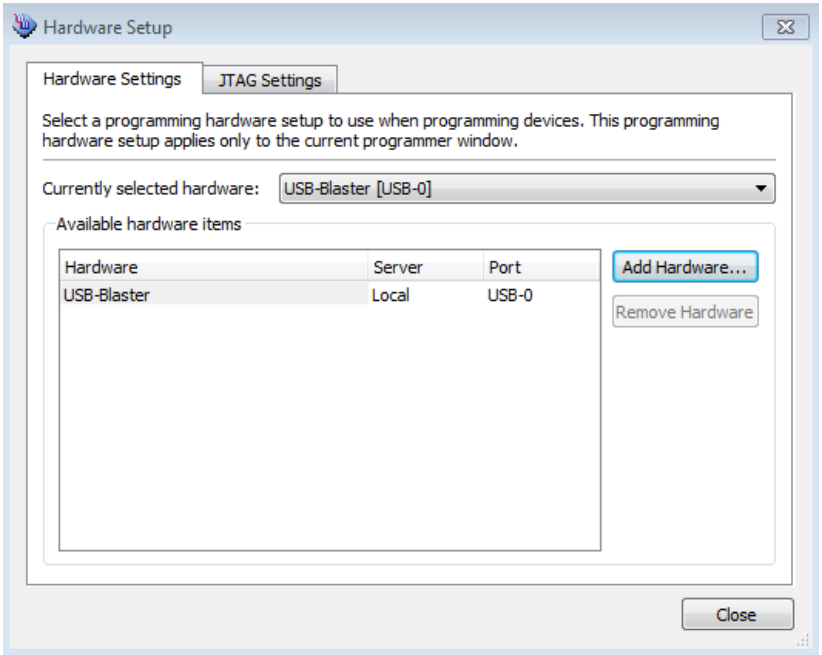

Figure 5.34: Hardware Setup

Now, click **Start** on the left of the programmer window to program the design onto the FPGA. A successful loading should be similar as shown in Figure 5.35. So far, the entire design is on the FPGA. As shown in Figure 5.36, the Hex Displays on FPGA are not flashing as usual after loading.

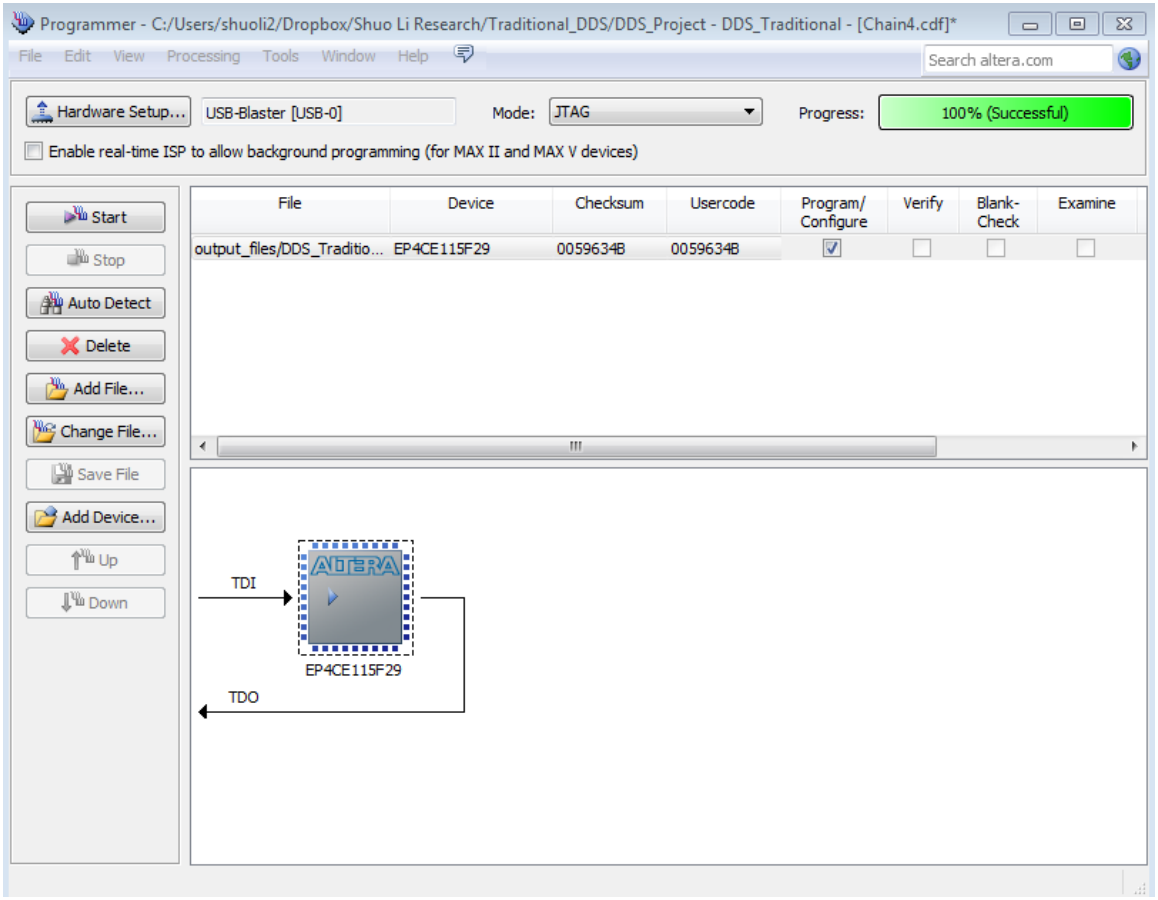

Figure 5.35: Successful Loading

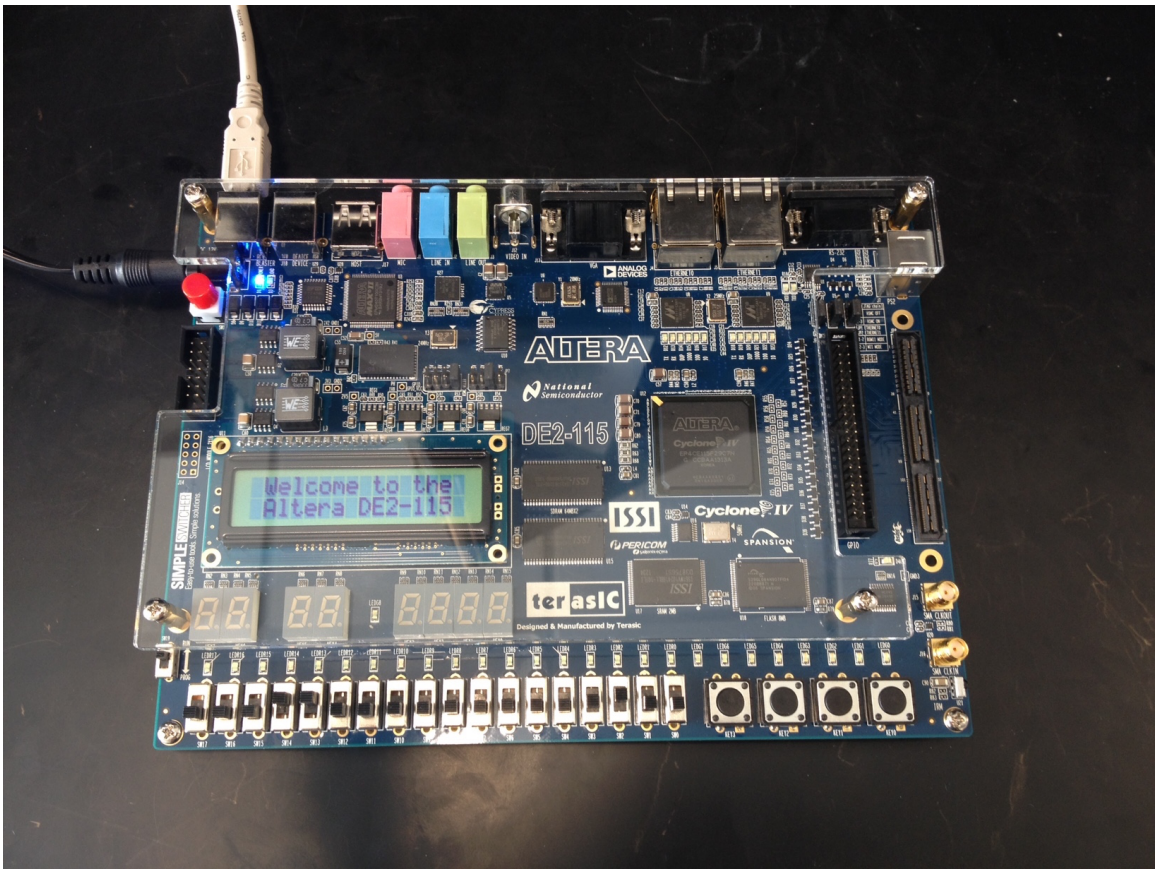

Figure 5.36: FPGA after Successful Loading

We can use the on board I/O, for example, switches to give FCW inputs and display the current phase word information or amplitude information on LEDs. This is a very straightforward and simple way to test the design.

5.5 Design Resources and Statistics

Resources Usage:

Altera FPGAs use the term Logic Element (LE) to describe a functional block that contains one LUT, one register, and some additional circuitry. To find out the usage of LUTs after compilation, go to **Compilation Report** tab. One the left, go to **Fitter ➔Resource Section ➔Resource Usage Summary** [13]. The summary report of both traditional and truncation spurs-free structures of DDS design should be displayed as Figure 5.37 and Figure 5.38.

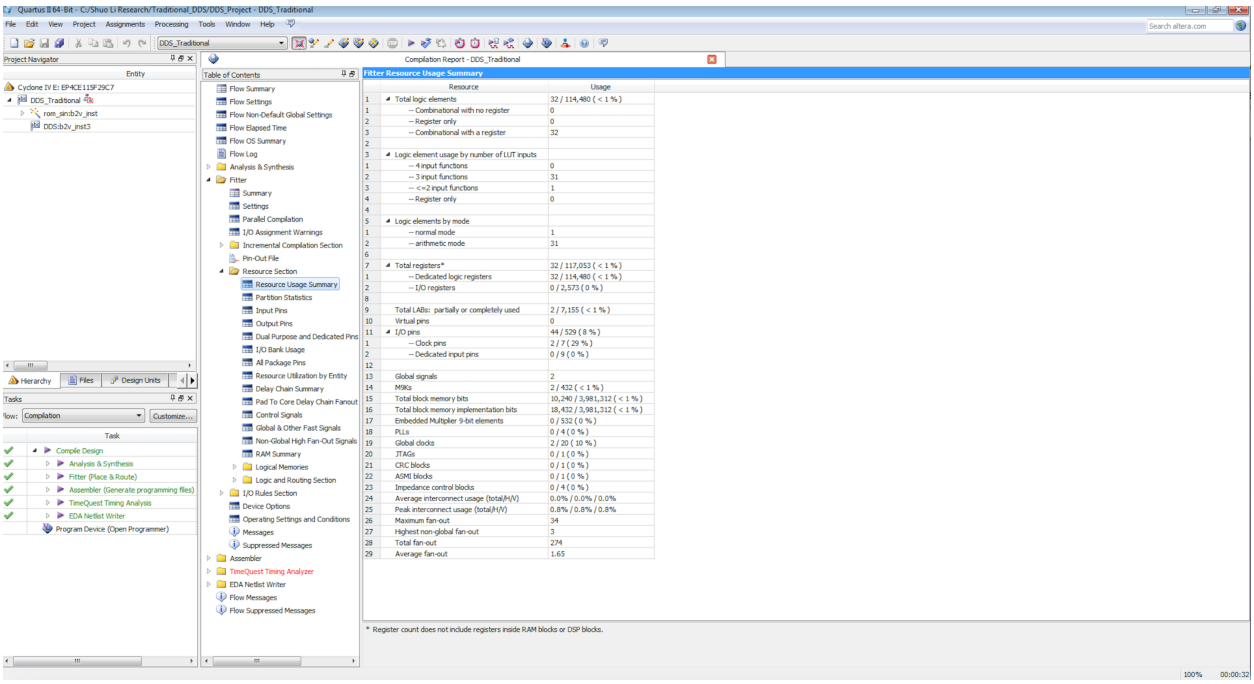

Figure 5.37: Fitter Resource Usage Summary for DDS with Traditional Structure

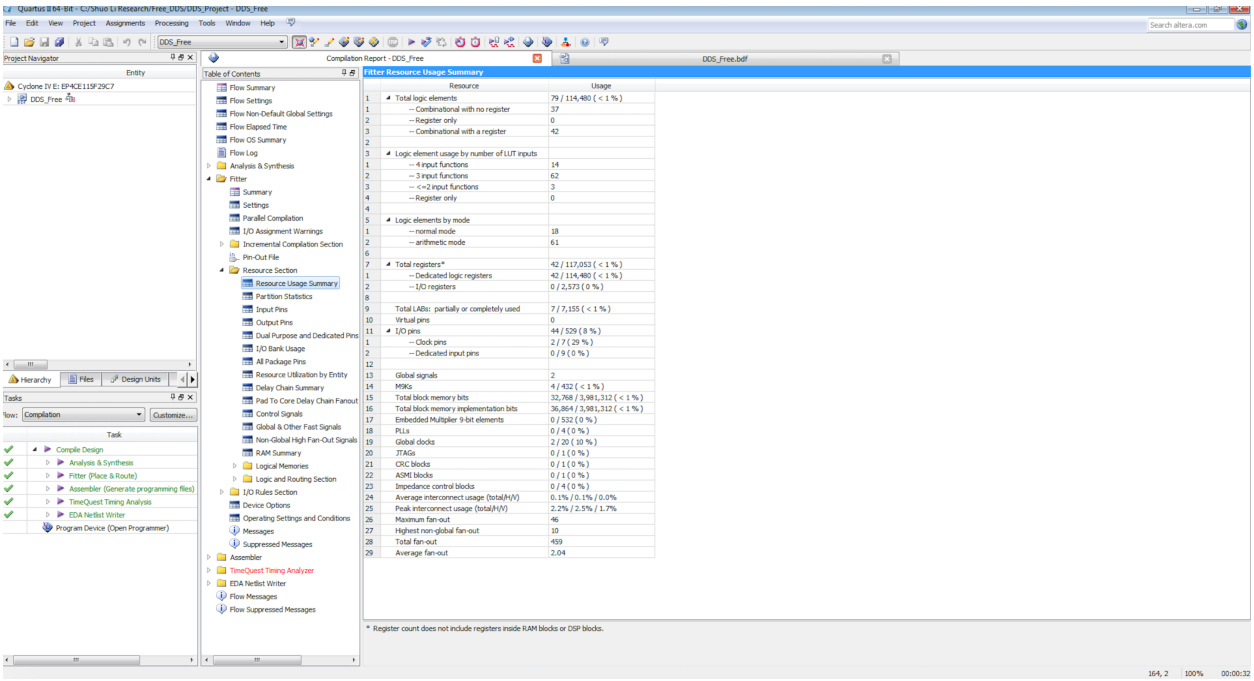

Figure 5.38: Fitter Resource Usage Summary for DDS with Truncation Spurs-Free Structure

In the Figure 5.37 and Figure 5.38, the total number of LEs is reported. The LEs are divided into 3 categories: combinational with no register, register only, and combinational with register. The total number of LUTs used should be the sum of the numbers of LEs that are combinational with no register and combinational with register.

Then, the numbers of LEs categorized by the number of LUT inputs are reported. The sum of the numbers of x input functions should be same as the number of LUTs.

After that, the total number of registers used is reported. The total number of registers is equal to the sum of the numbers of LEs that are combinational with register and register only.

Timing Analysis:

The **TimeQuest Timing Analyzer** is used for timing analysis in Quartus II.

The maximum allowed frequency for the system clock can be found under **TimeQuest Timing Analyzer ➔F_{max} Summary**

Power Consumption Analysis

The **PowerPlay Power Analyzer** shown in Figure 5.39 is used for power consumption analysis in Quartus II. To activate the PowerPlay Power Analyzer during compilation, go to **Assignments ➔Settings**. Under **Category**, you will find **PowerPlay Power Analyzer Settings**. Check the box of **Run PowerPlay Power Analyzer during compilation** and click **Apply ➔OK**. Then, after the compilation, the power analysis report can be found in **PowerPlay Power Analyzer ➔Summary** in the **Compilation Report tab** [13].

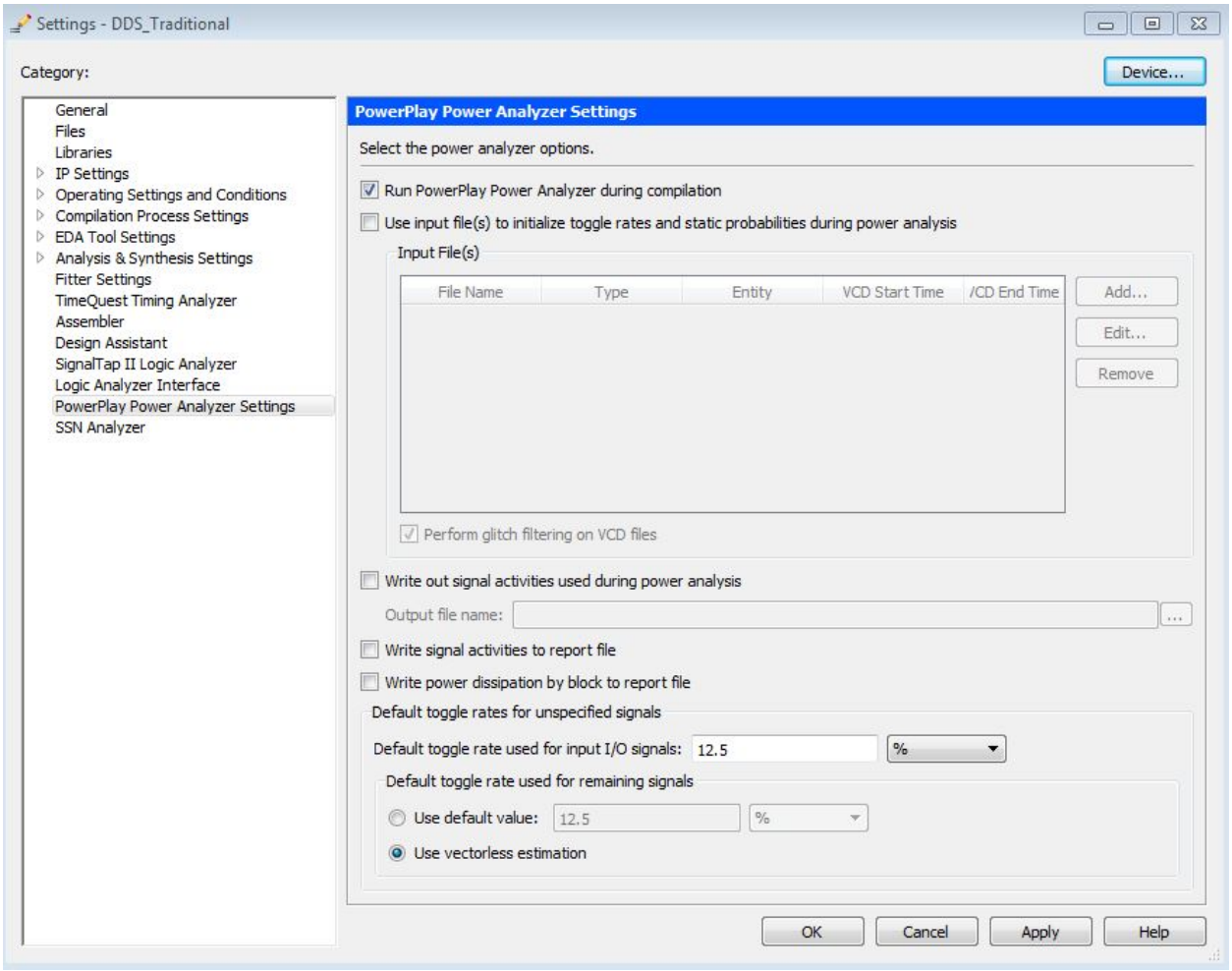

Figure 5.39: Activate PowerPlay Power Analyzer

5.6 Instructions on Writing Testbench

To simulate our design, we need to write a testbench.sv file (see Figure 5.40). For the DDS in this thesis, the testbench is fairly simple. It includes a clock generation block as well as an initial block to initialize all the input signals. A "#" sign means "delay" and also we need to specify the time unit and time precision for our simulation.

Generally, we include all the inputs and outputs in the testbench. In this design, we want to monitor some selected internal signals, so we can just manually add the output signals to the simulation waveforms.

Note: The time precision should be at least as precise as the time unit. For details of how to apply the testbench in ModelSim, please refer to **Appendix A**.

```
1   module testbench();
2
3   timeunit 1ns;
4   timeprecision 1ns;
5
6   reg clk, reset;
7
8   dds U0(.clk(clk), .reset(reset));
9
10  always begin : CLOCK_GENERATION
11  # 10 clk = ~clk;
12  end
13
14  initial begin
15  clk = 0;
16  reset = 0;
17  #2 reset = 1;
18  #5 reset = 0;
19  end
20  endmodule
```

Figure 5.40: The testbench.sv file for Simulating DDS

5.7 Behavioral/Gate Level Simulation in Altera-ModelSim

In this thesis, we will use ModelSim to simulate the DDS design. Please refer to **Appendix A** for tutorial on how to select and set up ModelSim as simulator in Altera Quartus II FPGA design software.

After adding testbench.sv into test bench and simulation files, we need to compile the entire design again. To start RTL simulation, as shown in Figure 5.41, select **Tools ➔ Run Simulation Tool ➔ RTL simulation** or simply click on the **RTL Simulation** button  in the tool bar.

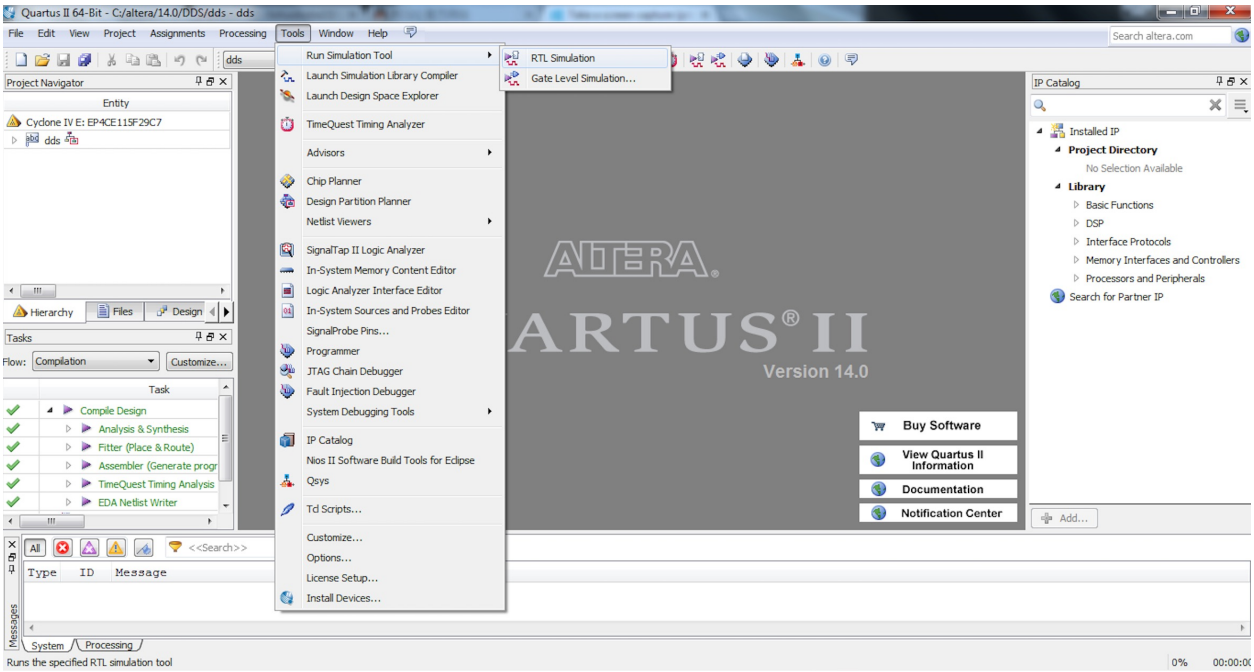

Figure 5. 41: Launch ModelSim

The pop-up console should be shown in **ModelSim-Altera** as Figure 5.42 if there are no compiling errors. Under testbench, select **U0**. In the sub-window **Objects**, **right click** any I/O signals or any internal signals you want to monitor, then select **Add Wave**.

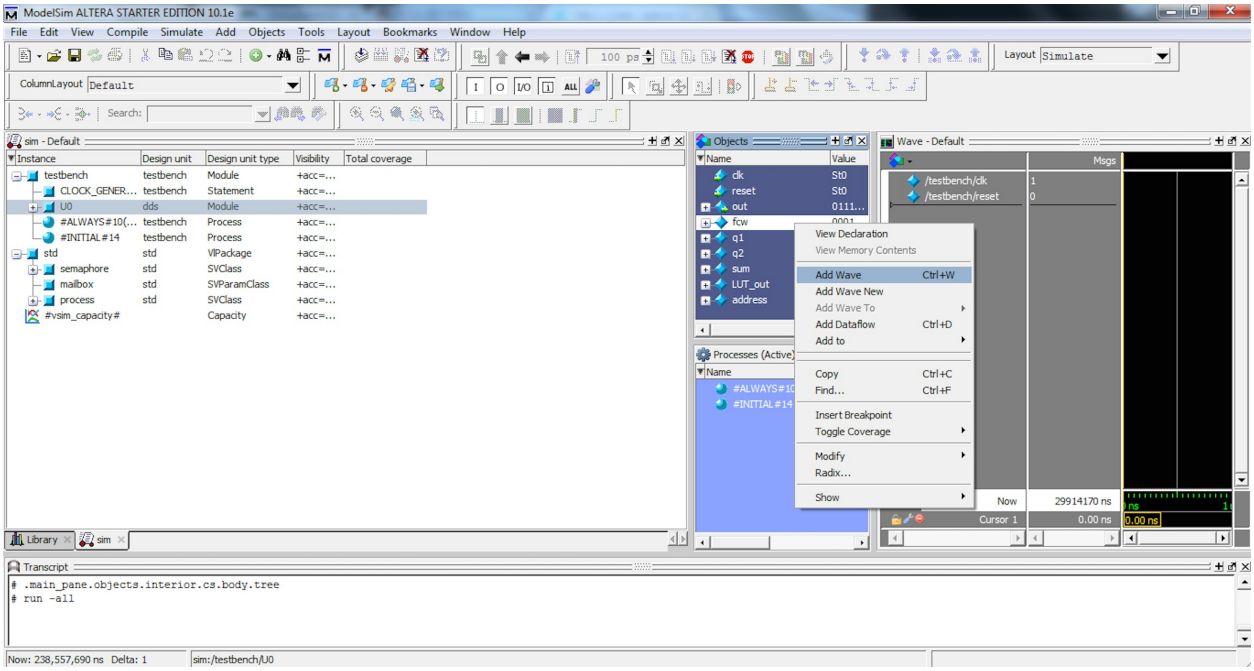

Figure 5.42: Add Wave to the Simulation Waveform

After adding all the signals you desire to select, go to the **command line** and type:

restart –f

This command will reset the waveform.

log –r *

This command will tell ModelSim to record all signals in the circuit recursively.

run 10000 ns

This command will run the testbench for 10000 ns.

Undock the simulation waveforms, then **Zoom Full** for more convenient reading. Right clicking on the waveform will zoom the display to a proper scale as shown in Figure 5.43 [13].

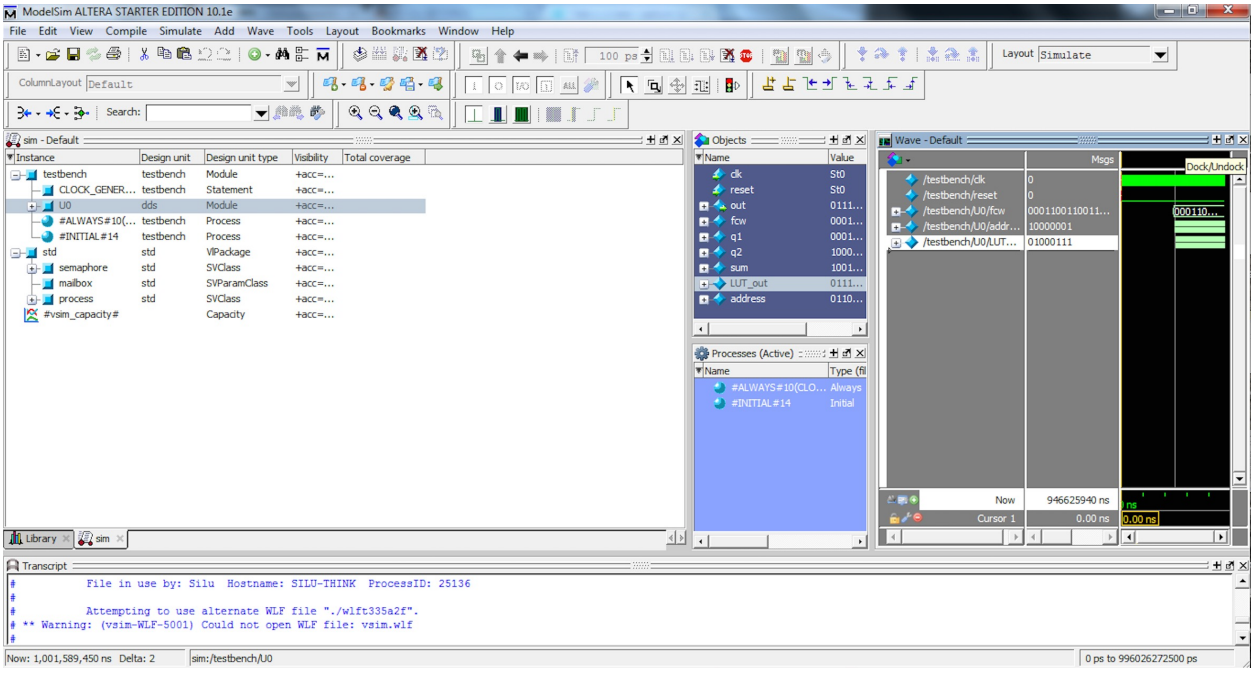

Figure 5.43: Getting Waveforms for Convenient Reading

Important Note [13]:

The internal compiler of ModelSim is slightly different from that of Quartus II, so a successful compilation in Quartus II does not guarantee successful compilation in ModelSim. If part of the design is not showing up in the design hierarchy, that is probably the reason. The designers will need to look through the error messages in the **Transcripts** and correct them to continue.

Also, simulations assume there are no delays in the circuits, which is obviously not the case in the real world. Designers need to take the timing issues into consideration when they design the circuits. If the result shown on the FPGA board is different from what is shown in simulation, that is probably the reason.

For Gate Level Simulation, follow a procedure similar to that used by RTL Simulation to launch ModelSim. ModelSim has an important feature of viewing signals as an analog waveform; that is the reason we don't have to implement an actual DAC and LPF to monitor the signal output.

Chapter 6. DDS MEASUREMENTS

6.1 Measurements of DDS with Traditional Structure

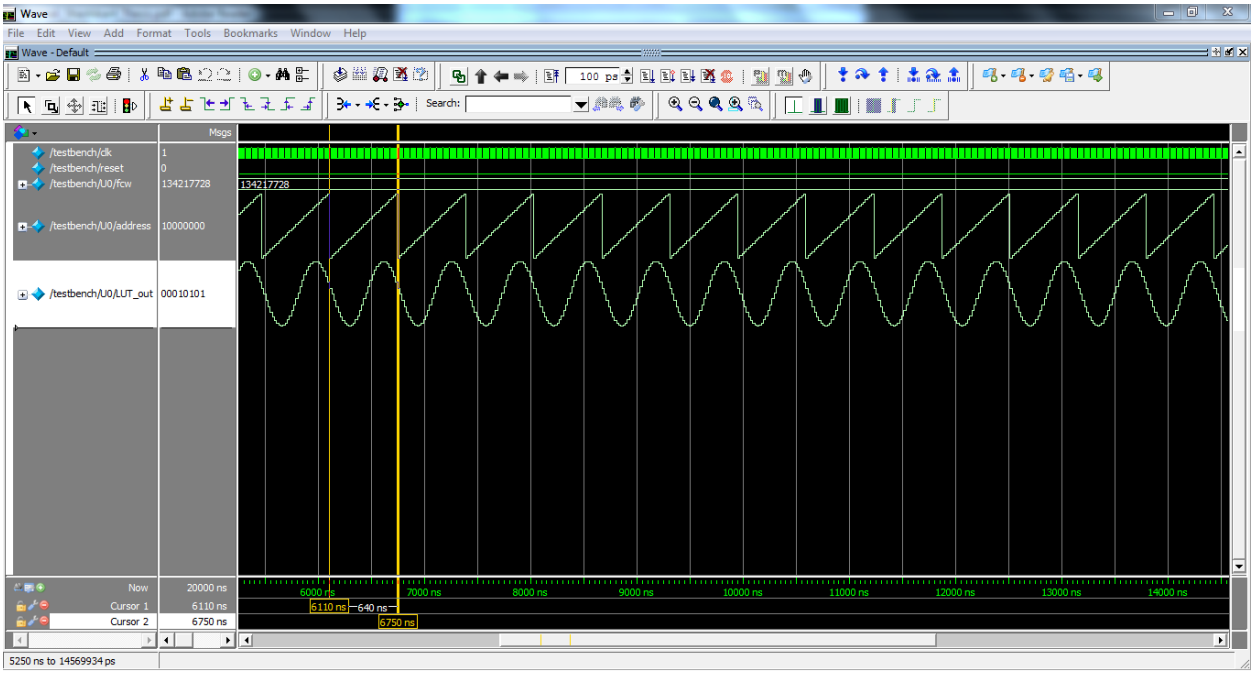

Figure 6.1: 32-bit 1.6MHz Traditional DDS Behavioral Simulation

The clock frequency is 50MHz due to the maximum on-board frequency on FPGA, so the clock cycle should be 20ns.

In Figure 6.1, given decimal number 134217728 (00001000000000000000000000000000 in unsigned binary) as the FCW, we can calculate the output frequency to be 1.6MHz based on Equation 2.7. From the distance between two cursors, we can see that one full cycle of sine wave is 640ns, so the output frequency matches what we should have in theory.

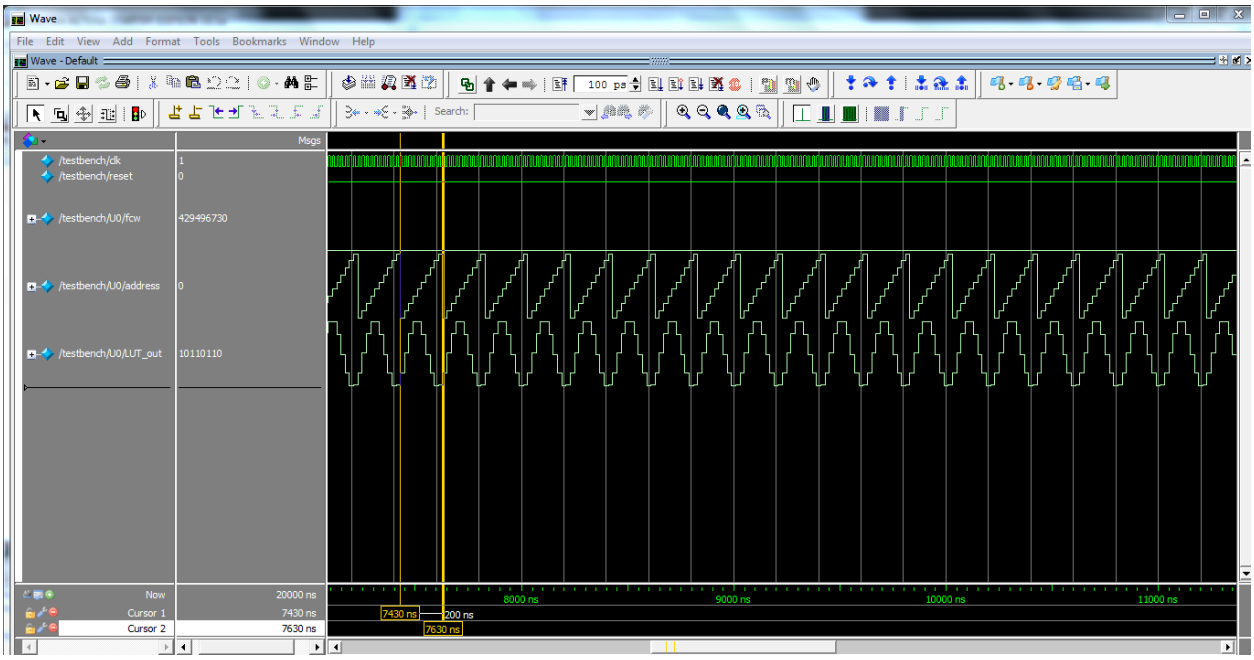

Figure 6.2: 32-bit 5MHz Traditional DDS Behavioral Simulation

In Figure 6.2, from the distance between the cursors, we can see one full clock cycle is 200ns, so the output frequency is 5MHz. As the frequency grows faster, we can see it takes fewer samples in each cycle, so the accuracy is not as good as 1.6MHz one.

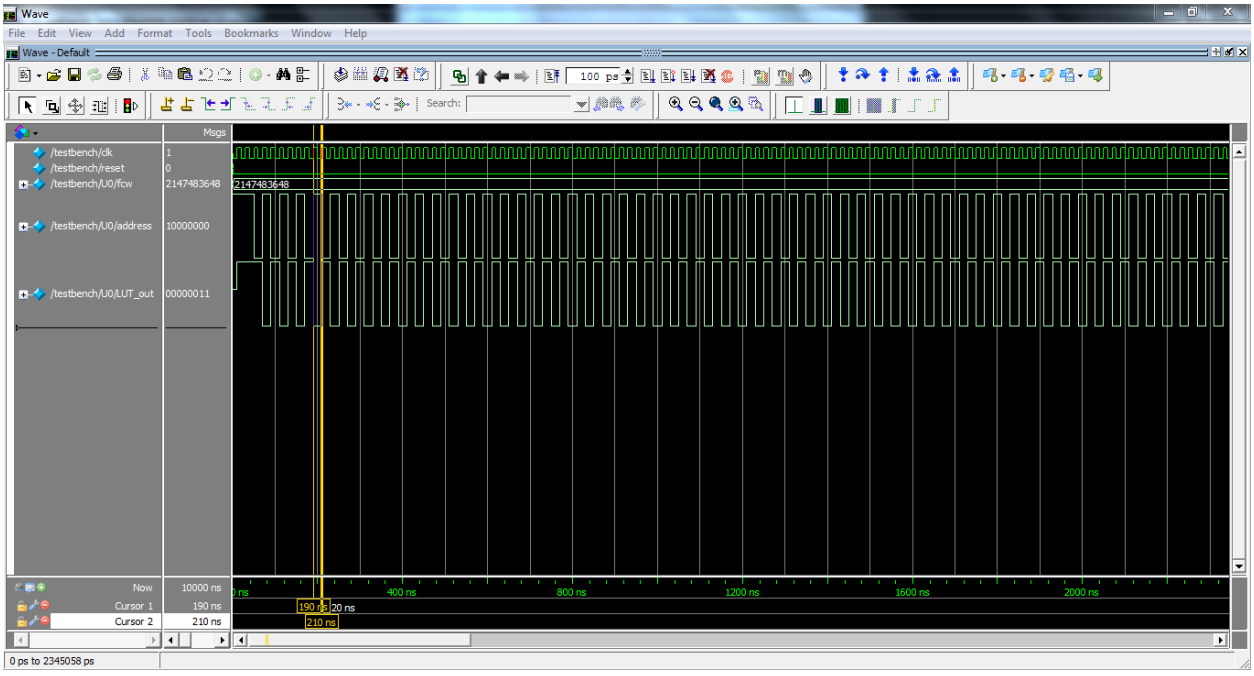

Figure 6.3: 32-bit 25MHz Traditional DDS Behavioral Simulation

In the Figure 6.3, we cannot see the sine wave anymore, because the sampling process has to obey the Nyquist theorem as we mentioned in Chapter 2.

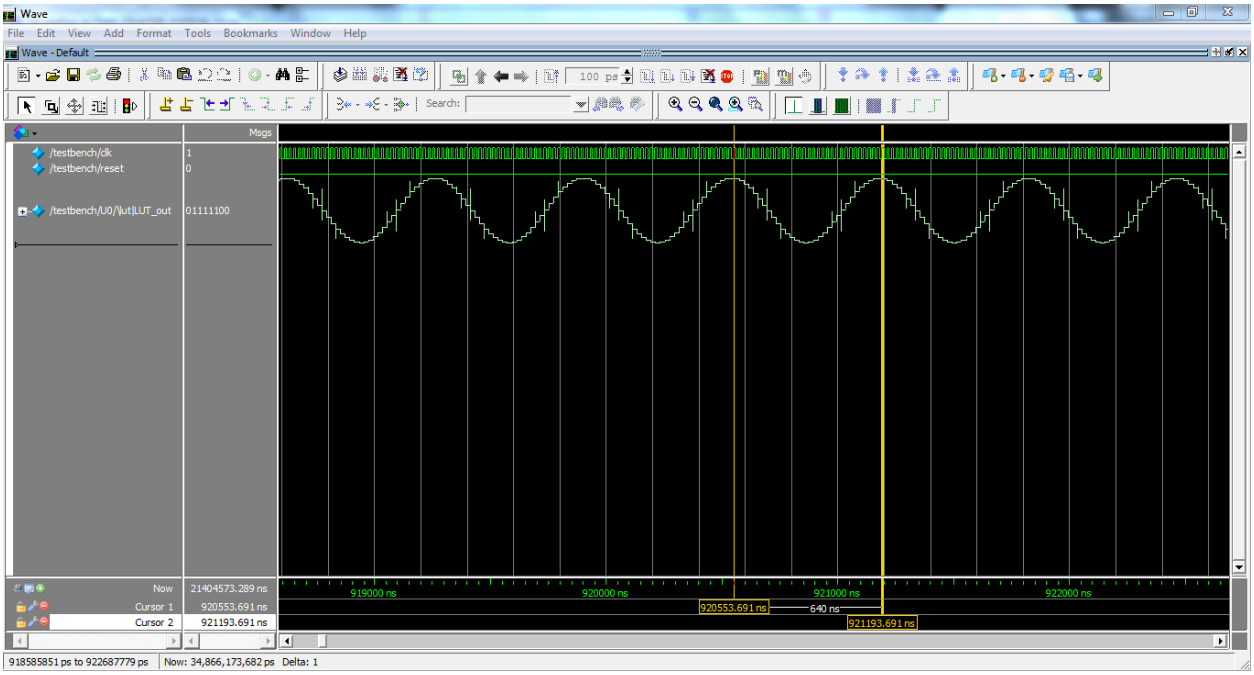

Figure 6.4: 32-bit 1.6MHz Traditional DDS Gate-Level Simulation

In the Figure 6.4, post map & route simulation, we can see the impurities in the output, which are the truncation spurs. In the Figure 6.5, the distortion will get worse with a higher frequency.

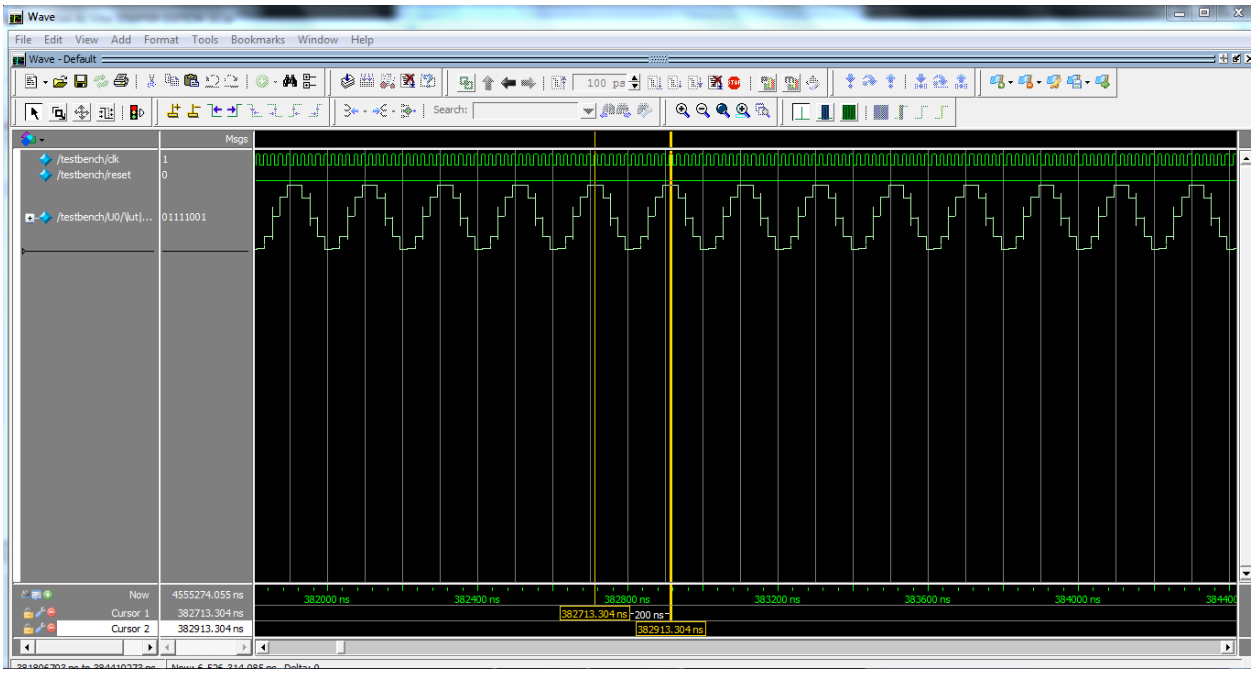

Figure 6.5: 32-bit 5MHz Traditional Spurs Gate-Level Simulation

6.2 Measurements of DDS with Truncation Spurs-Free Structure

Figure 6.6 and Figure 6.7 verify that the DDS with Truncation Spurs-Free Structure has the same functionality as the traditional one.

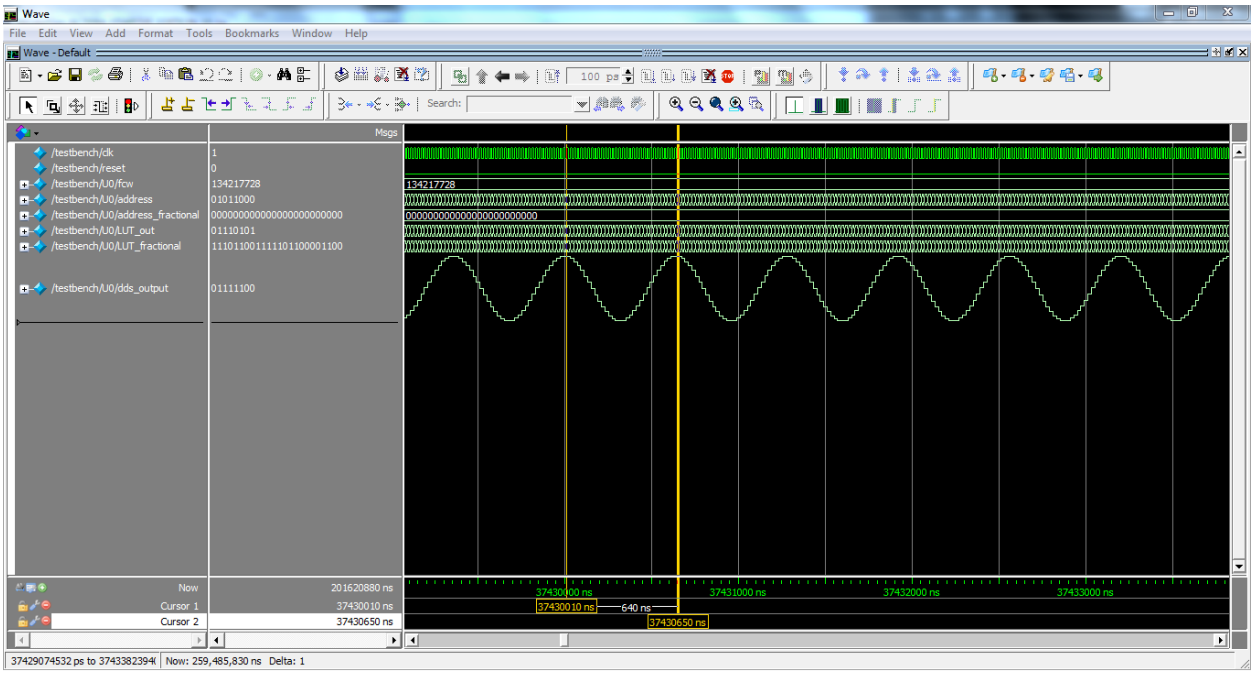

Figure 6.6: 32-bit 1.6MHz Truncation Spurs-Free DDS Behavioral Simulation

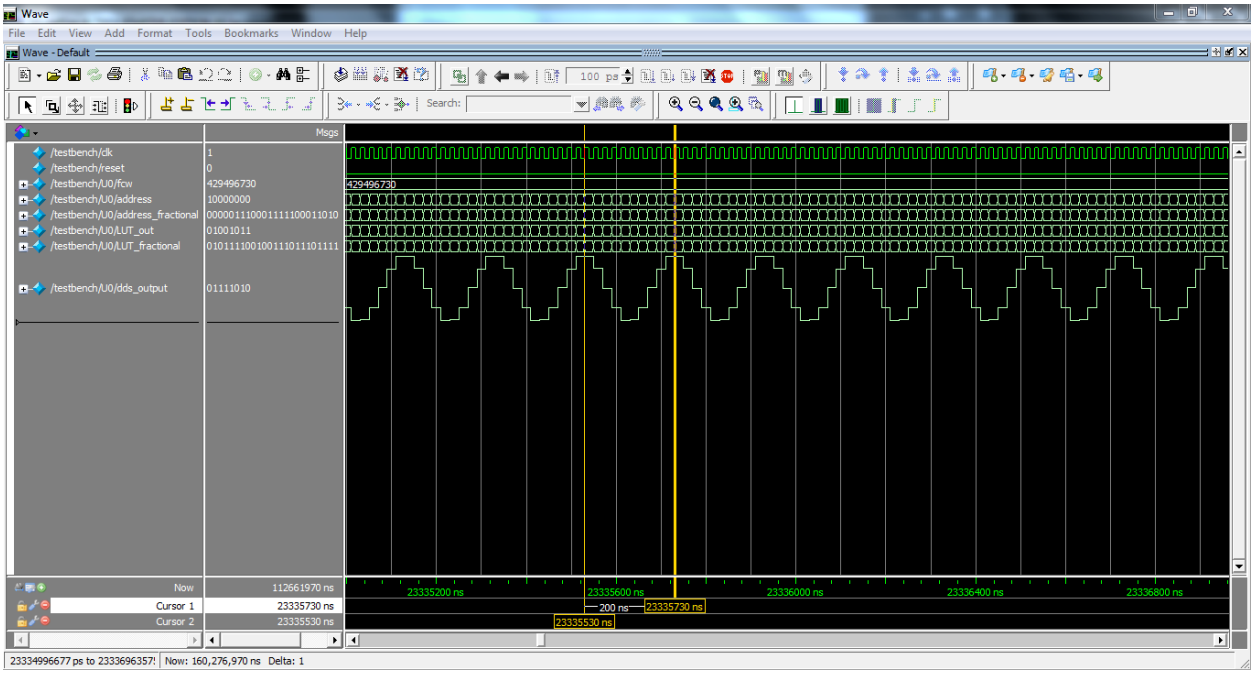

Figure 6.7: 32-bit 5MHz Truncation Spurs-Free DDS Behavioral Simulation

From Figure 6.8 and Figure 6.9, we compare the post map & route simulation result for signals LUT_out and dds_out. LUT_out should be the same as the final output of traditional DDS and dds_out is the final output of the truncation spurs-free DDS. We can see clearly that the truncation spurs are eliminated by the new structure.

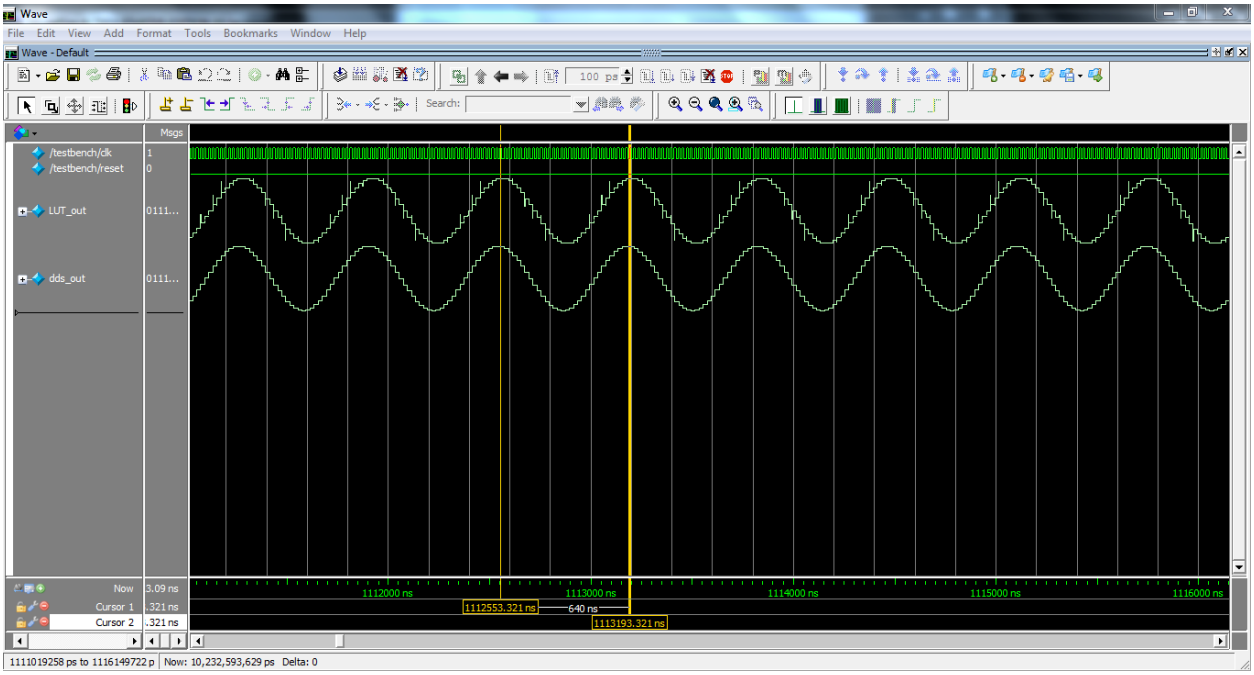

Figure 6.8: 32-bit 1.6MHz Truncation Spurs-Free DDS Gate-Level Simulation

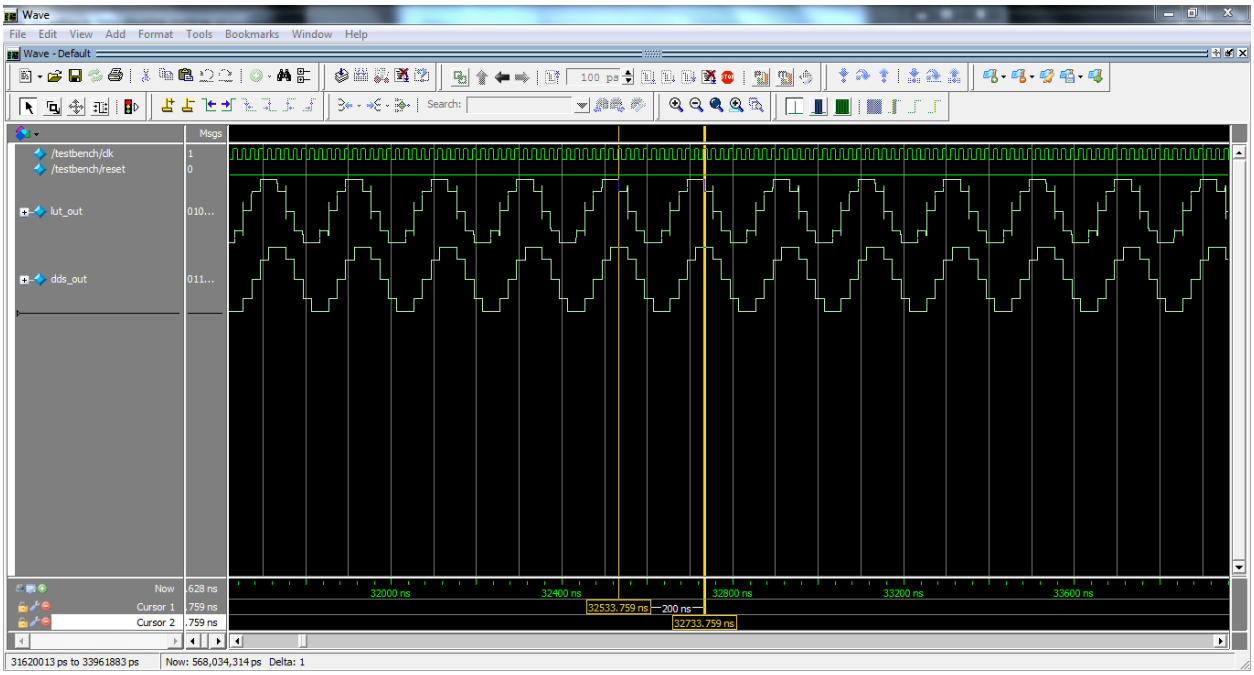

Figure 6.9: 32-bit 5MHz Truncation Spurs-Free DDS Gate-Level Simulation

Chapter 7. CONCLUSION

7.1 Summary

This thesis serves as a comprehensive tutorial on designing two different structures of DDS on FPGA (traditional structure and truncation spurs-free structure). It starts by introducing the background of both DDS and providing several design examples in Verilog. Then, the thesis addresses the essential phases of FPGA design flow and gives a detailed step by step instruction on designing digital circuits on Altera DE2-115 FPGA with Altera Quartus II FPGA development software. In the end, the thesis compares the simulation results of both structures of DDS and verifies the improvement of the output signal quality.

7.2 Future Work

1. Since this thesis work is a preliminary research project to verify the functionality of a new structure of DDS, the digital design uses an FPGA design approach. However, the resources limitation and maximum on-board frequency of FPGA will limit the maximum frequency of the DDS. Future work could use an ASIC approach to design the DDS and fabricate the real circuit for testing.

2. Future research could focus on eliminating quantization noise spurs by creating a new structure of DDS and building it on FPGA for verification.

3. As mentioned in this thesis, DDS has many advantages over the analog counterpart. Future research topics related to involving DDS into a more complex circuit design are also worth exploring, for example, serving as the local oscillator of a digital PLL.

4. Pipelined structure for the PA and segmented DAC are also desired to accelerate the speed of the DDS. For our design, it is not necessary because the speed is mainly limited by the maximum on-board clock frequency of FPGA; however, if further research is conducted with an ASIC design approach, especially for DDS MMIC design, hardware architecture should be considered carefully. For details please refer to [21].

Appendix A: SETTING UP MODELSIM

In Quartus II, select **Assignments ➔Settings**; the settings window should display as shown in Figure A.1. On the left of the window, click on **EDA Tool Settings** and set the simulation tool to **ModelSim-Altera**; the format is **Verilog HDL**.

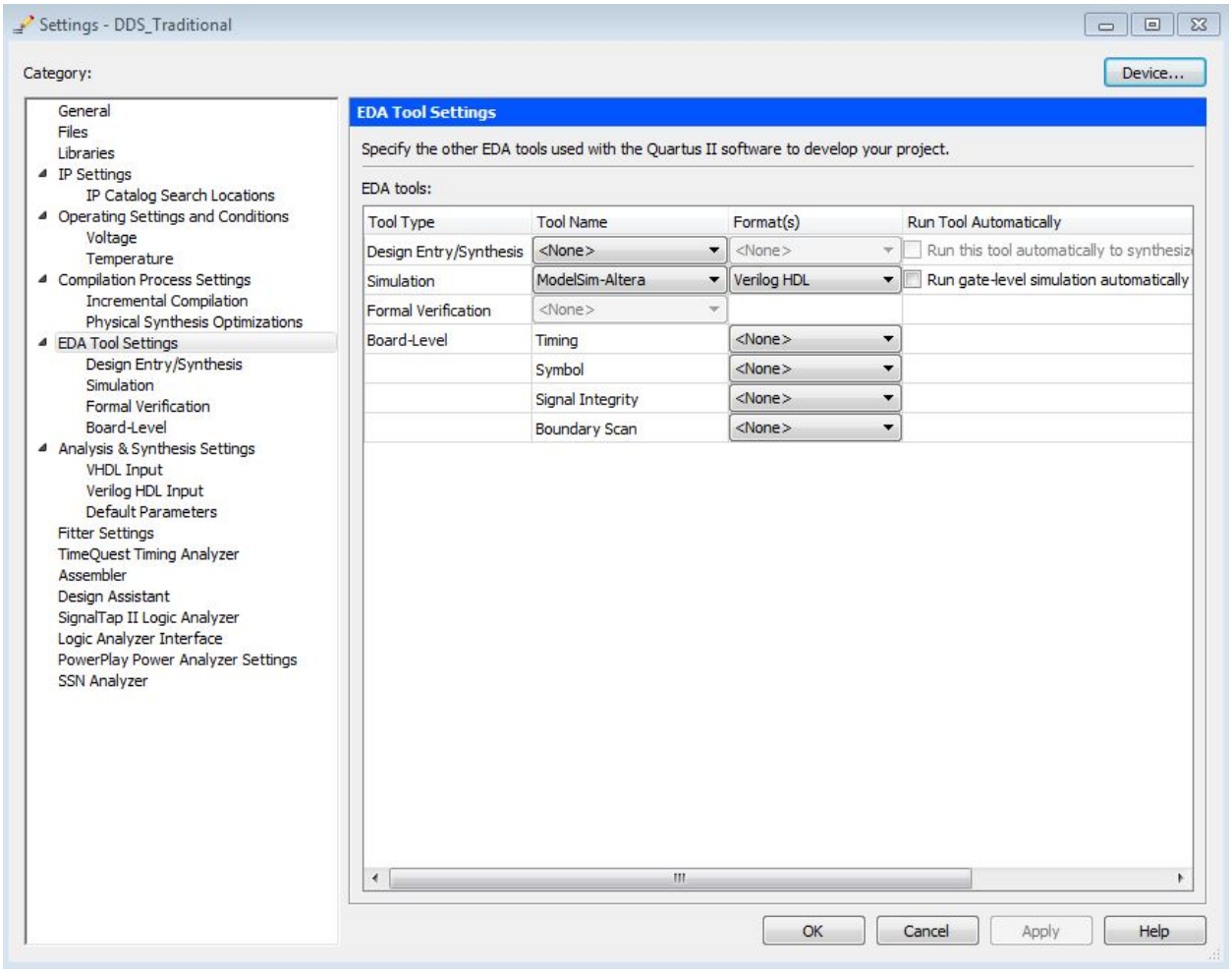

Figure A.1: EDA Tool Settings

Next, select **Simulation** under **EDA Tool Settings**. Choose **ModelSim-Altera** as the **Tool name**, **Verilog HDL** as the **Format for output netlist** and "**simulation/modelsim**" as **Output directory**. The window should display as Figure A.2.

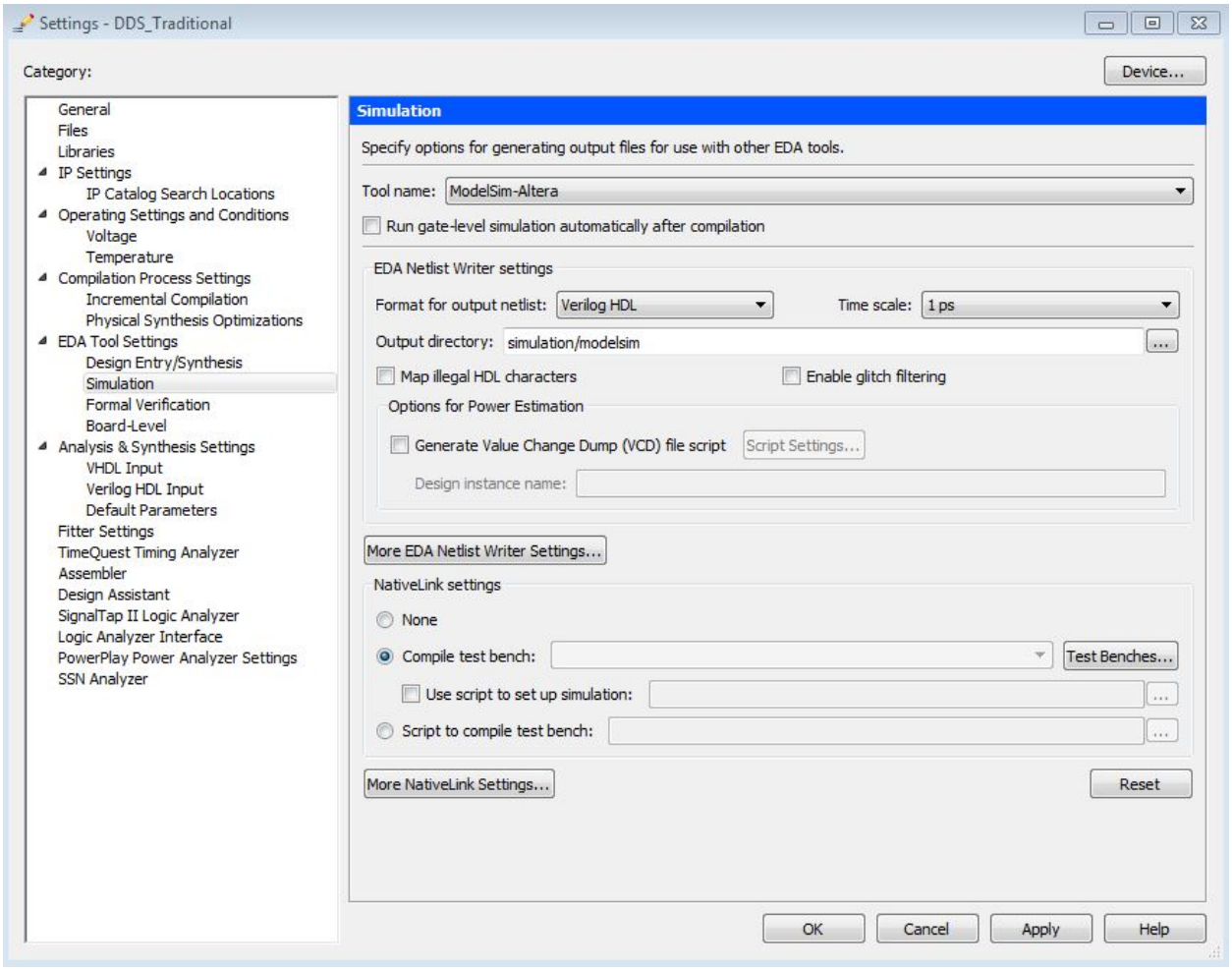

Figure A.2: Simulation

At the bottom of Figure A.2, there is a **NativeLink settings** section. Under this section, select **Compile test bench**, then click on **Test Benches**. A Test Benches window should pop up. Click on **New**, then a window shown as Figure A.3 should pop up. Enter **testbench** for **Test bench name** and **Top level module** in test bench, 1000 ns as the simulation end time, and then click on the **…** button to add **testbench.v** into test bench and simulation files [13]. After that, click **OK**.

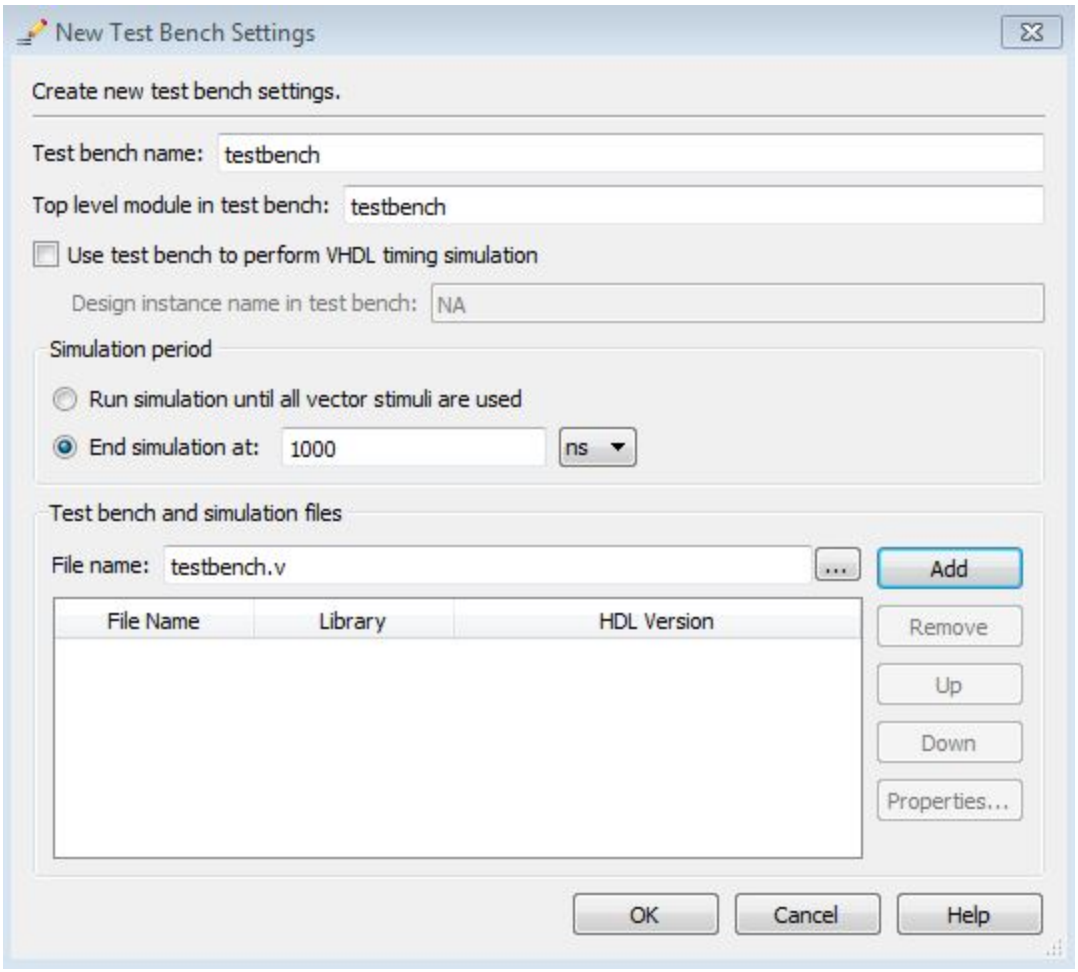

Figure A.3: New Test Bench Settings

So far, we finish setting up ModelSim as our simulator and are ready to run the testbench for simulation.

Appendix B: VERILOG MODULES

Besides the modules included in the body of this thesis, the Verilog codes for other primary modules are provided here.

/*Truncator*/

```
module phase_truncator (clk, reset, pa_out, address, address_fractional);

input clk, reset;
input [31:0] pa_out;
output [7:0] address;
output [23:0] address_fractional;

reg [7:0] address;
reg [23:0] address_fractional;

always @(posedge clk or posedge reset)
        if (reset == 1'b1)
                begin
                        address <= 8'b0;
                        address_fractional <= 24'b0;
                end
        else
                begin
                        address <= pa_out[31:24];
                        address_fractional <= pa_out[23:0];
                end
endmodule
```

/*Register*/

```
module register(clk, reset, d, q);

input clk, reset;
input [31:0] d;
output [31:0] q;

reg [31:0] q;

always @ (posedge clk or posedge reset)
begin
        if (reset == 1'b1)
                q <= 32'd0;
        else
                q <= d;
end
endmodule
```

```
/*LUT*/
module look_up_table(clk, reset, address, LUT_fractional, LUT_out);

input clk, reset;
input [7:0] address;
output [23:0] LUT_fractional;
output [7:0] LUT_out;

reg [7:0] LUT_out;
reg [23:0] LUT_fractional;
reg [31:0] LUT [255:0];

always@(posedge clk or posedge reset)
begin
if(reset == 1'b1)
begin
LUT[0]<=32'b0;
LUT[1]<=32'b00000011001001101110100100100110;
LUT[2]<=32'b00000110010011010101010100000101;
LUT[3]<=32'b00001001011100101100011001100111;
LUT[4]<=32'b00001100100101101100000001000001;
LUT[5]<=32'b00001111101110001100010110111101;
LUT[6]<=32'b00010010110110000101101001011000;
LUT[7]<=32'b00010101111101010000000111101101;
LUT[8]<=32'b00011001000011100100000011001011;
LUT[9]<=32'b00011100001000111001101111001010;
LUT[10]<=32'b00011111001101001001100001011100;
LUT[11]<=32'b00100010010000001011110010100010;
LUT[12]<=32'b00100101010001111000111101111010;
LUT[13]<=32'b00101000010010001001100010011000;
LUT[14]<=32'b00101011010000110110000010010111;
LUT[15]<=32'b00101110001101110111000100001001;
LUT[16]<=32'b00110001001001000101010010001010;
LUT[17]<=32'b00110100000010011001011011010101;
LUT[18]<=32'b00110110111001101100010011010101;
LUT[19]<=32'b00111001101110110110110010110011;
LUT[20]<=32'b00111100100001110001110111101101;
LUT[21]<=32'b00111111010010010110100101100101;
LUT[22]<=32'b01000010000000011110000101110011;
LUT[23]<=32'b01000100101100000001100111110101;
LUT[24]<=32'b01000111010100111010100001011111;
LUT[25]<=32'b01001001111011000010001111001111;
LUT[26]<=32'b01001100011110010010010100011010;
LUT[27]<=32'b01001110111110100100011011011101;
LUT[28]<=32'b01010001011011110010010110001111;
LUT[29]<=32'b01010011110101110101111110001011;
```

```
LUT[30]<=32'b01010110001100101001010100100101;
LUT[31]<=32'b01011000100000000110100010110110;
LUT[32]<=32'b01011010110000000111111010101010;
LUT[33]<=32'b01011100111100100111110110010000;
LUT[34]<=32'b01011111000101100000111000100110;
LUT[35]<=32'b01100001001010101101101101101001;
LUT[36]<=32'b01100011001100001001001010100001;
LUT[37]<=32'b01100101001001101110001101101011;
LUT[38]<=32'b01100111000011010111111111001010;
LUT[39]<=32'b01101000111001000001110000110010;
LUT[40]<=32'b01101010101010100110111110010010;
LUT[41]<=32'b01101100011000000011001101011111;
LUT[42]<=32'b01101110000001010010001110100011;
LUT[43]<=32'b01101111100110001111111100000010;
LUT[44]<=32'b01110001000110111000011011001000;
LUT[45]<=32'b01110010100011000111111011110011;
LUT[46]<=32'b01110011111010111010111000111001;
LUT[47]<=32'b01110101001110001101111000010100;
LUT[48]<=32'b01110110011100111101101011001001;
LUT[49]<=32'b01110111100111000111001101110001;
LUT[50]<=32'b01111000101100100111100111111110;
LUT[51]<=32'b01111001101101011100001101000111;
LUT[52]<=32'b01111010101001100010011100001001;
LUT[53]<=32'b01111011100000110111111111110010;
LUT[54]<=32'b01111100010011011010101110100101;
LUT[55]<=32'b01111101000001001000101010111101;
LUT[56]<=32'b01111101101010000000000011010111;
LUT[57]<=32'b01111110001101111111010010010001;
LUT[58]<=32'b01111110101101000100111110010001;
LUT[59]<=32'b01111111000111001111111010001010;
LUT[60]<=32'b01111111011100011111000100111010;
LUT[61]<=32'b01111111101100110001101001110001;
LUT[62]<=32'b01111111111000000011000000010001;
LUT[63]<=32'b01111111111110011110101100010000;
LUT[64]<=32'b01111111111111111000011101111001;
LUT[65]<=32'b01111111111100010100010001101110;
LUT[66]<=32'b01111111110011110010010000100101;
LUT[67]<=32'b01111111100110010010101111101011;
LUT[68]<=32'b01111111010011110110010000100000;
LUT[69]<=32'b01111110111100011101100000111001;
LUT[70]<=32'b01111110100000001001011010111101;
LUT[71]<=32'b01111101111110111011000101000001;
LUT[72]<=32'b01111101011000110011110001100110;
LUT[73]<=32'b01111100101101110100111111011001;
LUT[74]<=32'b01111011111110000000011001001011;
LUT[75]<=32'b01111011001001010111110101101111;
```

```
LUT[76]<=32'b01111010001111111101010111110101;
LUT[77]<=32'b01111001010001110011001110000100;
LUT[78]<=32'b01111000001110111011110010111000;
LUT[79]<=32'b01110111000111011001101100010110;
LUT[80]<=32'b01110101111011001111101100001100;
LUT[81]<=32'b01110100101010100000101111100101;
LUT[82]<=32'b01110011010101001111111111000100;
LUT[83]<=32'b01110001111011100000101110011110;
LUT[84]<=32'b01110000011101010110011100101100;
LUT[85]<=32'b01101110111010110100110011101010;
LUT[86]<=32'b01101101010011111111101000000111;
LUT[87]<=32'b01101011101000111010111001100000;
LUT[88]<=32'b01101001111001101010110001110101;
LUT[89]<=32'b01101000000110010011100101011011;
LUT[90]<=32'b01100110001110111001110010111010;
LUT[91]<=32'b01100100010011100010000010110111;
LUT[92]<=32'b01100010010100010001000111110001;
LUT[93]<=32'b01100000010001001011111101110001;
LUT[94]<=32'b01011110001010010111101010011111;
LUT[95]<=32'b01011011111111111001011100110101;
LUT[96]<=32'b01011001110001110110101100110010;
LUT[97]<=32'b01010111100000010100111011001100;
LUT[98]<=32'b01010101001011011001110001100110;
LUT[99]<=32'b01010010110011001011000001111010;
LUT[100]<=32'b01010000010111101110100110010100;
LUT[101]<=32'b01001101111001001010100000111101;
LUT[102]<=32'b01001011010111100100111011101111;
LUT[103]<=32'b01001000110011000100001000000010;
LUT[104]<=32'b01000110001011101110011110100011;
LUT[105]<=32'b01000011100001101010011110111110;
LUT[106]<=32'b01000000110100111110101111110000;
LUT[107]<=32'b00111110000101110001111101110111;
LUT[108]<=32'b00111011010100001010111100100001;
LUT[109]<=32'b00111000100000010000100100111100;
LUT[110]<=32'b00110101101010001001110110000100;
LUT[111]<=32'b00110010110001111101110100001111;
LUT[112]<=32'b00101111110111110011101001000001;
LUT[113]<=32'b00101100111011110010100010111000;
LUT[114]<=32'b00101001111110000001110100110110;
LUT[115]<=32'b00100110111110101000110110010101;
LUT[116]<=32'b00100011111101101111000010110010;
LUT[117]<=32'b00100000111011011011111001011000;
LUT[118]<=32'b00011101110111111011011100110011;
LUT[119]<=32'b00011010110011000111110010111001;
LUT[120]<=32'b00010111101101010110000100011000;
LUT[121]<=32'b00010100100110101001011100100011;
```

```
LUT[122]<=32'b00010001011111001001101001000001;
LUT[123]<=32'b00001110010110111110011001010110;
LUT[124]<=32'b00001011001110001111011110110100;
LUT[125]<=32'b00001000000101000100101100000100;
LUT[126]<=32'b00000100111011100101110100110101;
LUT[127]<=32'b00000001110001111010101101100111;
LUT[128]<=32'b11111110101000001011001011011011;
LUT[129]<=32'b11111011011110011111000011011010;
LUT[130]<=32'b11111000010100111110001010100101;
LUT[131]<=32'b11110101001011110000010101100010;
LUT[132]<=32'b11110010000010111101011000001000;
LUT[133]<=32'b11101110111010101101000101001010;
LUT[134]<=32'b11101011110011000111001110000110;
LUT[135]<=32'b11101000101100010011100010110000;
LUT[136]<=32'b11100101100110011001110001000000;
LUT[137]<=32'b11100010100001100001100100011101;
LUT[138]<=32'b11011111011101110010100110001011;
LUT[139]<=32'b11011100011011010100011100011011;
LUT[140]<=32'b11011001011010001110101010010001;
LUT[141]<=32'b11010110011010101000101111011000;
LUT[142]<=32'b11010011011100101010000111101100;
LUT[143]<=32'b11010000100000011010001011001010;
LUT[144]<=32'b11001101100110000000001101011011;
LUT[145]<=32'b11001010101101100011011101100001;
LUT[146]<=32'b11000111110111001011000101101010;
LUT[147]<=32'b11000101000010111110001010111001;
LUT[148]<=32'b11000010010001000011101100111001;
LUT[149]<=32'b10111111100001100010100101100110;
LUT[150]<=32'b10111100110100100001101001000001;
LUT[151]<=32'b10111010001010000111100100111110;
LUT[152]<=32'b10110111100010011011000000110000;
LUT[153]<=32'b10110100111101100010011100111100;
LUT[154]<=32'b10110010011011100100010011001001;
LUT[155]<=32'b10101111111100100110110101101110;
LUT[156]<=32'b10101101100000110000001111100011;
LUT[157]<=32'b10101011001000000110100011110010;
LUT[158]<=32'b10101000110010101111101101101001;
LUT[159]<=32'b10100110100000110001100000001010;
LUT[160]<=32'b10100100010010010001100101111011;
LUT[161]<=32'b10100010000111010101100000111100;
LUT[162]<=32'b10100000000000000010101010010101;
LUT[163]<=32'b10011101111100011110010010001110;
LUT[164]<=32'b10011011111100101101011111011011;
LUT[165]<=32'b10011010000000110101001111010101;
LUT[166]<=32'b10011000001000111010010101101011;
LUT[167]<=32'b10010110010101000001011100010101;
```

```
LUT[168]<=32'b10010100100101001111000011001110;
LUT[169]<=32'b10010010111001100111100000000000;
LUT[170]<=32'b10010001010010001110111110000011;
LUT[171]<=32'b10001111101111001001011110001011;
LUT[172]<=32'b10001110010000011010110110100000;
LUT[173]<=32'b10001100110110000110110010010111;
LUT[174]<=32'b10001011100000010000110010000110;
LUT[175]<=32'b10001010001110111110000101011110;
LUT[176]<=32'b10001001000010001100000111000000;
LUT[177]<=32'b10000111111010000011100100110110;
LUT[178]<=32'b10000110110110100101010111101011;
LUT[179]<=32'b10000101110111111010000111000111;
LUT[180]<=32'b10000100111101110010001111000110;
LUT[181]<=32'b10000100001000100001111111110001;
LUT[182]<=32'b10000011011000000101011101011011;
LUT[183]<=32'b10000010101100011110100000011010;
LUT[184]<=32'b10000010000101101110110101000011;
LUT[185]<=32'b10000001100011110111111011100110;
LUT[186]<=32'b10000001000110111011001000001010;
LUT[187]<=32'b10000000101110111001100010101001;
LUT[188]<=32'b10000000011011110100000110101111;
LUT[189]<=32'b10000000001101101011100011110111;
LUT[190]<=32'b10000000000100100000011101000111;
LUT[191]<=32'b10000000000000010011001001010010;
LUT[192]<=32'b10000000000001000011110010110101;
LUT[193]<=32'b10000000000110110010010111110110;
LUT[194]<=32'b10000000010001011110101010001000;
LUT[195]<=32'b10000000100001001000001111000111;
LUT[196]<=32'b10000000110101101110011111111001;
LUT[197]<=32'b10000001001111010000101001010110;
LUT[198]<=32'b10000001101101101101101100000000;
LUT[199]<=32'b10000010010001000100011100001111;
LUT[200]<=32'b10000010111001010011100010001110;
LUT[201]<=32'b10000011100110011001011001111111;
LUT[202]<=32'b10000100011000010100010011100010;
LUT[203]<=32'b10000101001111000010010010110101;
LUT[204]<=32'b10000110001010100001001111111111;
LUT[205]<=32'b10000111001010101110110111001101;
LUT[206]<=32'b10001000001111101000101000111110;
LUT[207]<=32'b10001001011001001011111010001001;
LUT[208]<=32'b10001010100111010101110011111111;
LUT[209]<=32'b10001011111010000011010100011000;
LUT[210]<=32'b10001101010001010001001101110101;
LUT[211]<=32'b10001110101100111100000111101100;
LUT[212]<=32'b10010000001101000000011110001111;
LUT[213]<=32'b10010001110001011010100010110100;
```

```
LUT[214]<=32'b10010011011010000110011100000001;
LUT[215]<=32'b10010101000111000000000101110001;
LUT[216]<=32'b10010110111000000011010001100011;
LUT[217]<=32'b10011000101101001011100110100010;
LUT[218]<=32'b10011010100110010100100001110000;
LUT[219]<=32'b10011100100011011001010110010001;
LUT[220]<=32'b10011110100100010101001101011000;
LUT[221]<=32'b10100000101001000011000110110011;
LUT[222]<=32'b10100010110001011101111000110110;
LUT[223]<=32'b10100100111101100000010000100111;
LUT[224]<=32'b10100111001101000100110010001111;
LUT[225]<=32'b10101001100000000101111001000101;
LUT[226]<=32'b10101011110110011101110111111011;
LUT[227]<=32'b10101110010000000110111001001101;
LUT[228]<=32'b10110000101100111010111111010001;
LUT[229]<=32'b10110011001100110100000100100011;
LUT[230]<=32'b10110101101111101011111011111000;
LUT[231]<=32'b10111000010101011100010000101001;
LUT[232]<=32'b10111010111101111110100111000101;
LUT[233]<=32'b10111101101001001100011100100001;
LUT[234]<=32'b11000000010110111111000111101001;
LUT[235]<=32'b11000011000111001111111000101110;
LUT[236]<=32'b11000101111001110111111001111001;
LUT[237]<=32'b11001000101110110000001111011100;
LUT[238]<=32'b11001011100101110001111000000001;
LUT[239]<=32'b11001110011110110101101100111101;
LUT[240]<=32'b11010001011001110100100010100011;
LUT[241]<=32'b11010100010110100111001000010100;
LUT[242]<=32'b11010111010101000110001001010001;
LUT[243]<=32'b11011010010101001010001100001101;
LUT[244]<=32'b11011101010110101011110100000001;
LUT[245]<=32'b11100000011001100011011111111101;
LUT[246]<=32'b11100011011101101001101011111101;
LUT[247]<=32'b11100110100010110110110000110111;
LUT[248]<=32'b11101001101001000011000100110011;
LUT[249]<=32'b11101100110000000110111011011011;
LUT[250]<=32'b11101111110111111010100110010000;
LUT[251]<=32'b11110011000000010110010100111100;
LUT[252]<=32'b11110110001001010010010101100011;
LUT[253]<=32'b11111001010010100110110100111101;
LUT[254]<=32'b11111100011100001011111111000010;
LUT[255]<=32'b11111111100101111001111111000001;
LUT_out <= 32'b0;
end

else
```

```
LUT_out <= LUT[address][31:24];
LUT_fractional <= LUT[address][23:0];

end
endmodule
```

To get the memory contents for LUT of traditional DDS, please check **Section 5.2, Step 6.**

/*Adder*/

```
module adder (a, b, sum);

input [31:0] a, b;
output [31:0] sum;

reg [31:0] sum;

always @ (a or b)
begin
        sum = a + b;
end
endmodule
```

References

[1] Analog Devices, "A technical tutorial on digital signal synthesis," Application Note, 1999. Available: http://www.analog.com/static/imported-files/tutorials/450968421DDS_Tutorial_rev12-2-99.pdf

[2] National Instruments, "Understanding direct digital synthesis," Application Note, 2006. Available: http://www.ni.com/white-paper/5516/en/

[3] Y. Yang, J. Cai, J. Schutt-Aine, (2013) "A novel truncation spurs free structure of direct digital synthesizer," *COMPEL - The International Journal for Computation and Mathematics in Electrical and Electronic Engineering*, vol. 32 issue no. 2, pp. 454 – 466.

[4] K. Bhagat, "Tutorial on designing and implementing a direct digital synthesizer (DDS) on a field programmable gate array (FPGA)," M.S. thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2012.

[5] C. Shan, Z. Chen, H. Yuan and W. Hu, "Design and implementation of a FPGA-based direct digital synthesizer," in *Electrical and Control Engineering (ICECE), 2011 International Conference*, pp. 614-617.

[6] T. M. Comberiate, "Phase noise spur reduction in an array of direct digital synthesizers," M.S. thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2010.

[7] H. Omran, K. Sharaf, and M. Ibrahim, "An all-digital direct digital synthesizer fully implemented on FPGA," in *Design and Test Workshop (IDT), 2009 4th International*, pp. 1-6.

[8] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, January 1949, vol. 37, no. 1, pp. 10-21.

[9] J. Vankka, *Digital Synthesizers and Transmitters for Software Radio*. Dordrecht, The Netherlands: Springer, 2005.

[10] A. A. Alsharef, M. A. Mohd, Ali and H. Sanusi, "Direct digital frequency synthesizer simulation and design by means of Quartus-ModelSim," *Journal of Applied Science*, 2012, vol. 12, no. 20, pp. 2172-2177.

[11] Analog Devices, "Determining if a spur is related to the DDS/DAC or to some other source," Application Note, 2007. Available: http://www.analog.com/static/imported-files/application_notes/131351807AN_927.pdf

[12] World of ASIC, 2012, website available: http://www.asic-world.com/

[13] ECE 385 Digital Systems Laboratory, 2014, website available: https://wiki.cites.illinois.edu/wiki/display/ECE385/Home

[14] CS 233 Computer Architecture, 2014, website available: https://wiki.cites.illinois.edu/wiki/display/cs233fa14/Home

[15] Altera DE2-115 Development and Education Board, 2014, website available: http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html

[16] ISE FPGA Design Flow Overview, Xilinx Inc., 2008, website available: http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm

[17] M. Chaitanya, "FPGA Design Flow," 2012, website available: http://digitaltagebuch.wordpress.com/2012/11/26/fpga-design-flow/

[18] FPGA Design Flow Overview, FPGA Central, 2011, website available: http://www.fpgacentral.com/docs/fpga-tutorial/fpga-design-flow-overview

[19] FPGA design implementation, CORE Technologies, 2009, website available: http://www.1-core.com/library/digital/fpga-design-tutorial/implementation_xilinx-shtml

[20] Altera, "DE2-115 User Manual," 2010. Available: ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Boards/DE2-115/DE2_115_User_Manual.pdf

[21] X. Geng, F. Da, J. Irwin and C. Jaeger, "An 11-Bit 8.6 GHz DDS MMIC with 10-bit segmented sine-weighted DAC," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 2, pp. 300-313, February 2010.