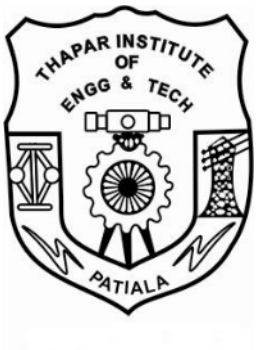
Design of High Performance Frequency Divider

THESIS

Submitted in partial fulfillment of the requirements for the award of the degree of

Master of Technology
in
VLSI Design and CAD



Submitted By
Varinder Deepak
Regn. No. 6040417

Under the supervision of

Mrs. Manu Bansal
Sr. Lecturer (ECED)
TIET,Patiala

DEPARTMENT OF ELECTRONIC AND COMMUNICATION ENGINEERING
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY
(DEEMED UNIVERSITY)
PATIALA(PUNJAB) – 147004
June 2006

Certificate

I **Varinder Deepak**, hereby certify that the work which is being presented in the thesis entitled, **"Design of High Performance Frequency Divider"**, by me in partial fulfillment of the requirements for the award of degree of Master of Technology in VLSI Design and CAD submitted in Department of Electronics and Communication Engineering, Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision of Mrs. Manu Bansal.

The matter presented in this thesis has not been submitted in any other University / Institute for the award of any degree.

Varinder Deepak
Roll No. 6040417

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Manu Bansal
Sr. Lecturer(ECED)
TIET, Patiala
Date…….

Countersigned by

Head, ECED
T.I.E.T, Patiala
Date…….

Dean of Academic Affairs
T.I.E.T, Patiala
Date…….

PREAMBLE

"Within each closure,

Lies the potential for an exciting beginning, Within this beginning,

Lies an astonishing future"

Acknowledgement

At this moment, I believe, is the most suitable time for me to express my heartfelt gratitude to the people who have helped me and supported me during my stay in TIET.

My sincerest gratitude goes to **Dr. R.S Kaler** (Head of Department) for giving me this exciting opportunity and providing the amicable and nurturing ambiance that has culminated in a pleasant experience. His expert guidance and suggestions always kept me motivated to meet some very challenging deadlines.

Although the Knowledge like electricity pervades everywhere, yet the Teacher is the point where it shines as light. I am extremely lucky to have an opportunity to blossom under the supervision and guidance of **Mrs. Manu Bansal** (Senior Lecturer) who helped me to grow like Phoenix out of the ashes of my shortcomings and failures. To me, she is not mere a person but a current of love who treaded with me through thick and thin.

I would like to thank **Mr. Anish Lal** for his unflinching desire to know what was going on in my thesis and for those million of opportunities he had given me to share his ideas and knowledge.

I am highly indebted to **Mr. Kulbir Singh**, for his whole-hearted support and suggestions. I am highly thankful to him that in spite of his demanding professional preoccupation he always made himself available for guidance and assistance to me in my thesis work.

I heart fully extend my regards and thanks to **Mr. Sanjay Batish**, who has imparted his immense knowledge in a very channelized way to me. I am very much thankful for his wise and synergic help throughout my training period.

I would like to thank the faculties of **ECED**, for technical suggestions as well as great collegial working atmosphere. The whole staff of **ECED** was highly cordial and helpful in all possible ways. It would have really difficult for me to have proceeded without their constant encouragement.

Varinder Deepak

Abstract

The frequency divider is an important building block in today's high speed circuits because it is an integral part of the phased locked loop employed in frequency synthesizer to generate frequencies from a single stable reference frequency. Mostly a crystal oscillator is used for the reference frequency. Most of the frequency synthesizer employs a Phase Locked Loops circuit, as this technique offer many advantages such as minimum complex architecture, low power consumption and a maximum use of Large Scale Integration technology. There are many designs in communication that require frequency synthesizer to generate a range of frequencies; such as cordless telephones, mobile radios and other wireless products. The accuracy of the required frequencies is very important in these designs as the performance is based on this parameter. One approach to this necessity could be to use crystal oscillators. It is not only impractical, but is impossible to use an array of crystal oscillators for multiple frequencies. Therefore some other techniques must be used to circumvent the problem. The main benefit of using Phase Locked Loop technique in frequency synthesizer is that it can generate frequencies comparable to the accuracy of a crystal oscillator and offer other advantages mentioned previously.

Considering the scope of the frequency divider, this work is devoted to the designing of an efficient frequency divider. Which can be employed in frequency synthesizer using phase locked loop technique. The work embodied the designing of divide by 64, divide by 65 prescalers, swallow counter to count down from the loaded number to zero, Main counters and a control unit to govern the operations of sub-functional units employed in final design of frequency divider.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ASIC	Application Specific Integrated Circuit
CLB	Configuration Logic Block
DLL	Delay Locked Loop
FPGA	Floating Point Gate Array
FSM	Finite State Machine
IOB	Input Output Block
LUT	Look Up Table
MUX	Multiplexer
PLA	Programmable Logic Array
PLD	Programmable Logic Devices
PLL	Phase Locked Loop
RAM	Random Access Memory
VCO	Voltage Control Oscillator
VHDL	VHSIC Hardware Description Language

Chapter 1
Introduction

1.1 General

A frequency divider is used in a frequency synthesizer is a circuit design that generate a new frequency from a single stable reference frequency [12,13]. Mostly a crystal oscillator is used for reference frequency. Most of the frequency synthesizer employs a phase locked loops circuit, as this technique offer many advantages such as minimum complex architecture, low power consumption and a maximum use of large scale integration technology. The accuracy of required frequencies is very important in these designs as the performance is based on this parameter. One approach to this necessity could be to use a crystal oscillator. It is not only impractical, but it is impossible to use an array of crystal oscillator for multiple frequencies. Therefore some other technique must be used to circumvent the problem. The main benefit of using phase locked loop technique in frequency synthesizer is that it can generate frequencies comparable to the accuracy of crystal oscillator and offer other advantages mentioned previously [2,6,7]. The frequency divider is the feedback element that forces the VCO output to be a multiple of the reference frequency [15,16]. Just as in any negative feedback system with enough forward path gain, the transfer function reduces to the inverse of the feedback element. In the case of the PLL [4,5,14], this forces the input frequency to be scaled by the prescaler divide ratio. A problematic situation can occur if the maximum operating frequency of the VCO is faster than that of the prescaler. For example, if the VCO begins oscillating at a frequency beyond the range of the prescaler, the output of the prescaler will either be a constant or possibly a divided version of its own natural frequency, both of which would be slower than the reference frequency. The loop would respond by forcing the VCO to oscillate faster until the integrator output becomes saturated and the VCO is trapped, operating at its highest output frequency. Because the prescaler must operate at a higher frequency than the VCO, this circuit is not trivial, especially if the technology requires a large effort to get the VCO to operate at these speeds [18]. The

speed of the prescaler is the ultimate limit in the design of integrated frequency synthesizers [20].

1.2 Problem Formation

Ripple counters are relatively simple, low-power circuits which are generally used as fixed frequency dividers. However, because the signal ripples through each of the stages sequentially, these counters can have relatively high levels of phase instabilities. Other forms of asynchronous dividers include regenerative dividers and injection locked oscillators. These circuits can operate at very high speeds but only over a relatively narrow band of frequencies. Because of such limitations, these types of Frequency Divider are not suitable for use in applications that require the oscillator to have a wide tuning range. The synchronous counters tend to have lower levels of phase instabilities, but higher levels of power consumption than ripple counters [21,22,23]. One of the biggest constraints in a high speed frequency divider design is the speed limit of the programmable divider N. A single divide by N unit can handle only up to 25MHz of frequency [25]. Therefore some special design techniques are necessary to implement a programmable divider in high speed design.

However there are many ways for overcoming this limitation of frequency. Such as,

1) VCO output may be fixed with the out put of a crystal oscillator and the resulting remaining or the difference frequency can be fed to the programmable divider.
2) The VCO output may be multiplied from a low value in the operating range of the programmable divider to the required high output frequency [18].
3) Or a fixed ratio divider capable of operating a high frequency may be interposed between VCO and the programmable divider.

The first method is most useful then the other two as it allows narrow channel spacing or high reference frequencies, but it has a drawback. Since the crystal oscillator and mixer are within the loop [1], any crystal oscillator noise or mixer noise appears in the divider output. The remaining two techniques are not as useful either due to their limitation.

1.3 Problem Statement

All the methods discussed above have their limitations, although all have been used in many applications. So, an improved frequency divider technique is implemented.

The actual programmable divider is used to switch the modulus of the prescaler between two consecutive values, e.g. between 64 and 65. If the programmable divider divides by *M*, and the prescaler is made to divide by *(P+1)* for *A* cycles and by *P* for the rest *(M-A)* cycles, the total division ratio will be

$$N= A (P + 1) + (M - A) P = MP + A$$

Where N is the number which divide the input frequency.

Chapter 2

Spartan-II Architecture & Synthesis Basics

FPGA is a general purpose programmable chip which can be programmed to carry out a specific hardware function specified by the user. It is an array of programmable logic blocks connected with programmable interconnects. FPGAs can have medium-to-high capacity (equivalent to that of thousands to millions) of logic gates [3,8]. FPGA is a device in which the logic structure can be directly configured by the end user without the use of an IC fabrication facility. It is a high density Programmable Logic Device containing small logic cells interconnected through a distributed array of programmable switches. This type of architecture produces statistically varying results in performance and functional capacity, but offers high register counts. Programmability typically is via volatile SRAM or one-time-programmable anti-fuses. It is a very complex PLD. These devices are fastest programmable logic devices with gate counts running into the millions. These devices are user customizable and programmable on an individual device basis. It would be impossible to give a complete description of all programmable logic devices in a space of a few pages, and consequently only certain noteworthy device architecture concept is presented in this chapter. As a good example of a modern FPGA architecture, the basic logic cell of Xilinx's Spartan-II device family is presented [10,18].

2.1 Spartan-II Architecture

Spartan-II FPGAs achieve high-performance, low-cost operation through advanced architecture and semiconductor technology [18,22]. Spartan-II devices provide system clock rates up to 200 MHz. Spartan-II FPGAs offer the most cost-effective solution while maintaining leading edge performance. In addition to the conventional benefits of high-volume programmable logic solutions, Spartan-II FPGAs also offer on-chip synchronous single-port and dual-port RAM (block and distributed form), DLL clock drivers, programmable set and reset on all flip-flops, fast carry logic, and many other features.

2.1.1 Block Diagram

The Spartan-II family of FPGA is implemented with a regular, flexible, programmable architecture of configuration logic blocks (CLB), surrounded by a perimeter of programmable input/output blocks(IOBs), interconnected by a powerful hierarchy of versatile routing resources. The architecture also provides advanced functions such as Block RAM. There are four delay locked loops (DLLs), one at each corner of the die, between the CLBs and the IOB columns. Stored values in the cells determine logic functions and the interconnections implemented in the FPGA.

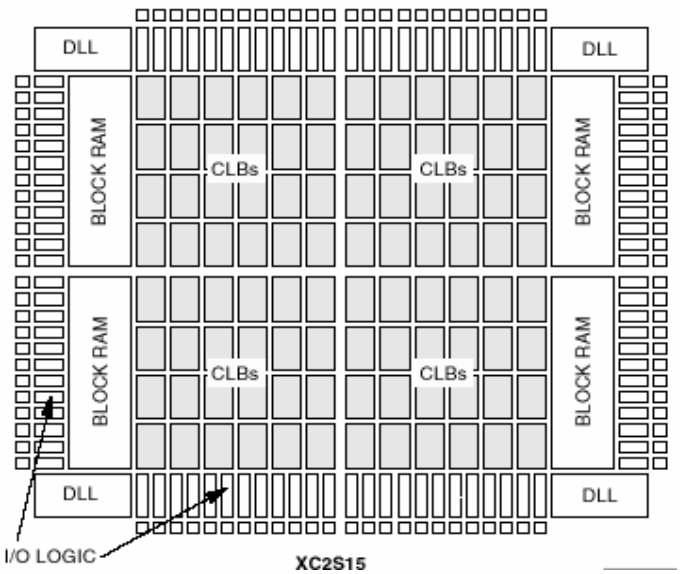


Figure-2.1 Basic Spartan-II Family FPGA Block Diagram

2.1.2 Configuration Logic Block

The basic building block of the Spartan-II CLB is the logic cell (LC). An LC includes 4-input function generator, carry logic, and storage element. Output from the function generator in each LC drives the CLB output and the D input of the flip plop. Each Spartan-II CLB contains four LCs, organized in two similar slices; single slice is shown in Figure2.2. In addition to the four basic LCs, the Spartan-II CLB contains logic that combines function generators to provide functions of five or six inputs.

2.1.2.1 Look Up Tables

Spartan-II function are implemented are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide 16×1 bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16×2 –bit or 32×1 bit synchronous RAM or 16×1 bit dual port synchronous

RAM. The Spartan-II LUT can also provide a16-bit shift register that is ideal for capturing high speed.

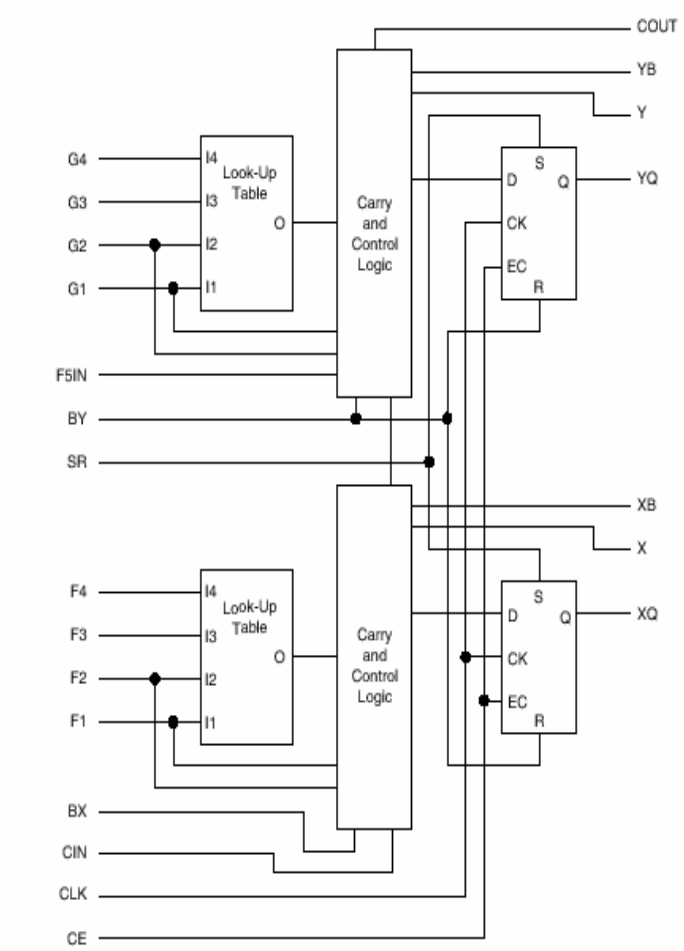


Figure-2.2 Spartan-II CLB Slice

In addition to the four basic LCs, the Spartan-II CLB contains logic that combines function generators to provide functions of five or six inputs. Consequently, when estimating the number of system gates provided by a given device, each CLB counts as 4.5 LCs. The storage elements in the Spartan-II slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. All control signals are independently invertible, and are shared by the two flip-flops within the slice.

The F5 multiplexer in each slice combines the function generator outputs. This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up to nine inputs. Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by

selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs.

2.1.3 Input/Output Block

The IOBs are located around all the logic and memory elements for easy and quick routing of signals on and off the chip. The Spartan-II IOB, as seen in Figure2.3, features inputs and outputs that support a wide variety of I/O signaling standards. These high-speed inputs and outputs are capable of supporting various state-of-the-art memory and bus interfaces. The three IOB registers function either as edge-triggered D-type flip-flops or as level sensitive latches.

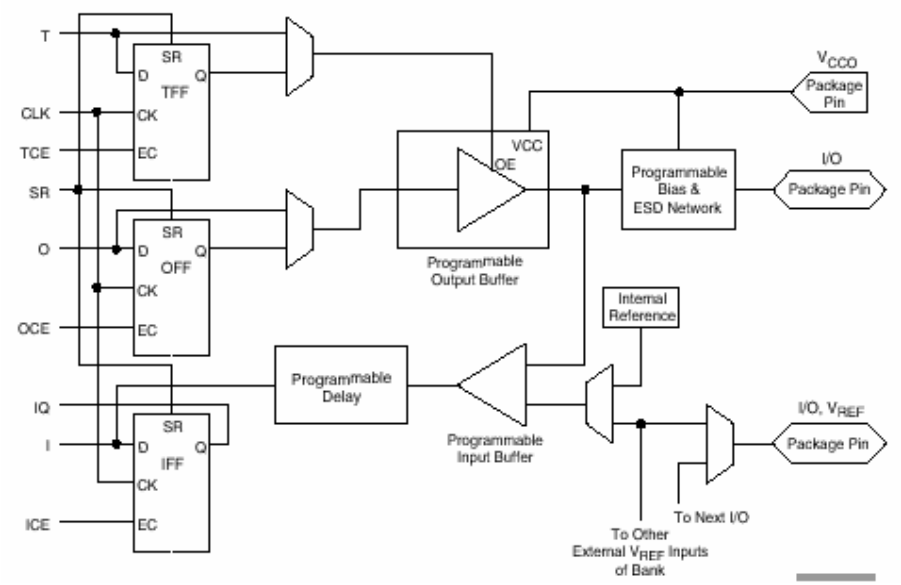


Figure-2.3 Spartan-II Input/Output Block

Each IOB has a clock signal (CLK) shared by the three registers and independent Clock Enable (CE) signals for each register. In addition to the CLK and CE control signals, the three registers share a Set/Reset (SR). For each register, this signal can be independently configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear.

2.1.4 Block RAM

Spartan-II FPGAs incorporate several large block RAM memories. Block RAM memory blocks are organized in columns. All Spartan-II devices contain two such columns, one along each vertical edge. These columns extend the full height of the chip. Each memory block is four CLBs high, and consequently, a Spartan-II device

eight CLBs high will contain two memory blocks per column, and a total of four blocks. Each block can be configured at ratios between 4Kx1 and 256x16.

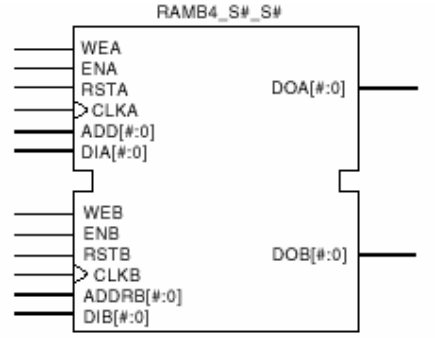


Figure-2.4 Dual Port Block RAM

2.1.5 Routing Matrix

It is the longest delay path that limits the maximum speed of any worst case design. Most Spartan-II signals are routed on the general purpose routing, and consequently, the majority of interconnect resources are associated with this level of the routing hierarchy.

The general routing resources are located in horizontal and vertical routing channels associated with the rows and columns CLBs. The Spartan-II also provides different types of routing these are Local Routing, I/O Routing, Dedicated Routing and Global Routing. The Spartan-II family provides high-speed, low-skew clock distribution through the primary global routing resources.

2.2 What is synthesis?

Synthesis = Translation + Optimization + Mapping

Consider the following circuit.

x<= (a and b) or (c and d);

The translation means the conversion of HDL to a generic level netlist.

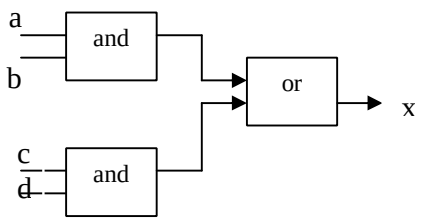
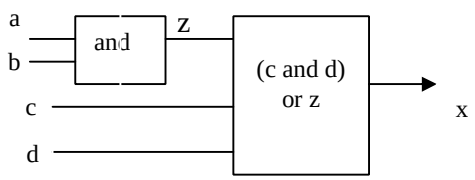


Figure-2.5 Translation of Equation for x

Optimization involves lot of techniques used for improving the characteristic of a circuit, e.g. minimization of logic, improving speed, reducing power dissipation [13].

Then the circuit is mapped to technology specific cells available in the library. The generic netlist is mapped to the cells from the library as shown below in Figure 2.6 and Figure 2.7.



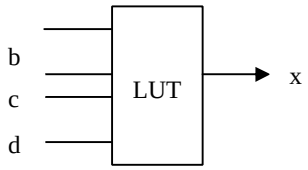


Figure-2.6 Optimization **Figure-2.7 Mapping**

Synthesis is an automated process with user level control. Synthesis is constraint driven. Constraints are user directives to the synthesis tools. Constraints are of different type such as timing constraints, area constraints.

2.3 Synthesis Coding Issues

A common misconception is that a synthesis compiler 'synthesizes VHDL' - this is incorrect. The tool synthesizes our design expressed in VHDL.

Understanding the hardware that is specified is the simplest rule for success. This is particularly important for critical timing. Conversely the easiest way to fail is write a model of design and then wonder why the synthesis tool didn't 'do the design' for us.

What does synthesize mean in this context? It means to 'transform a logic design specification into an implementation [24,25]. A synthesis tool simply handles the details of this transformation for us. This section contains examples of designer coding problems.

2.3.1 Long Signal Paths - Nested ifs

Multiple nested if or elsif clauses can specify long signal paths.

```
if sig = "000" then       -- first branch
elsif sig = "001" then    -- second branch
elsif sig = "010" then    -- third branch
elsif sig = "011" then    -- fourth branch
elsif sig = "100" then    -- fifth branch
else                      -- last branch
end if;
```

This code is an inefficient way to describe logic - a case statement would be much better. A good example is the test for the fourth branch, which depends on three previous tests and describes a long signal path, with the resulting logic delay.

```
case sig is
```

```
when "000" =>          -- first branch
when "001" =>          -- second branch
when "010" =>          -- third branch
when "011" =>          -- fourth branch
when "100" =>          -- fifth branch
when others =>         -- last branch
end case;
```

In practice, if the branches contain very little logic, or there are few branches, then there may be little difference. However, the case statement generally results in a better implementation.

2.3.2 Long Signal Paths - Loops

Loops are very powerful, but each iteration of a loop replicates logic. A variable that is assigned in one iteration of a loop and used in the next iteration results in a long signal path. This signal path may not be obvious. An example where a long signal path is the expected behavior might be a carry chain (the variable c below):

```
function "+" (a,b:bit_vector)return bit_vector is      -- assumes a,b descending
variable sum : bit_vector (a'length downto 0);
variable c :bit := '0';
begin
for i in a'reverse_range loop
   sum(i) := a(i) xor b(i) xor c;
   c := (a(i) and c) or (b(i) and c) or (a(i) and b(i));
end loop; sum(a'length) := c; return sum;
end;
```

2.3.3 Simulation-optimized code

It is likely that code written for optimal simulation speed will not be an optimal description of the logic. In the following example it is assumed that only one control input will be active at a time. The description is efficient for simulation, but a poor logic description because the independence of the control signals is not described within the VHDL code.

```
 out1 <= '0';
 out2 <= '0';
 out3 <= '0';
 if in1 = '1' then
out1 <= '1';
elsif in2 = '1'
then out2 <= '1';
elsif in3 = '1'
then out3 <= '1';
end if;
```

The independence of the control signals needs to be contained within the design description. The result may be slightly slower simulation, but a smaller logic implementation after synthesis.

```
out1 <= '0';
 out2 <= '0';
out3 <= '0';
if in1 = '1' then
out1 <= '1';
end if;
if in2 = '1' then
out2 <= '1';
end if;
if in3 = '1' then
out3 <= '1';
end if;
```

Note that the issue is not a long signal path, but an unclear specification of the design. The best optimizer in the world can't turn an inefficient algorithm into an efficient one. An algorithm that is efficient from one viewpoint may not be efficient from another.

2.3.4 Port Mode inout or buffer

Simply an issue of over specification - inout specifies bi-directional dataflow, buffer, like out, specifies unidirectional dataflow. There are very few occasions in hardware design when bi-directional data flow on a single wire is actually what we want. Use inout when one wants to specify a signal path that is actually routed through a pin, such as a Xilinx IOB. Designers often use inout when they have a logical output they wish to read from, in this case use mode buffer. This results in a signal path internal to the target device It is not a good idea to use inout on lower levels of hierarchy when separately compiling each design unit. Doing so may be a problem for third party linkers. If the design units are compiled at the same time, the implementation will be two wires, one for data flow in each direction.

2.3.5 Depending on Initial Value

The initial value of a signal or variable is the value specified in the object's declaration (if not specified there is a default initial value). The initial value of such an object is its value when created. Signals and variables declared in processes are created at 'time zero'. Variables declared in subprograms are created when the subprogram is called.

The value at time zero has no clear meaning in the context of synthesis, therefore, the initial value of signals and process variables must be used with care. This issue does not arise with the initial value of variables declared in subprograms.

One should not depend on the initial value of signals or process variables if they are not completely specified in the process in which they are used. In this case, the compiler will ignore the time zero condition and use the driven value - effectively ignoring the

single transition from the time zero state. If such signals or variables are not assigned, we may reliably use their initial value. Obviously, signals assigned in another process will never depend upon the initial value. For example:

```
begin
signal res1 : bit := '0';
process (tmpval,INIT)
begin
if (tmpval = 2**6 -1) then
res1 <= '1';
elsif (INIT ='1') then
res1 <= '1';
end if;
end process;
```

In this case 'res1' is never assigned low - the code will be synthesized as a pull-up. However during simulation at time zero, 'res1' starts at '0', makes one transition to '1' and stays there. If this is really the intent, the solution is to use a flip-flip.

This design probably depends upon a wire floating low at power up, and probably has no realizable implementation. A solution might be:

```
process (tmpval,INIT)
begin
if (tmpval = 2**6 -1) then
res1 <= '1';
elsif (INIT ='1') then
res1 <= '1';
else
res1 <= '0';       -- drive it low
end if;
end process;
```

2.3.6 Unintended Latches

Latches are inferred using incomplete specification in an if statement. The following example specifies a latch gated by 'address_strobe', which may not be the intent.

```
process (address, address_strobe)
begin
if address_strobe = '1' then
decode_signal <= address = "101010";
end if;
end process;
```

This says, when address_strobe is '0', then decode_signal holds its previous value, resulting in the latch implementation. In this case the intent is probably to ignore decode_signal when address_strobe is '0'. However, it need to be explicit.

```
if address_strobe = '1' then
decode_signal <= address = "101010";
else
decode_signal <= false;
end if;
```

2.3.7 Unintended Combinational Feedback

It is possible to specify unintended combinational feedback paths by using variables (declared in a process) before they are assigned, or by incomplete specification.

In the following example, if the ReadPtr(i) is never equal to '1', Qint keeps its previous value. It may be a characteristic of the design that one bit of ReadPtr is always '1', but nothing says this is so. Qint is incompletely specified and a feedback path exists, which includes Qint when ReadPtr is all zeros.

```
process (ReadPtr, Fifo)
begin
for i in ReadPtr'range loop
if ReadPtr(i) = '1' then
Qint <= Fifo(i);
end if;
end loop;
end process;
```

This case is coded for by making certain Qint is always assigned, in which case its value is defaulted to all zeros and the unintended feedback path is removed.

```
process (ReadPtr, Fifo)
begin
Qint <= (others => '0');          -- because of possible comb feedback
for i in ReadPtr'range loop
if ReadPtr(i) = '1' then
Qint <= Fifo(i);
end if;
end loop;
end process;
```

2.3.8 Observe the Register Inference Conventions

Synthesis tools infer storage devices (such as latches and flip flops) from incomplete assignment of variables or signals. To the other extreme it is possible to specify storage elements that the synthesis tool won't recognize. For example:

```
process (clk1,clk2)
begin
if rising_edge(clk1) then
if rising_edge(clk2) then
q <= d;
end if;
end if;
end process;
```

This probably describes a flip-flop that loads when its two clocks change at the same instant. It will function during simulation (because of the discrete nature of simulation time) but no hardware element has this behavior, and the compiler will report an error.

Thus Spartan-II FPGA has a regular, flexible and programmable architecture of CLBs. The Spartan-II gives users high performance, abundant logic resources and rich feature set.

2.4 Design Flow

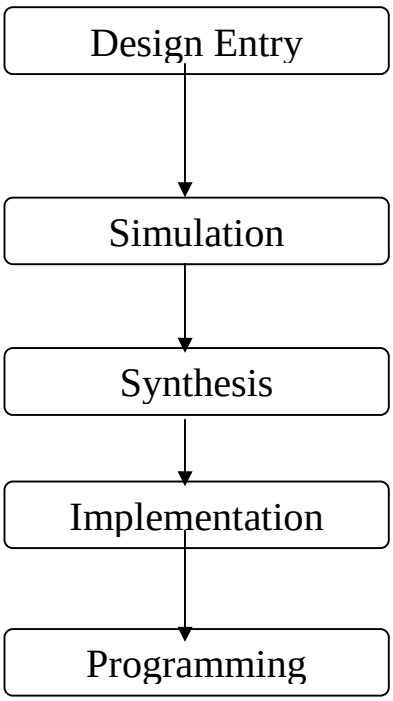


Figure-2.8 Design Flow

To start new design sequence is as follows

- DESIGN ENTRY
- SIMULATION
- SYNTHESIS
- IMPLEMENTATION
- PROGRAMMING

2.4.1 Design Entry

User can enter the design by three ways.

- HDL
- FSM

- SCHEMATIC

HDL (Hardware description language) can be written in two languages

- VHDL (very high speed integrated circuit) HDL
- VERILOG

In HDL there are, there are two parts in a programme

- ENTITY
- ARCHITECTURE

Entity- It is description of input and output.

Architecture-is functional behavior CODE of design. Once code has been completed, then syntax checks has to be performed and error free code has to be added in the project manager by using the option ADD TO PROJECT from PROJECT icon in the task Bar of HDL editor

2.4.2 Simulation

In simulation user can verify the functionality of his design by applying various input signal combination and observing the result the output results. The simulation is performed on gate level Netlist

2.4.3 Synthesis

It is the process which converts HDL CODE in to gate level circuit in the form of NET LIST. This process is Target Technology dependent and hence user must select proper DEVICE, FAMILY, PARTNUMBER and SPEED GRADE. Also user may select SYNTHESIS SETTINGS like CLOCK FREQUENCY, OPTIMIZATION FOR SPEED or AREA.

2.4.4 Implementation

Implementation is the process in which the design is passed through various stages by TRANSLATE,MAPPING,TIME ANALYSIS and BITSTREAM. For locking input and output signal to particular pins of the device user must write UCF (user constraint file) before implementation and guide the same file to implementation tool through the option set control files. Output of implementation is .jed for CPLD and .bit file for FPGA, which can be directly program in to target device.

2.4.5 Programming

This is the process by which user can physically download the design programming files from PC to Target device using programming cable.

- To programme CPLD, select Boundary scan (JTAG) mode
- To programme FPGA, select Boundary scan, slave serial mode or Master serial mode.

Chapter 3

Frequency synthesizer

A frequency synthesizer generates a range of output frequencies from a single stable reference frequency of a crystal oscillator. Many applications in communication require a range of frequencies or a multiplication of a periodic signal For example, in most of the FM radios, a phase-locked loop frequency synthesizer technique is used to generate 101 different frequencies. Also most of the wireless transceiver designs employ a frequency synthesizer to generate highly accurate frequencies, varying in precise steps, such as from 600 MHz to 800 MHz in steps of 200 KHz [15]. Frequency Synthesizer are also widely used in signal generators and in instrumentation systems, such as spectrum analyzers and modulation analyzers

3.1 Frequency synthesis techniques

Several different frequency synthesis techniques have been presented in the literature over the years. They can be quite clearly divided into three separate categories, namely direct analog synthesis, direct digital synthesis, and indirect analog synthesis [21,24,25]. In this context, “indirect” refers to a system based on some kind of a feedback action, whereas “direct” refers to a system having no feedback.

3.1.1 Direct analog synthesis

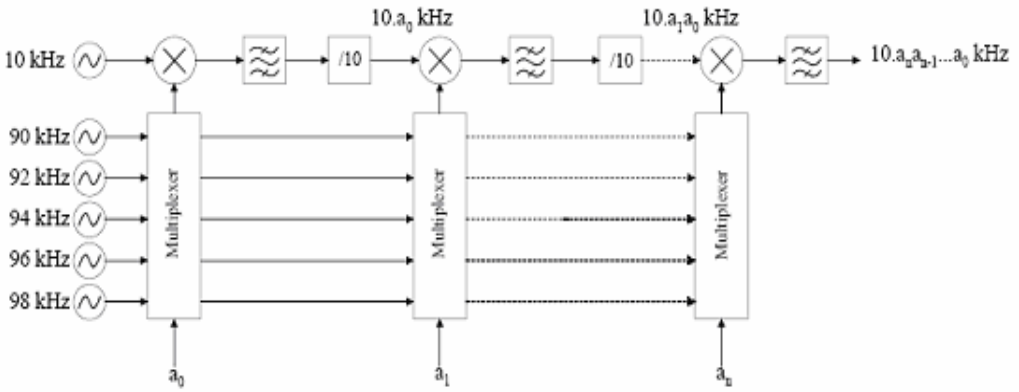


Figure-3.1 The Block Diagram of a Direct Analog Synthesizer

Figure 3.1 shows an example of a direct analog synthesizer. The frequency resolution is achieved by mixing signals of certain frequencies, and then dividing the resulting frequency down. Theoretically, this process can be repeated arbitrarily many times to achieve a finer frequency resolution.

Advantages of the direct analog synthesis are very fast switching times and, in theory, arbitrarily fine frequency resolution. However, this technique requires a very large amount of hardware, as can clearly be seen even from the simple example block diagram (Figure 3.1). Also visible in the figure 3.1 is the fact that the synthesized frequency is lower than the highest input frequency. Therefore, the use of direct analog synthesis techniques in high-frequency applications is severely limited.

Also noise is a problem in direct analog synthesis. To achieve a reasonably low-noise output signal, all input frequencies will have to be low-noise crystal oscillators, resulting in a lot of external components. Moreover, all the mixers, band pass filters, and dividers are in the signal path, meaning that their noise will also contribute to the phase noise in the synthesized frequency.

For the reasons mentioned above, the use of direct analog synthesizers is limited to low frequencies and to applications that are not too sensitive to noise. Even in these applications, they are relatively expensive compared to the synthesis techniques presented in the following sections. Therefore, very few direct analog synthesizers, if any, are used in commercial applications.

3.1.2 Direct digital synthesis

Figure 3.2 shows the basic principle of direct digital synthesis. The desired output frequency is fed to the phase accumulator as a digital word. The phase accumulator increments its output value by this word once every clock cycle. When the full scale of the accumulator is reached, it wraps around. The output of the phase accumulator is thus a digital ramp signal, whose period is the same as that of the desired output frequency. In other words, the phase accumulator output contains information about the instantaneous phase of the synthesized frequency.

The amplitude of a sinusoidal signal at different phase values is stored in the sine read-only memory (ROM). The instantaneous phase of the desired output signal is

used as the address to the ROM, and the output is the instantaneous amplitude of the synthesized signal [9,10].

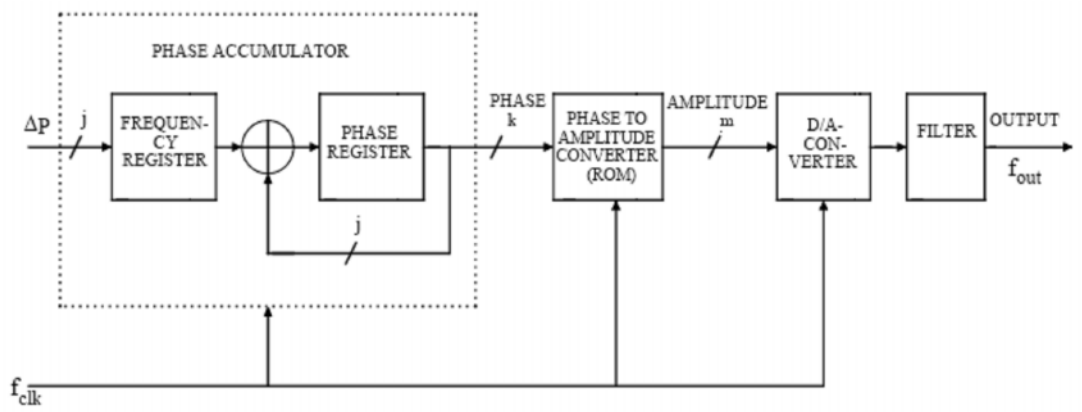


Figure-3.2 The block diagram of direct digital synthesizer

To get an analog output signal, the amplitude information has to be converted to the analog domain in the digital-to-analog converter (DAC). The output of the DAC contains a lot of spurious tones, harmonics, etc., that have to be filtered out before the signal can be used. The smoothing filter in the output of the DAC attenuates the harmonics to an acceptable level, but the in-band spurious tones still remain. Their frequencies are predictable, but as they are in the signal band, they will not be attenuated by the filter. Direct digital synthesis has some very strong advantages. It has arbitrarily fine frequency resolution and a very high switching speed. Also, different phase, frequency, and amplitude modulations can be implemented in the digital domain, and require only a small amount of extra hardware. Due to the fact that most of the signal processing is done in the digital domain, direct digital synthesis also lends itself very well to full integration in a CMOS or BiCMOS technology.

Until recently, the main disadvantage of direct digital synthesis has been the speed requirement and the huge power dissipation in the digital parts of the circuit, i.e. the phase accumulator and the sine ROM. However, with modern deep submicron CMOS technologies, the power dissipation has been dramatically reduced, and the achievable speed has become fairly high. Now, the bottleneck in the direct digital synthesizer is the DAC. Demands on the DAC clock frequency, resolution, and linearity are overwhelming. This limits the use of direct digital synthesizers in high-frequency applications. However, they have become defacto standard in high-performance low-

frequency signal generators. Recently, they have also found use in cellular base station applications.

3.1.3 Indirect analog synthesis

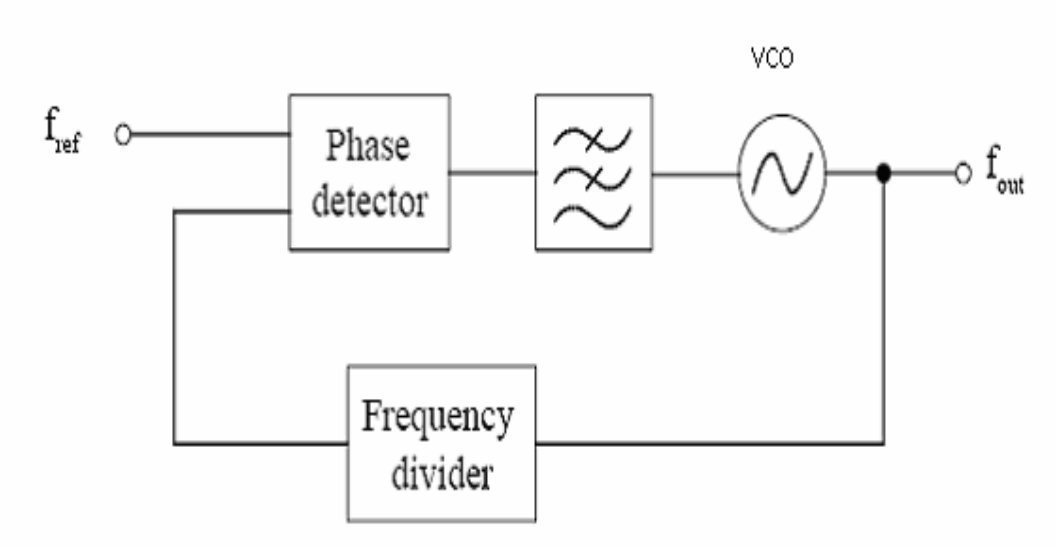


Figure-3.3 The Block Diagram of Simple Phase Locked Loop

Figure 3.3 shows the block diagram of the phase-locked loop, i.e. an indirect analog synthesizer, at its simplest [20, 27, 28]. Here, the synthesis is based on the feedback action of the loop. The output frequency is divided down in the frequency divider. The phase of the output signal of the divider is compared with the phase of a reference signal in the phase detector. The output of the phase detector is low pass filtered to generate a control voltage for the voltage-controlled oscillator (VCO). If the phase of the frequency divider output lags the phase of the reference frequency, the phase detector steers the VCO to a higher frequency, and vice versa.

Indirect analog synthesis, or the phase-locked loop, is the most suitable technique for the synthesis of high-frequency sinusoidal signals. No block has to operate at a frequency higher than the output frequency. Also, the only component that is necessarily external is the reference frequency oscillator (or at least the crystal used as the resonator in the oscillator).

However an improved frequency divider technique is implemented, which is shown in Figure 3.4.

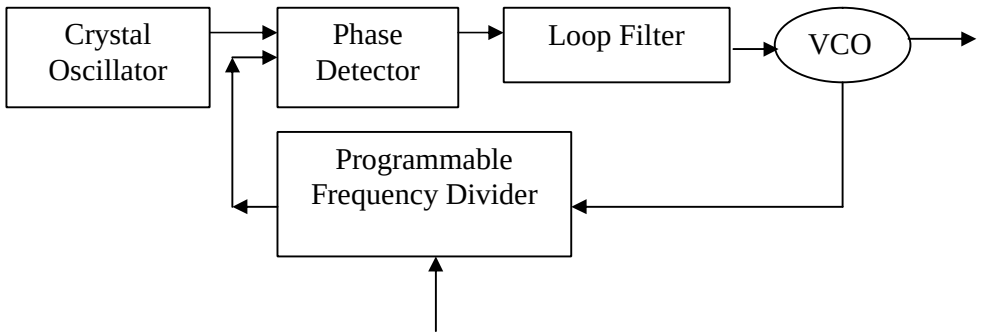


Figure-3.4 A Frequency Synthesizer

This circuit design makes use of a high frequency divider using some fixed value prescalers. In one mode it divides by P and in the other mode it divides by P+1 depending on the logic state of the control input .The prescaler reduces the high frequency by division to a lower frequency, so the rest of the circuit sees only a fraction of high output frequency. For example, if a prescaler of 20 is used at the output of 900 MHz, then the rest of the circuit only sees 45 MHz. These prescalers usually can be made to handle high frequencies in the units of MHz. In the figure a special low frequency counter is used to control the division ratio of the prescaler and consists of two programmable counter, the swallow counter (A), Main counter (M) and some control logic. Initially the two counters are loaded with the values M and A, where A<M and the modulus control signal is low, so the prescaler divides by (P+1). The counters are both decremented on every rising edge of the output of the prescaler unit, until the counter A reaches zero. When A becomes zero, the modulus control signal becomes high and the prescaler start dividing by P until the value of the M counter reaches zero. At this point both the counter are reset and the process begins again .The prescaler thus divides by (P+1) for the count value of the counter A and by (P) for M-A times. This relation can be best explained with the following equation

$$\mathbf{N = A(P+1)+(M-A)P = MP+A}$$

There are some limitations imposed by the architecture of the system on the values of M and A, Since the two modulus prescaler does not change modulus until counter A reaches zero, therefore count value in counter M should never be less than the value in counter A. This constraint also limits the minimum count, a system may reach to,

which is equal to A (M+1), since A is the maximum possible value, the swallow counter can have. So by varying the value of A, a large range of integer values can be obtained and so is the output frequencies. To use this system entirely overcomes the problems of programmable divider in high speed designs.

Chapter 4
Simulation And Synthesis Tools

Every design of VHDL is simulated and synthesized in various Tools. ModelSim SE 5.5e is used for simulation and Xilinx ISE 6.1i is used for synthesis in this thesis work

4.1 Model Sim SE 5.5 e

No matter how extraordinary or innovative a hardware design is; it is essentially useless until its proper functioning has been verified. An obvious method of testing a design is to put it out in the world and observe it in the environment for which it was designed. This is not always wise, however. Sometimes it is unsafe, economically unfeasible, or logistically impossible to do initial design testing out in the field. Furthermore, it is often important to test individual components of a design before putting them together, in order to debug them or identify trouble spots. For this reason, simulation of designs is a crucial aspect of product development [26]. ModelSim SE 5.5e is a very useful tool for performing such simulation.

4.1.1 Features of ModelSim SE 5.5e

- Its performance is equal to the most demanding simulations.
- It provides VHDL, Verilog™, and mixed-language support.
- Its powerful debugging capabilities can solve the most difficult problems.

4.1.2 Multimillion-Gate FPGA Designs

Multimillion-gate designs require high performance for all simulations, as well as the capacity to handle the demands of gate-level timing simulation. Today, the ModelSim SE 5.5e tool is used on designs exceeding 25 million gates. The ModelSim SE 5.5e tool offers a number of new performance-enhancing optimizations. ModelSim SE 5.5e has been updated with improved memory management, IEEE library performance optimizations, and other intelligent compiler advances that facilitate a broad range of designs. ModelSim's third-generation global optimization technology continues to improve RTL (Register Transfer Language) and gate-level performance

across many design styles

4.1.3 Tuning Design for Performance

Larger designs mean more tests. Typically, billions of vectors are simulated against large designs, so any drag on simulator performance can dramatically increase the amount of time you spend on verification [26]. The ModelSim SE 5.5e release includes a new option that significantly improves simulation throughput. After a design is compiled, it must be loaded into the simulator. This process is called “elaboration.” Elaboration, especially for large gate-level simulations with timing can consume a significant part of the overall simulation run time.

By analyzing the entire design flow, ModelSim SE’s integrated Performance Analyzer can uncover bottlenecks such as the impact of test bench tools, .vcd file generation, or inefficient HDL coding styles often identifying additional opportunities for better throughput. Measuring the performance impact of all areas of the environment gives the power to make better technology decisions.

4.1.4 Best User Interface and Debug Tools

The complexity of multimillion gate designs makes it no longer feasible to debug errors on the lab bench. What you need is an integrated debug environment with full access to the internal components of the design. The ModelSim SE 5.5e simulation tool delivers the industries most tightly integrated and feature rich solution for debugging. Source code debugging, waveform generation and comparison, an enhanced Dataflow window, and code coverage are some of the features of ModelSim SE 5.5e.

To simplify design entry and editing, the Source window has new code templates and design wizards to help you create VHDL code. All language constructs are available with a click of the mouse. Context sensitive expansion of templates means ,it is not required to know which constructs go where. The design wizards walk through building more complex HDL blocks, including parameterizable logic blocks, test bench stimuli, and new design objects. Advanced developers can use the code templates as an interactive language reference manual. Saving simulation data is easier with waveform viewing and exporting. These features allow you to save simulation data for viewing or comparing, even while the simulation is still running.

4.1.5 Strengths and Limitations:

One of the greatest advantages of using a ModelSim SE 5.5e software simulation is that the entire product need not be tested all at once. Individual components can be tested as they are designed, and ModelSim SE 5.5e makes it easy to take a modular approach to design and testing. In the real-world, it can sometimes be difficult to test individual components until other components are completed. It is important to keep in minds that even the most complete testing and simulation cannot guarantee real world success. While simulation is a crucial step toward validating a product, it cannot replace real world testing and actual system implementation. It is important to realize that there are a lot of variables and factors at play in the real world, and that simulation has limitations. ModelSim SE 5.5e cannot simulate mechanical wear and tear, or electronic deterioration, and therefore does not model the real system with complete accuracy. ModelSim SE 5.5e is a tool, not a complete design solution, and there are some aspects of design for which it is not perfect. However it is a valuable tool, and plays an important role in hardware design.

ModelSim SE 5.5e has a good looking future in product design and development. Its functions are extremely beneficial, and they are sure to become more complete and robust as they continue to be developed. Future versions will likely incorporate even more features, and the software is expected to advance along with the technology it attempts to emulate. It is sure to be in the toolbox of many hardware designers for years to come.

4.2 Hardware Description Language

Hardware Description Languages (HDLs) are used to describe the behavior and structure of system and circuit designs. An understanding of FPGA architecture allows to create Hardware Description Language code that effectively uses Field Programme Gate Array system features.

4.2.1 Advantages of Using HDLs to Design FPGAs

Using HDLs to design high-density FPGAs is advantageous for the following reasons.

• *Top-Down Approach for Large Projects*—HDLs are used to create complex designs.

The top-down approach to system design supported by HDLs is advantageous for large projects that require many designers working together. After the overall design plan is determined, designers can work independently on separate sections of the code.

• *Functional Simulation Early in the Design Flow*—the functionality of design can be verified early in the design flow by simulating the HDL description. Testing the design decisions before the design is implemented at the RTL or gate level allows to make any necessary changes early in the design process.

• *Synthesis of HDL Code to Gates*—synthesizing the hardware description to a design implemented with gates. This step decreases design time by eliminating the need to define every gate. Synthesis to gates also reduces the number of errors that can occur during a manual translation of a hardware description to a schematic design. Additionally, application of the automation techniques used by the synthesis tool (such as machine encoding styles or automatic I/O insertion) during the optimization of your design to the original HDL code, resulting in greater efficiency.

• *Early Testing of Various Design Implementations*—HDLs allow to test different implementations of design early in the design flow. The synthesis tool can be used to perform the logic synthesis and optimization into gates. Additionally, Xilinx® FPGAs allow to implement the design at computer. Since the synthesis time is short, designer has time to explore different architectural possibilities at the Register Transfer Level (RTL). Designer can reprogram Xilinx® FPGAs to test several implementations of design.

• *Reuse of RTL Code* —Designer can retarget RTL code to new FPGA architectures with a minimum of recoding.

Chapter 5

Design of Frequency Divider and Simulation Results

High speed frequency synthesizer designs incorporate high speed Dual-Modulus or Multi-Modulus dividers. Such circuit divides the input frequency by one of the moduli according to a control input [5]. A circuit diagram of a frequency Divider for the proposed synthesizer is shown in Figure 5.3. It consists of two Prescalers, a Main counter, a Swallow counter and a control unit. An FSM as shown in figure 5.1 has been coded in VHDL to provide an alternative approach for the prescribed design.

5.1 Frequency Divider with Finite State Machine

To represent the working of the frequency divider an FSM is devised and then coded in VHDL. The coding is done in the behavioral modeling style.

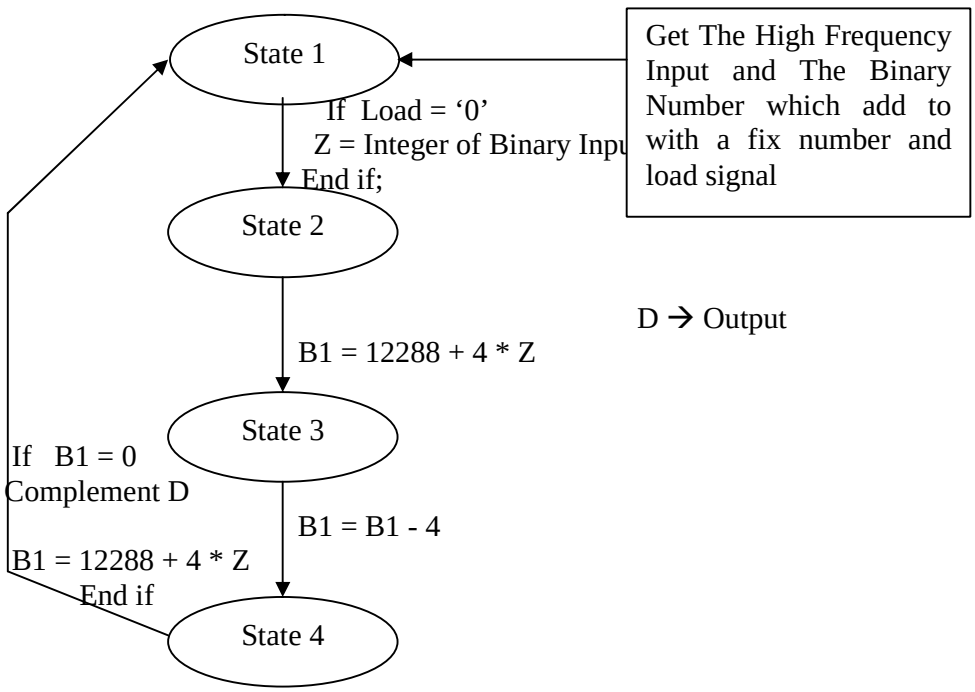


Figure-5.1 FSM of Frequency Divider

5.2 Prescaler

Frequency dividers are also called prescalers. There are two prescalers in the two modulus divider for the proposed design namely, a divide by 64 and a divide by 65 prescalers. The prescalers and their simulation results for the proposed design are presented in the following sections.

5.3 Divide By 64

Asynchronous dividers are the simplest form of prescalers. They consists of a series of D flip flops, where each D flip flop's inverted output is connected back to it's input, making it a divide by two circuit. If the input is fed into the clock signal of the circuit the output frequency will be half of input frequency. A circuit configuration of such a circuit, and its input output behavior is shown in figure 5.2.

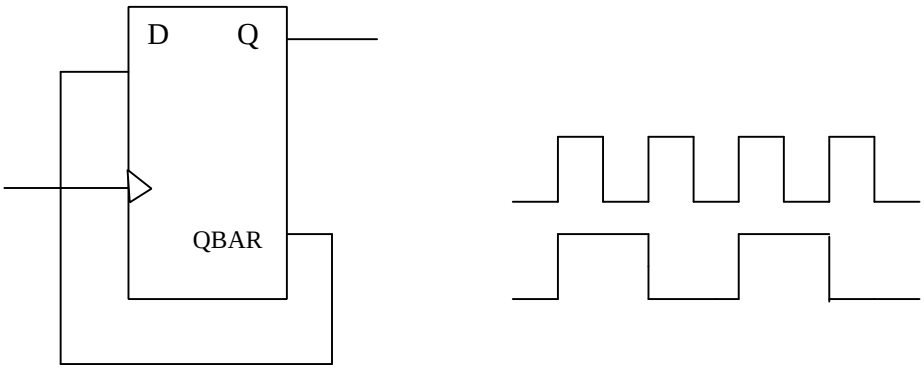


Figure-5.2 A Divide By 2 Prescaler

A very nice feature of this circuit is that the output is perfectly symmetrical square wave regardless of whether the input square wave is symmetrical or not. By cascading several D flip-flops in the same configuration, it is easy to make divide by 2^n circuit. The non-inverting output of one flip flop can be used as an input to the next flip flop to make it divide by 4 circuits. Thus to divide an input frequency by 64, the need is to have 6 D flip flops connected in this configuration. The simulation of Prescaler 64 was performed using ModelSim SE 5.5e and figure 5.4 shows the input and output waveform of the circuit. The circuit is performing as expected, one period of output square wave. Therefore output frequency is equal to the input frequency divided by the integer value 64 periods of the input square wave.

5.4 Divide by 65

This prescaler is more complicated to implement as compared to a divide by 64 prescaler. The reason is the odd number division. There are two ways to build this circuit, one is completely synchronous and the other is mixed. Since the first method is more complex as compare to second one, the second method is employed, which is asynchronous and synchronous mixed design. In this method the circuit is divided in two units. One unit is divide by 5 circuit and the other one is divide by 13. The output of the first unit will be fed in to second unit, and the whole circuit will be a divide by 65 prescaler circuit. These two circuits are basically ring counter with the number of states corresponding to division number. For example the need is to divide by 5, the ring counter will have five stages only and will count in a ring fashion. Same is true for a divide by 13 unit; it will have 13 stages and will also count in the ring fashion. The number of flip-flops required can be found from the number of stages. So, we need three D flip-flops for the divide by 5 circuit, as it has only five stages and four D flip-flops for the divide by 13 circuits. All the flip-flops are rising edge triggered. This figure 5.5 shows the result of divide by 65 prescaler simulated on Modesim SE 5.5e Tool.

5.5 Swallow Counter

A swallow counter in the proposed divider is a programmable down counter .For the proposed frequency divider, the swallow counter needs to count down from the loaded number to zero. The simulation result of the prescribed design of swallow counter for the proposed frequency divider is shown in figure 5.6. It repeats the counting down sequence from the same number until the loaded number is changed externally. It has the option of Load, which is required to enable the loading in to the swallow counter with the required division number. To generate 10 different frequencies, the range of number that needs to be loaded in the swallow counter is 8 to 18. Thus 5 flip-flops are required for the design. Based on the design of swallow counter up to 32 output frequencies can be synthesized with no extra hardware. The swallow counter is reset through the load value, whenever the main counter outputs a pulse and the counting sequence can be started again.

5.6 Main Counter

The main counter is also a frequency divider circuit, just like the prescaler. It divides by 48.The input of main counter is the output of prescaler unit. For the division number of 3080, it takes first 8 pulses of divide by 65 prescaler and remaining 40 pulses of divide by 64 units and yield only one pulse. This pulse is divided by 3080. The circuit is design with the same technique as used for the divide by 65 unit. It first divide the input by 16 and then the second unit divides this output by 3 yielding a total of divide by 48 output. The simulation of the main counter is shown in figure 5.7 which divides the input by 48.

5.7 Control Unit

The control logic of the proposed design consists of five blocks namely A, B, C, D, E as shown in the figure 5.3. Block E consists of swallow counter and OR gates. The output of block E serves as the enable input of MUX used in block B and block D. and input of MUX used in block A. The output of the MUX used in block A goes to divide by 64 prescaler where the input of divide by 65 prescaler comes from the output of ex-or gate of block A. At the very first cycle, the output of the OR gate is zero, so it enables the loading operation. As soon as the swallow counter is loaded with a value, the output of the OR gate becomes high at next clock cycle, thus disabling the Load operation of swallow counter. The high output of block E which serves as the control input of MUX in block D enables the clock input of the swallow counter. The output of block E also act as a control input for the MUX in block B. Whenever the output of block E is high, the divide by 65 prescalr unit is active and the input is divided by 65, and the main counter in block C receives the pulses from divide by 65 prescaler unit. At every pulse to the main counter, the swallow counter counts down and when it reaches zero, the output of block E become low. After that the divide by 64 prescaler unit is active and the clock input of the swallow counter is connected to the output of the main counter again since the output of main counter. There is no pulse, because it is still receiving pulses from the divide by 64 prescaler unit and has not yet received 48 pulses, the swallow counter is paused during this time. When the pulses to the Main Counter reached 48, the Main Counter outputs a pulse and the swallow counter is once again loaded with the division number & the sequence begins once again.

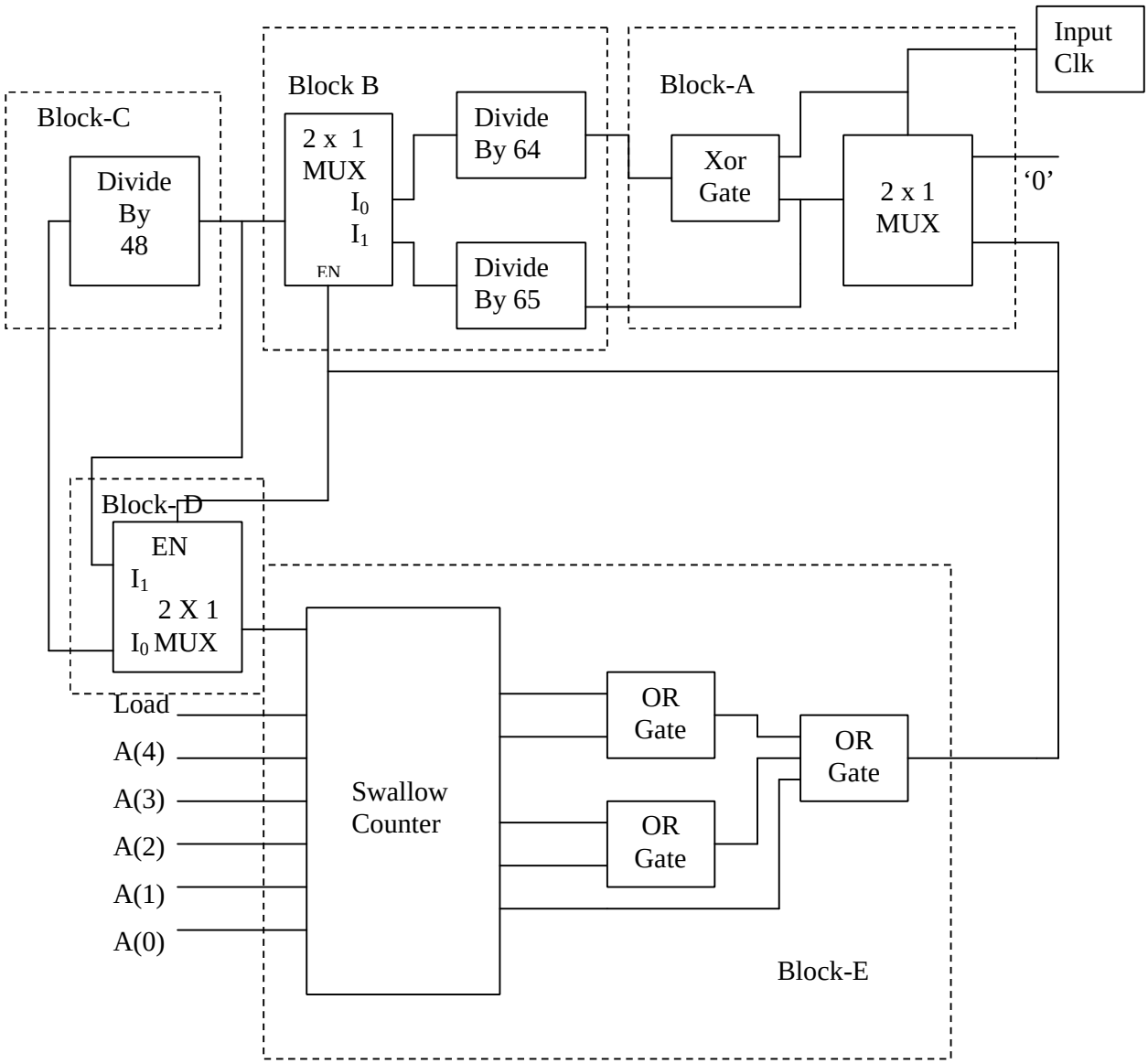


Figure-5.3 Logic and Block Diagram of Control Unit

5.8 Simulation Results

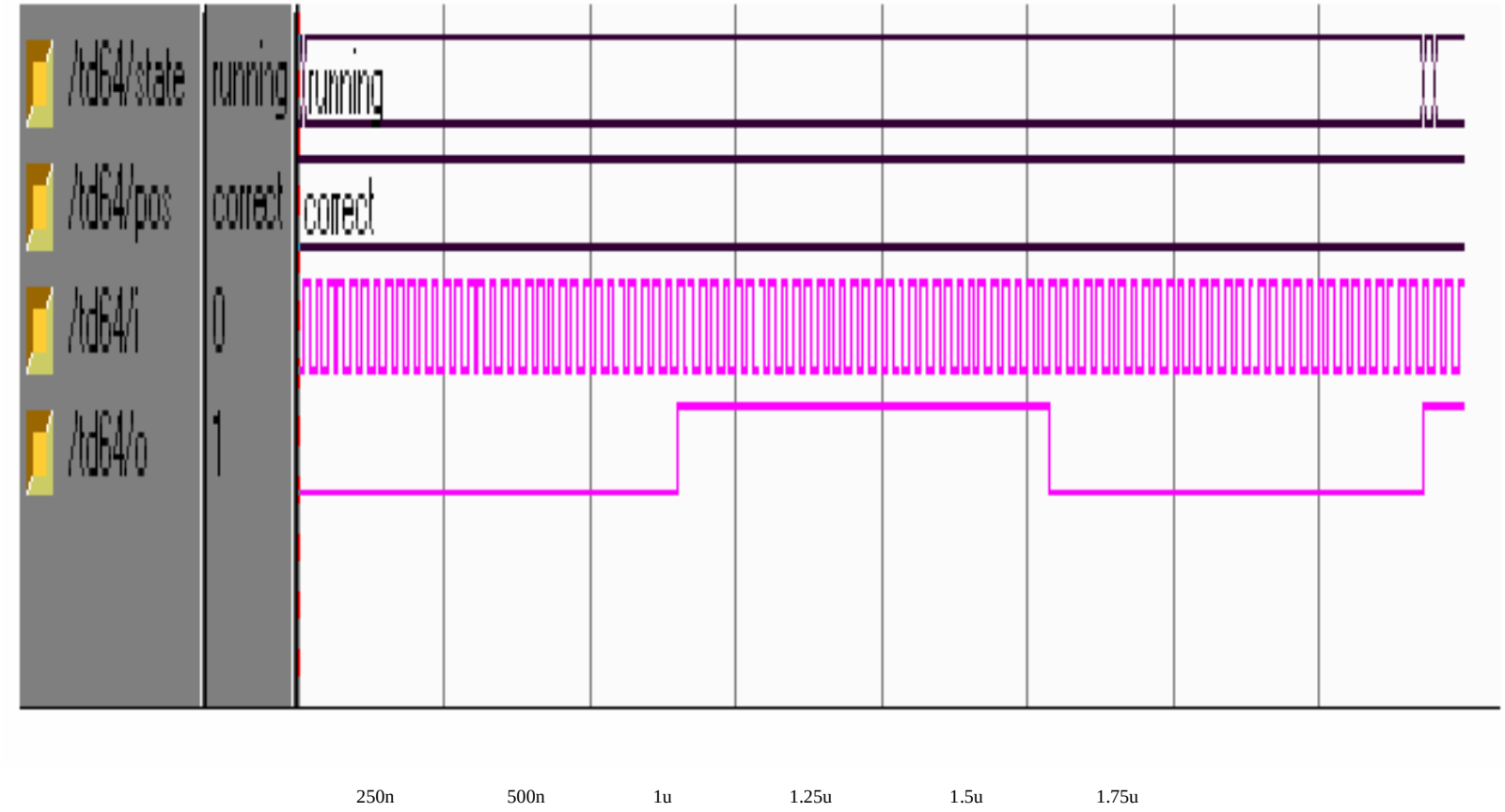


Figure-5.4 Divide By 64 Prescaler

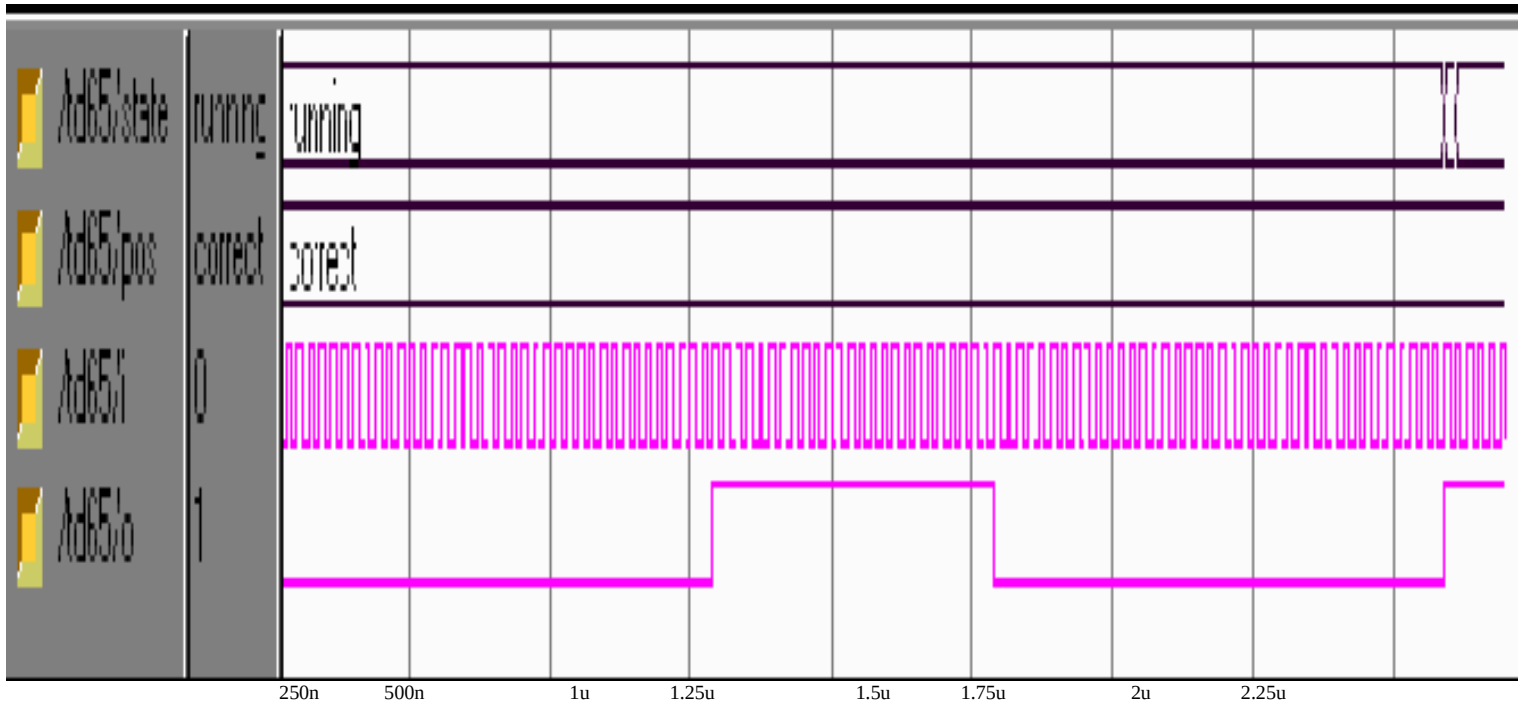


Figure-5.5 Divide By 65 Prescaler

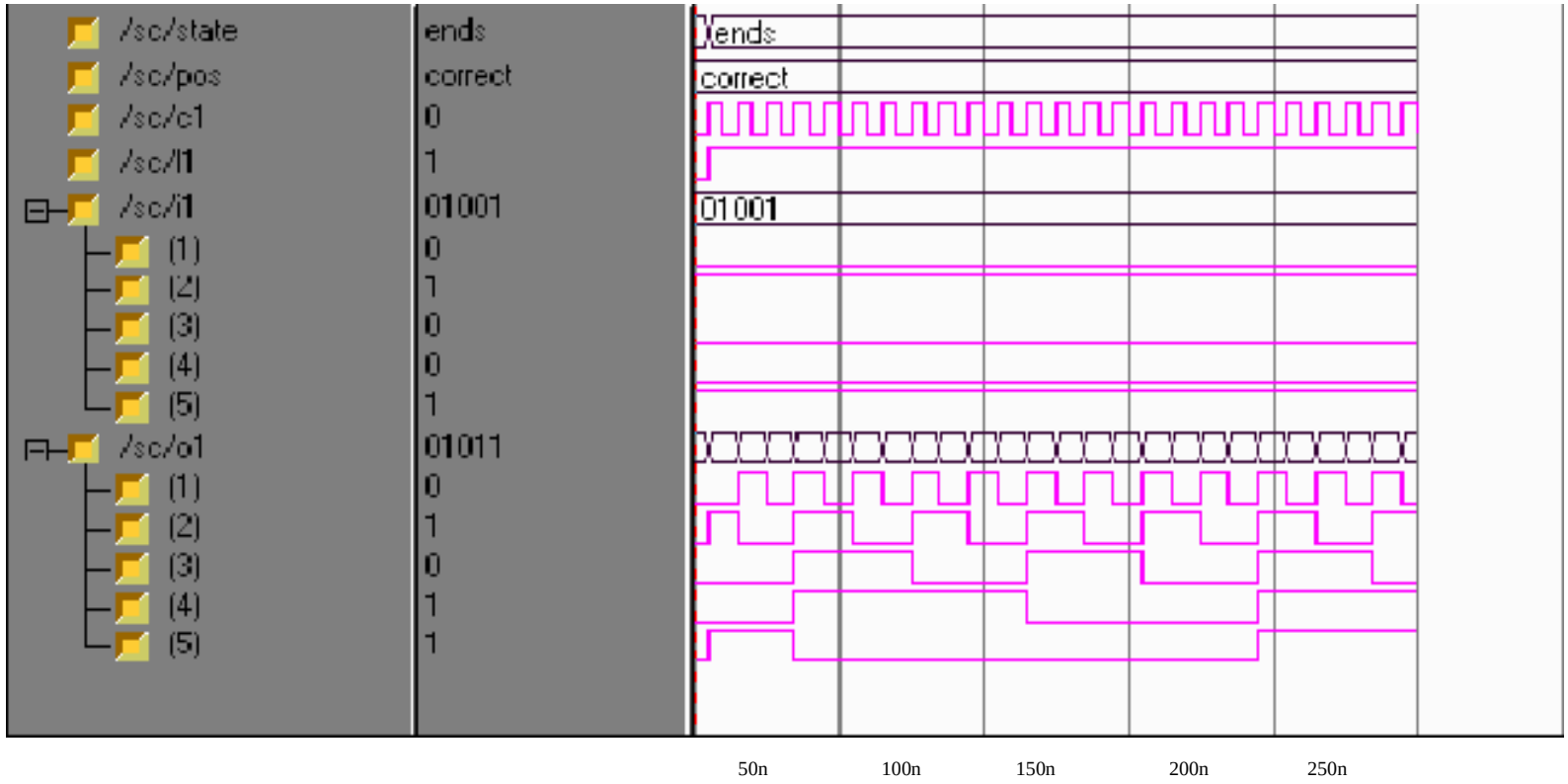


Figure-5.6 Swallow Counter With Input Value 8

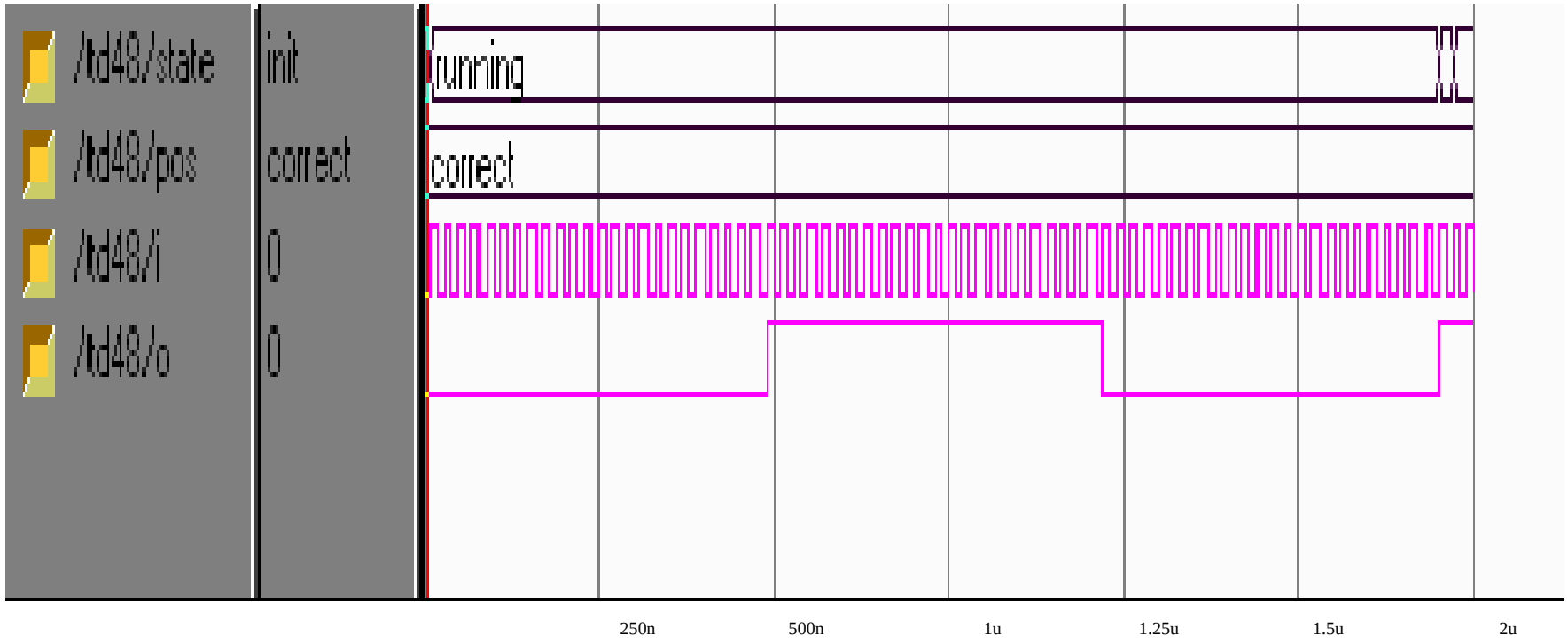


Figure-5.7 Main Counter

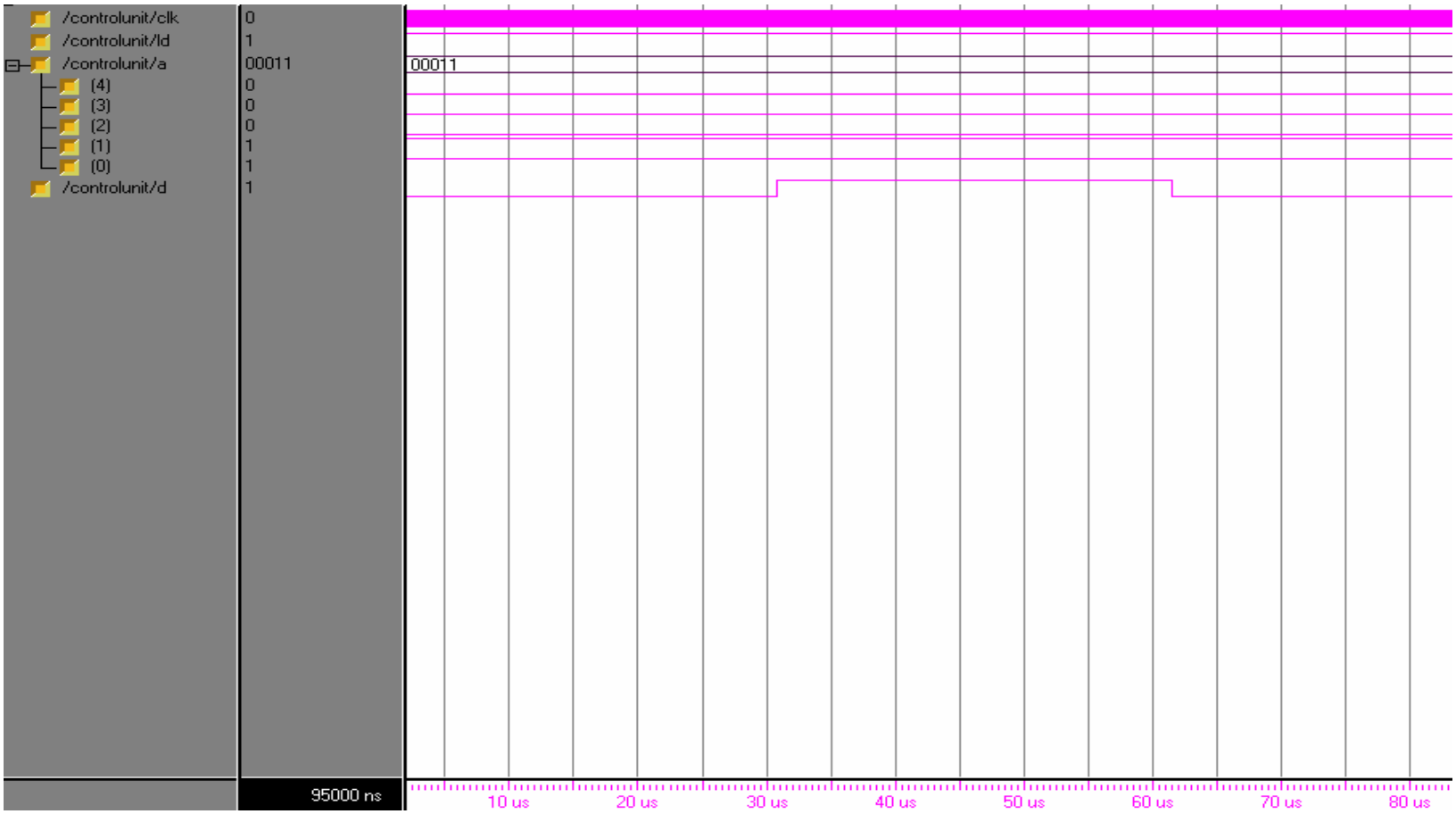


Figure-5.8 Control Unit By FSM

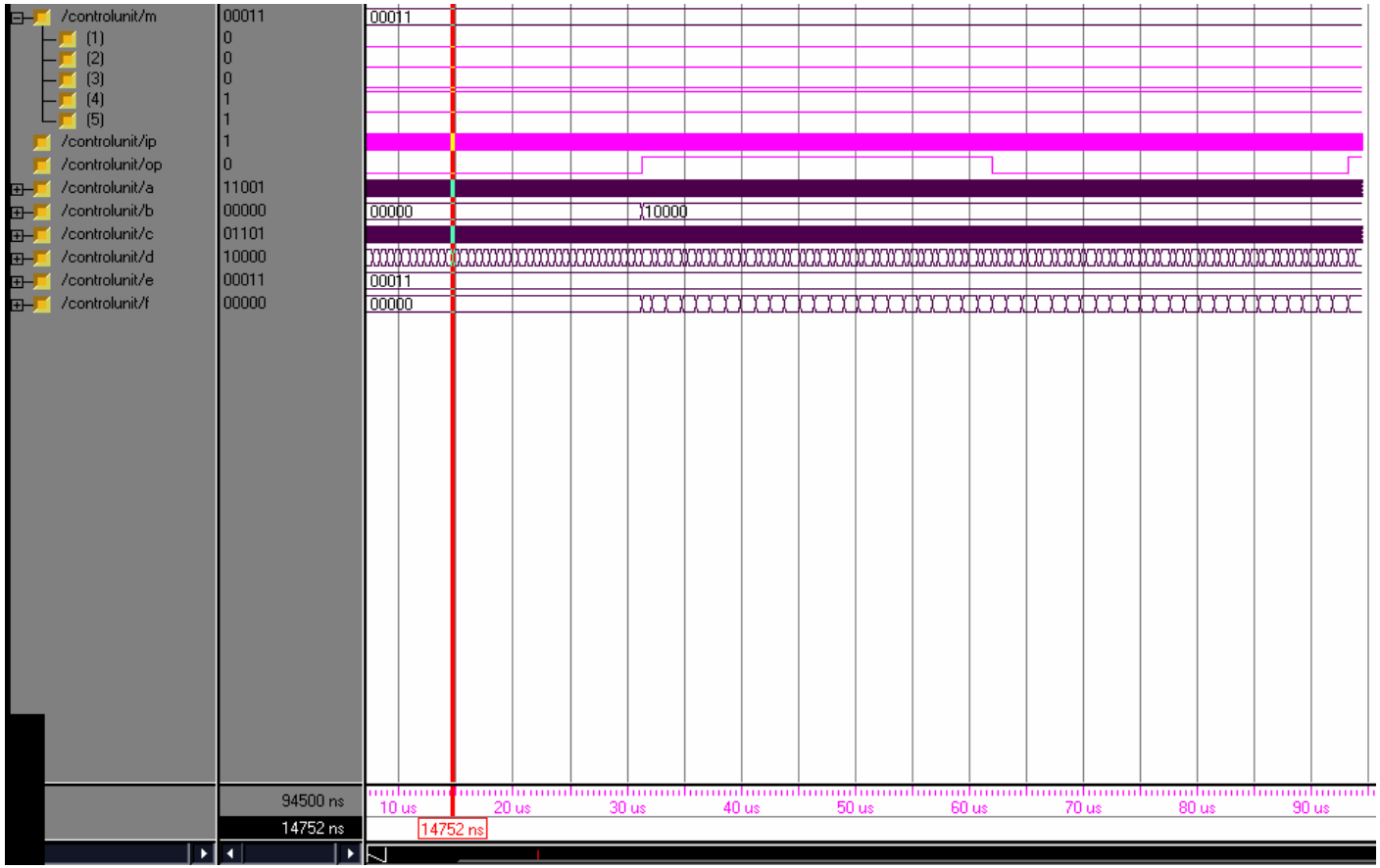


Figure-5.9 Structural Model Control Unit

Chapter 6

Results and Discussion

6.1 Results

To implement the frequency divider, an FSM is devised to write codes in VHDL. A frequency range of 929 MHz to 934 MHz and a step size (channel spacing) of 300 KHz is chosen as a design specifications. So now the division number (N) can be found as below

N = [output frequency / channel spacing] = [(929MHz – 932MHz)/300KHz]

= 3080 – 3090

All the possible intermediate frequencies are listed in table 6.2. Firstly behavioral simulation model is generated on Modelsim SE 5.5e then the designs are synthesized and the synthesis report is generated.

6.2 Synthesis Report

Synthesis Reports are generated on Xilinx ISE 6.1i tool. The device family used is Spartan 2E.

Table-6.1 Synthesis Report of FSM and Structure Modeling

	FSM	**Structural Model**
No. of Slices	52	20
No. of FFs	42	33
No. of input LUT	96	22
Min. Arrival Time Before CLK(ns)	6.235	4.079
Max CLK to O/P Delay(ns)	6.436	6.347
Maximum Frequency(MHz)	98.323	212.134

The maximum output frequency thus obtained is very low. Moreover, the design is realized by more hardware. So another approach is adopted to achieve the goal based on the Structural Modeling of the design. The modeling embodies the design of prescalers, counters and a control unit to control all the sub functional units.

Based on the structural modeling, codes are written in VHDL and finally structure in Xilinx ISE 6.1i. The synthesis report is summarized in table 6.1. It can be asserted from the synthesis reports of the two designs based on FSM and Structural Modeling that the Structural Model of the design overshadows the design based on FSM in terms of speed.

Table-6.2 The Range Of Possible Output Frequencies

Swallow counter	Divide By 'N'	Output Frequency(MHz)
0	3072	926.6
1	3073	926.9
2	3074	927.2
3	3075	927.5
4	3076	927.8
5	3077	928.1
6	3078	928.4
7	3079	928.7
8	3080	929.0
9	3081	929.3
10	3082	929.6
11	3083	929.9
12	3084	930.2
13	3085	930.5
14	3086	930.8
15	3087	931.1
16	3088	931.4
17	3089	931.7
18	3090	932.0
19	3091	932.3
20	3092	932.6
21	3093	932.9
22	3094	933.2

23	3095	933.5
24	3096	933.8
25	3097	934.1
26	3098	934.4
27	3099	934.7
28	3100	935.0
29	3101	935.3
30	3102	935.6
31	3103	935.9

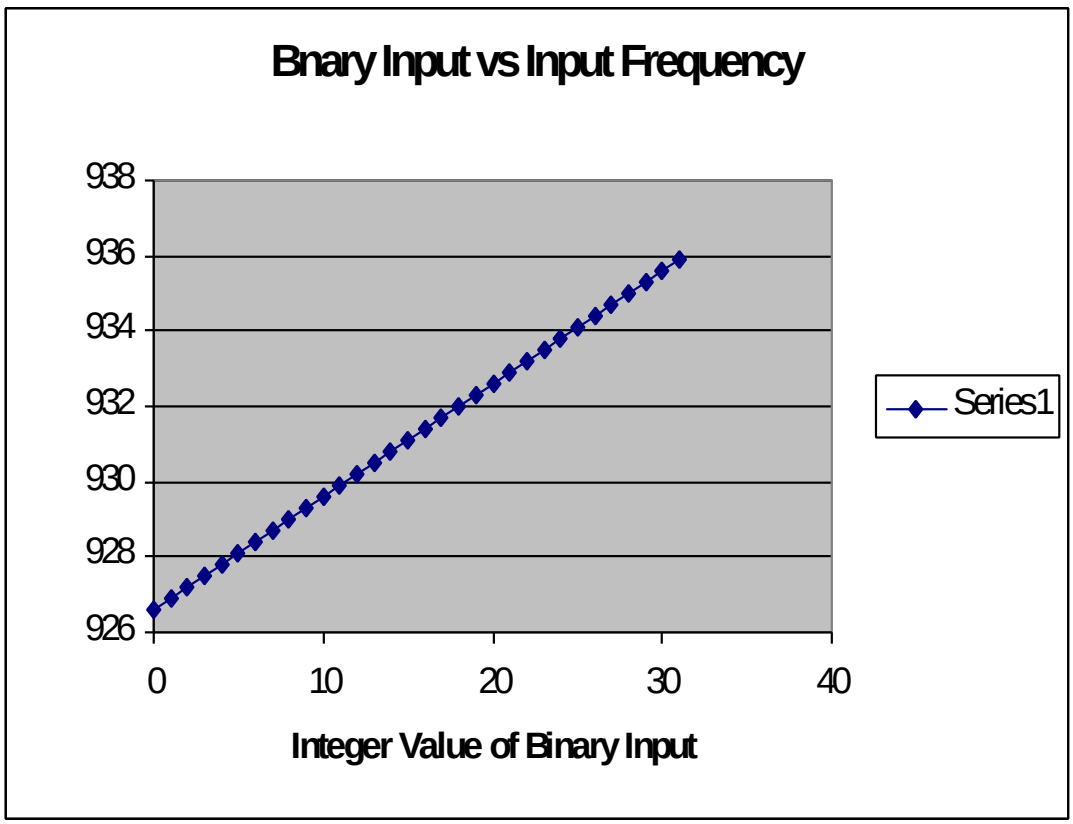


Figure-6.1 Binary Input vs Input Frequency

Chapter 7

Conclusion and Future work

Frequency Synthesizers using PLL incorporate high speed frequency dividers. In this work, frequency divider is first implemented according to FSM and later by Structural Modeling. The maximum frequency obtained from the design based on FSM is very low (98.323 MHz) whereas the maximum frequency observed in the design process through Structural Modeling is very high (212.134 MHz) and hence obviates the need of Structural Modeling of the frequency dividers in the design process. By varying the value of swallow counter, a large range of integer value can be obtained and so is the output frequency. The use of Structural Modeling in design overcomes the problem of programmable divider in high speed designs. The designs are simulated on Modelsim SE 5.5e while the synthesis reports are generated on Xilinx ISE 6.1i. It can be inferred from the synthesis report of the two designs that the design based on Structural Modeling is superior then the design based on FSM.

There are some design issues that could be explored and incorporated in future.

- The design only optimized speed, but other parameters like power dissipation and effect of temperature can be dealt with, in future.
- It is worth while to compare results with full custom ASICs to visualize the effect of device technology on the area and performance cost of architectural feature support.

References

[1] A. Hajimiri, T. Lee, “A General Theory of Phase Noise in Electrical Oscillators,” IEEE Journal of Solid-State Circuits, vol 33, no. 2, pp. 179-194, 1998.

[2] A. Przedpelski, “PLL Primer - Part 1”, RF design, Englewood, CO, pp. 270-271, 1983

[3] Altera, “Programmable logic device family data book”, 2000.

[4] B. Razavi, “A Study of Phase Noise in CMOS Oscillators,” IEEE Journal Solid-State Circuits, vol 31, no. 3, pp. 331-343, 1996.

[5] B. Razavi, “Analysis, Modeling, and Simulation of Phase Noise in Monolithic Voltage-Controlled Oscillators,” Proceedings of the Custom Integrated Circuits Conference, 1995

[6] D. Gavin, “A PLL Synthesizer Utilizing a New GaAs Phase Frequency Comparator”, GigaBit Logic Data Book.

[7] D. H. Wolaver, Phase-Locked Loop Circuit Design, Prentice Hall, New Jersey, 1991.

[8] H. Lee, and M. Flynn, “Coarse Grained Carry Architecture for FPGA”, Proceedings of the ACM/SIGDA International Symposium on FPGAs, Monterey, 2000.

[9] J. Craninckx, M.S.J. Steyaert, “A 1.75-GHz/3-V dual-modulus divide-by-128/129 prescaler in 0.7-µm CMOS”,IEEE Journal of Solid-State Circuits, vol 38, no. 6 pp. 866- 874, 2003

[10] J.M.C. Wong, V.S.L. Cheung, H.C. Luong, “A 1-V 2.5-mW 5.2-GHz frequency divider in a 0.35-/spl mu/m CMOS process,” IEEE Journal of Solid-State Circuits, vol 38, no. 10, pp. 1643 -1650, 2003

[11] J. Rose et al, “Architecture of Field Programmable Gate Arrays,” Proceedings
of IEEE, vol 81, pp. 1013-1029, 1993.

[12] J. Rutman, “Characterization of Phase and Frequency Instabilities in Precision Frequency Sources Fifteen Years of Progress,” IEEE, vol 66, pp. 1048-1076, 1978

[13] Mark McClure, “Residual Phase Noise of Digital Frequency Dividers,” Microwave Journal, pp. 124-130, 1992.

[14] Mehmet Soyuer, Robert Meyer, “Frequency Limitations of a Conventional Phase-Frequency Detector,” IEEE J. Solid-State Circuits, vol. SC-25, no. 4, pp. 1019-1022, 1990.

[15] Mijuskovic, D. Bayer, M.; Chomicz, T. Garg, “Cell-based fully integrated CMOS frequency synthesizers,” IEEE Journal of Solid-State Circuits, vol 29, no.3, pp. 271-279, 1994.

[16] Navid Foroudi, “A High-Speed CMOS Dual-Modulus Frequency Divider for Mobile Radio Frequency Synthesizers,” Thesis Department of electronics, Carleton University Ottawa, Canada, 1991.

[17] Neil H. E. weste and Kamran Eshraghian. “Principles of CMOS VLSI Design A System Perspective” Addison Wesley, Second edition, 1992.

[18] Nhat M. Nguyen, Robert G. Meyer, “A 1.8 GHz Monolithic LC Voltage Controlled Oscillator,” International Solid-State Circuits Conference, 1992.

[19] S.D. Haynes et al, “Configurable Multiplier Blocks for use within a FPGA”,IEEE Transaction on Computers, vol 3, no.1, pp. 638-639, 1998.

[20] S. Pellerano, S. Levantino, C. Samori, A.L. Lacaita, “A 13.5-mW 5-GHz frequency synthesizer with dynamic-logic frequency divider” ,IEEE Journal of Solid-State Circuits, vol 39, no. 2, pp 372-379 2004

[21] S. Shimizu, K. Yoshihara, T. terada, K. Ishida, Y. Kitaura, and C. Takubo, “Delta-Sigma Modulation in Fractional-N Frequency Synthesis,” IEEE J.Solid-State Circuits, vol. SC-25, no. 2, pp. 539-545, 1990.

[22] S.D. Haynes et al, “Flexible Reconfigurable Multiplier Blocks Suitable for Enhancing the Architecture of FPGAs”, Proceedings of Custom Integrated Circuit Conference, San Diego, California, pp. 191-194, 1999.

[23] T. Riley, M. Copeland, and T. Kwasniewski, “Delta-Sigma Modulation in Fractional-N Frequency Synthesis,” IEEE J. Solid-State Circuits, vol 28, no. 5, pp. 553-559, 1989.

[24] Ulrich L. Rohde, Digital PLL Frequency Synthesizers: Theory and Design, Prentice Hall, New Jersey, 1983.

[25] V. Reinhart, K. Gould, K. McNab, and M. Bustamante, “A Short Survey of Frequency Synthesizer Techniques,” Proceedings of the 40th Annual Frequency Control Symposium, Philadelphia, pp. 355-65, 1986.

[26] Xilinx Corporation Inc, “Programmable logic Databook”, 2000

[27] Z. Lao, J. Jensen, K. Guinn, M. Sokolich, “1.3 V supply voltage 38 GHz static frequency divider” Electronics Letters, vol 40, no. 5, Pp 224-230, 2004.

[28] Z. Shu, K.L. Lee, B.H. Leung, “1.3 GHz Ring-Oscillator-Based CMOS Frequency Synthesizer With a Fractional Divider Dual-PLL Architecture”, IEEE Journal of Solid-State Circuits, vol 39, no. 3, pp 423-430, 2004

[29] www.altera.com

[30] www.xilinx.com

HDL Synthesis Report of FSM

Design Statistics
IOs : 8

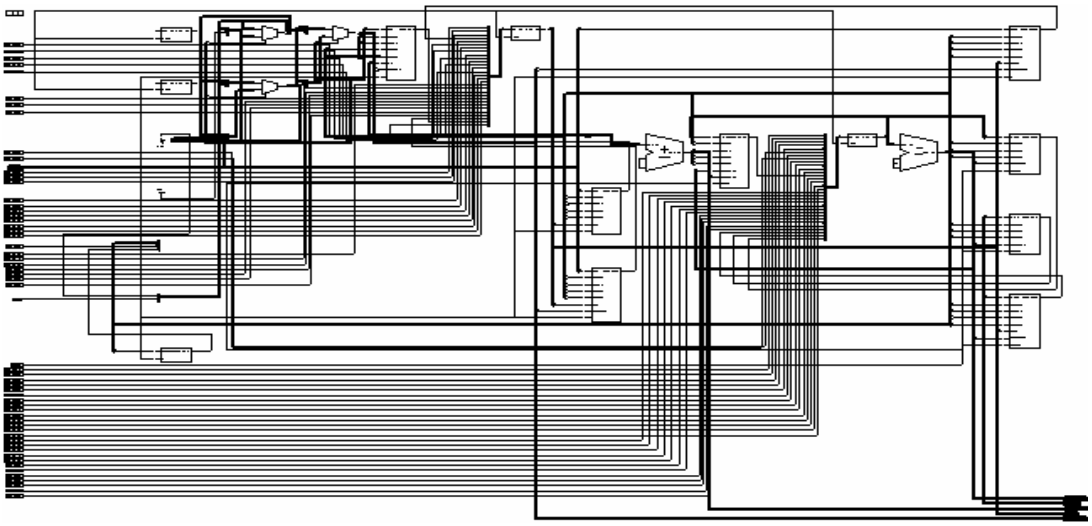
Macro Statistics:
Multiplexers : 3
2-to-1 multiplexer : 3
Adders/Subtractors : 2
20-bit adder :1
32-bit subtractor :1

Cell Usage
BELS : 161
GND : 1
LUT1 : 32
LUT2 : 4
LUT3 : 4
LUT3_L : 5
LUT4 : 39
LUT4_L : 12
MUXCY : 31
VCC : 1
XORCY : 32
FlipFlops/Latches : 42
LD : 42
Clock Buffers : 1
BUFGP : 1
IO Buffers : 7
IBUF : 6
OBUF : 1

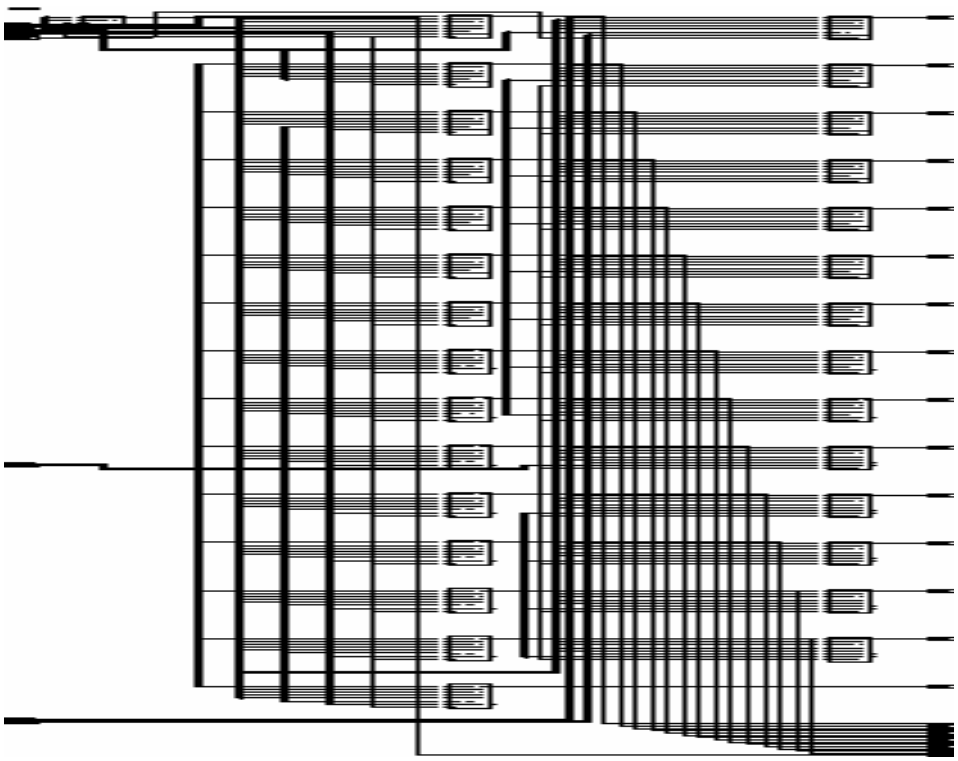
HDL synthesis Report of Structural Model

Macro Statistics
Registers : 48
1-bit register : 48
Multiplexers : 3
2-to-1 multiplexer : 3
Xors : 8
1-bit xor2 : 8

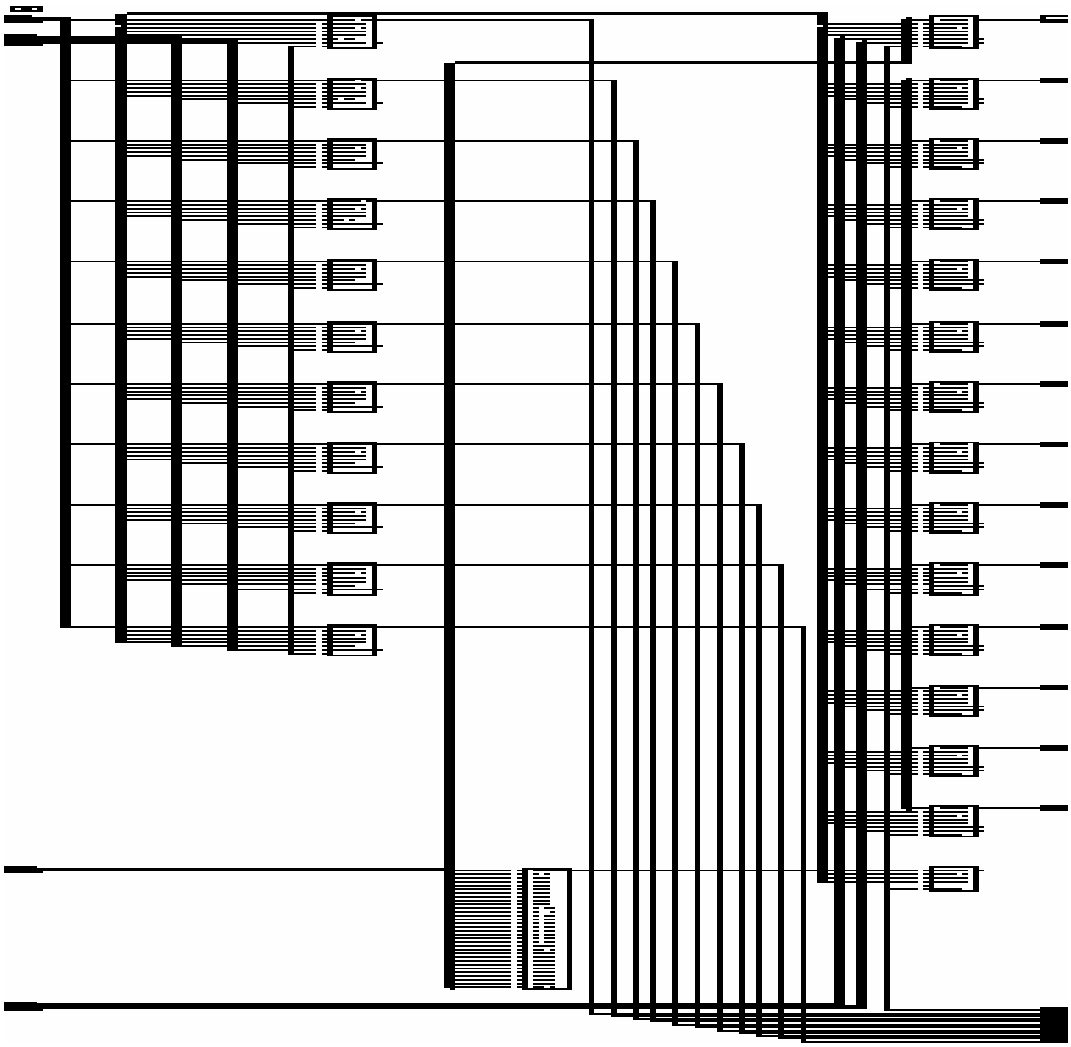
RTL Schematic of FSM



RTL-1 Schematic Frequency Divider Using FSM



RTL-2 Schematic Frequency Divider Using FSM



RTL-3 Schematic of Frequency Divider using FSM

VHDL CODE OF CONTROL UNIT

```
entity controlunit is
port ( M : in bit_vector ( 1 to 5); clk : inout bit; op : inout bit; ld1
: inout bit);
end controlunit;
architecture controlunit of controlunit is
component eg
port (p1 : in bit_vector(1 to 5); ld: in bit; clk : in bit; u : out bit);
end component;
component cg
port (ip48 : in  bit ; op48 : out bit);
end component;
component dg
port (I1,I2,s1 : in bit; y:out bit);
end component;
component ag
port (a,b: in bit; c,z: inout bit);
end component;
component bg
port(t64,t65,s : in bit; t4 : out bit);
end component;
signal w1,w2,w3,w4,w5 : bit:= '0' ;

begin
lw1: ag port map (clk,w5,w1,w2);
lw2: bg port map (w1,w2,w5,w3);
lw3: cg port map (w3,op);
lw4: dg port map (w3,op,w5,w4);
lw5: eg port map (m,ld1,w4,w5);
end controlunit;

entity ag is
port (a,b: in bit; c,z : inout bit);
end ag;
architecture ag of ag is
component dg
port (I1,I2,s1 : in bit; y:out bit);
end component;
component xorg
port(j1,k1 : in bit ; l1 : out bit);
end component;
signal n : bit;
```

```
begin
n <= '0';
p1: dg port map(n,b,a,z);
p2: xorg port map (a,z,c);

end ag;

entity dg is
port (I1,I2,s1 : in bit; y:out bit);
end dg;
architecture dg of dg is
begin
process(I1,I2,s1)
begin
if ( s1 = '0') then
y <= I1;
else
y <= I2;
end if;
end process;
end dg;

entity bg is
port(t64,t65,s : in bit; t4 : out bit);
end bg;
architecture bg of bg is
 component divideby65
port (ip65 : in  bit ; op65 : out bit);
end component;
component divideby64
port (ip64 : in  bit ; op64 : out bit);
end component;
component dg
port (I1,I2,s1 : in bit; y:out bit);
end component;
signal g,h : bit;
begin
m1 : divideby64 port map (t64,g);
m2 : divideby65 port map (t65,h);
m3:  dg port map (g,h,s,t4);
end bg;

entity eg is
port (p1 : in bit_vector(1 to 5); ld: in bit; clk : in bit; u : out bit);
end eg;
architecture eg of eg is
```

```
component swallowcounter
port (sck : in bit; ld : in bit; i : in bit_vector ( 1 to 5);
o : out bit_vector ( 1 to 5));
end component;
component or2g
port(l1,m1 : in bit ; Y1: out bit);
end component;
component or3g
port(l2,m2,n2: in bit ; Y2: out bit) ;
end component;
signal k : bit_vector(1 to 5);
signal k1,k2 : bit;
begin
z1: swallowcounter port map (clk,ld,p1,k);
z2: or2g port map (k(1),k(2),k1);
z3: or2g port map (k(3),k(4),k2);
z4: or3g port map(k1,k2,k(5),u);
end eg;


entity swallowcounter is
port (sck : in bit; ld : in bit; i : in bit_vector ( 1 to 5);
o : out bit_vector ( 1 to 5));
end swallowcounter;
architecture swallowcounter of swallowcounter is
component dff1
port (d,clk : in bit  ; q,qbar : out bit);
end component;
component xnorg
port(j2,k2 : in bit ; l2 : out bit);
end component;
component and2g
port (a1,b1 : in bit ; c1: out bit);
end component;
component or2g
port(l1,m1 : in bit ; Y1: out bit);
end component;
signal s,t,u,v : bit_vector ( 1 to 5);
signal q: bit_vector ( 1 to 5) := "11111";
signal qbar: bit_vector ( 1 to 5) := "00000";
signal s1,s2,s3 : bit;
signal p :bit_vector ( 1 to 10);
signal w,z:bit_vector ( 1 to 3);
begin
s <= i;
s3 <= not ld;
```

```
s2 <= ld;
s1 <= sck;
l1: and2g port map (s3, s(1), p(1));
l2: and2g port map (s3, s(2) ,p(3));
l3: and2g port map (s3, s(3), p(5));
l4: and2g port map (s3, s(4), p(7));
l5: and2g port map (s3 ,s(5), p(9));
l6: or2g   port map (p(1),p(2),t(1));
l7: or2g   port map (p(3),p(4),t(2));
l8: or2g   port map (p(5),p(6),t(3));
l9: or2g   port map (p(7),p(8),t(4));
l10: or2g   port map (p(9),p(10),t(5));
m1: and2g port map (s2 ,u(1), p(2));
m2: and2g port map (s2 ,u(2), p(4));
m3: and2g port map (s2 ,u(3), p(6));
m4: and2g port map (s2 ,u(4), p(8));
m5: and2g port map (s2 ,u(5), p(10));
u(1) <= v(1);
o1: and2g port map (v(1),v(2),w(1));
o2: and2g port map (w(1),v(3),w(2));
o3: and2g port map (w(2),v(4),w(3));
n1: xnorg port map (v(1),v(2),u(2));
n2: xnorg port map (w(1),v(3),u(3));
n3: xnorg port map (w(2),v(4),u(4));
n4: xnorg port map(w(3),v(5),u(5));

p1: dff1 port map (t(1),s1,q(1),v(1));
p2: dff1 port map (t(2),s1,q(2),v(2));
p3: dff1 port map (t(3),s1,q(3),v(3));
p4: dff1 port map (t(4),s1,q(4),v(4));
p5: dff1 port map (t(5),s1,q(5),v(5));
o <= q;
end swallowcounter;

entity cg is
port (ip48 : in  bit ; op48 : out bit);
end cg ;
architecture cg of cg is
component divideby3
port (ip3 : in  bit ; op3 : out bit);
end component;
component divideby16
port (ip16 : in  bit ; op16 : out bit);
end component;
signal s1,s2,s3:bit;
begin
```

```
s1 <= ip48;
l1: divideby3 port map (s1,s2);
l2: divideby16 port map (s2,s3);
op48 <= s3;
end cg;

entity divideby3 is
port (ip3 : in  bit ; op3 : out bit);
end divideby3;
architecture divideby3 of divideby3 is
component dff1
port (d,clk : in bit  ; q,qbar : out bit);
end component;
component xnorg
port(j2,k2 : in bit ; l2 : out bit);
end component;
signal q: bit_vector ( 1 to 2 ):= "11";
signal qbar: bit_vector ( 1 to 2 ):= "00";
signal s: bit_vector ( 1 to 2 );
signal s1,s2 : bit;
begin
s <= q;
s1 <= ip3;
l1 : dff1 port map (s(2),s1,q(1),qbar(1));
l2: dff1 port map (s2,s1,q(2),qbar(2));
l3 : xnorg port map (q(1),q(2),s2);
op3 <= q(2);
end divideby3;

entity divideby16 is
port (ip16 : in  bit ; op16 : out bit);
end divideby16;
architecture divideby16 of divideby16 is
component dff1
port (d,clk : in bit  ; q,qbar : out bit);
end component;
signal q: bit_vector ( 1 to 4 ):= "1111";
signal qbar: bit_vector ( 1 to 4 ):= "0000";
signal s: bit_vector ( 1 to 4 );
signal s1,s2,s3 : bit;
begin
s <= qbar;
s1 <= ip16;
l1: dff1 port map (s(1),s1,q(1),qbar(1));
l2: dff1 port map (s(2),qbar(1),q(2),qbar(2));
```

```
l3: dff1 port map (s(3),qbar(2),q(3),qbar(3));
l4: dff1 port map (s(4),qbar(3),q(4),qbar(4));
op16 <= q(4);
end divideby16;

entity divideby65 is
port (ip65 : in  bit ; op65 : out bit);
end divideby65;
architecture divideby65 of divideby65 is
component divideby5
port (ip5 : in bit; op5 : out bit);
end component;
component divideby13
port (ip13 : in bit; op13 : out bit);
end component;
signal s1,s2,s3 : bit;
begin
s1 <= ip65;
m1: divideby5 port map (s1,s2);
m2: divideby13 port map (s2,s3);
op65 <= s3;
end divideby65;

entity divideby5 is
port (ip5 : in bit; op5 : out bit);
end divideby5;
architecture divideby5 of divideby5 is
component dff1
port (d,clk : in bit  ; q,qbar : out bit);
end component;
component  and2g
port (a1,b1 : in bit ; c1: out bit);
end component;
component xorg
port(j1,k1 : in bit ; l1 : out bit);
end component ;
signal q : bit_vector ( 1 to 3) := "000";
signal qbar : bit_vector (1 to 3) := "111";
signal s1,s2,s3,s4,s5 : bit;
begin
s4 <= ip5;
l1: and2g port map (q(2),q(3),s1);
l2: and2g port map (qbar(1),qbar(3),s3);
l3: xorg port map (q(2),q(3),s2);
l4: dff1 port map (s1,s4,q(1),qbar(1));
```

```
l5: dff1 port map (s2,s4,q(2),qbar(2));
l6: dff1 port map (s3,s4,q(3),qbar(3));
op5 <= q(1);
end divideby5;

entity divideby13 is
port (ip13 : in bit; op13 : out bit);
end divideby13;
architecture divideby13 of divideby13 is
component da
port (da1,db1,dc1,dd1 : in bit ; yda : out bit);
end component;
component db
port (da2,db2,dc2,dd2 : in bit ; ydb : out bit);
end component;
component dc
port (dc3,dd3 : in bit ; ydc : out bit);
end component;
component dd
port (da4,db4,dc4,dd4 : in bit ; ydd : out bit);
end component;
component dff1
port (d,clk : in bit  ; q,qbar : out bit);
end component;
signal q : bit_vector(1 to 4 ) := "0000";
signal qbar : bit_vector(1 to 4 ) := "1111";
signal y: bit_vector(1 to 4 );
signal ip131 : bit;
begin
ip131 <= ip13;
l5: da port map (q(1),q(2),q(3),q(4),y(1));
l6: db port map (q(1),q(2),q(3),q(4),y(2));
l7: dc port map (q(3),q(4),y(3));
l8: dd port map (q(1),q(2),q(3),q(4),y(4));
l1: dff1 port map (y(1),ip131,q(1),qbar(1));
l2: dff1 port map (y(2),ip131,q(2),qbar(2));
l3: dff1 port map (y(3),ip131,q(3),qbar(3));
l4: dff1 port map (y(4),ip131,q(4),qbar(4));
op13 <= q(1);
end divideby13;

entity da is
port (da1,db1,dc1,dd1 : in bit ; yda : out bit);
end da;
architecture da of da is
begin
```

```
yda <= (da1 and (not db1)) or (db1 and dc1 and dd1 );
end da;

entity db is
port (da2,db2,dc2,dd2 : in bit ; ydb : out bit);
end db;
architecture db of db is
begin
ydb <= ((not db2) and dc2 and dd2) or ((not da2) and db2 and
(not dc2)) or ((not da2) and db2 and (not dd2));
end db;

entity dc is
port (dc3,dd3 : in bit ; ydc : out bit);
end dc;
architecture dc of dc is
begin
ydc <= dc3 xor dd3;
end dc;

entity dd is
port (da4,db4,dc4,dd4 : in bit ; ydd : out bit);
end dd;
architecture dd of dd is
begin
ydd <= (dc4 and (not dd4)) or ((not da4) and (not dd4)) or ((not
db4) and (not dd4));
end dd;

entity divideby64 is
port (ip64 : in  bit ; op64 : out bit);
end divideby64;
architecture divideby64 of divideby64 is
component dff1
port (d,clk : in bit  ; q,qbar : out bit);
end component;
signal qbar64:  bit_vector (1 to 6 ) := "111111";
signal q64: bit_vector (1 to 6 ) := "000000";
signal s : bit_vector (1 to 6 );
signal ip641 : bit;
begin
s <= qbar64;
ip641 <= ip64;
m1:  dff1 port map (s(1) ,ip641, q64(1), qbar64(1));
m2:  dff1 port map (s(2), qbar64(1),q64(2), qbar64(2) );
m3:  dff1 port map (s(3), qbar64(2), q64(3), qbar64(3));
```

```
m4:  dff1 port map (s(4), qbar64(3) , q64(4), qbar64(4) );
m5:  dff1 port map (s(5), qbar64(4) , q64(5), qbar64(5) );
m6:  dff1 port map (s(6), qbar64(5) , op64, qbar64(6) );
end divideby64;
entity dff1 is
port (d,clk : in bit  ; q,qbar : out bit);
end dff1;
architecture dff1 of dff1 is
begin
process (d,clk)
begin
if (clk = '1' and clk'event ) then
q <= d;
qbar <= not d;
end if;
end process;
end dff1;

entity and2g is
port (a1,b1 : in bit ; c1: out bit);
end and2g;
architecture and2g of and2g is
begin
c1<= a1 and b1;
end and2g;

entity and3g is
port (a2,b2,c2 : in bit ; d2: out bit);
end and3g;
architecture and3g of and3g is
begin
d2<= a2 and b2 and c2;
end and3g;

entity and4g is
port (a3,b3,c3,d3 : in bit ; e3: out bit);
end and4g;
architecture and4g of and4g is
begin
e3<= a3 and b3 and c3 and d3;
end and4g;

entity or2g is
port(l1,m1 : in bit ; Y1: out bit);
end or2g;
```

```
architecture or2g of or2g is
begin
y1<= l1 or m1;
end or2g;

entity or3g is
port(l2,m2,n2: in bit ; Y2: out bit) ;
end or3g;
architecture or3g of or3g is
begin
y2<= l2 or m2 or n2 ;
end or3g;

entity or4g is
port(l3,m3,n3,O3: in bit ; Y3: out bit);
end or4g;
architecture or4g of or4g is
begin
y3<= l3 or m3 or n3 or o3 ;
end or4g;

entity or5g is
port(l4,m4,n4,O4,p4: in bit ; Y4: out bit);
end or5g;
architecture or5g of or5g is
begin
y4<= l4 or m4 or n4 or o4 or p4 ;
end or5g;

entity nand2g is
port(a11,b11 : in bit ; c11: out bit);
end nand2g;
architecture nand2g of nand2g is
begin
c11<= a11 nand b11;
end nand2g;

entity nand3g is
port(a12,b12,c12 : in bit ; d12: out bit);
end nand3g;
architecture nand3g of nand3g is
begin
d12<= not (a12 and b12 and c12);
end nand3g;
```

```
entity nand4g is
port(a13,b13,c13,d13 : in bit ; e13: out bit);
end nand4g;
architecture nand4g of nand4g is
begin
e13<= not (a13 and b13 and c13 and d13);
end nand4g;

entity nor2g is
port(l11,m11 : in bit ; Y11: out bit);
end nor2g;
architecture nor2g of nor2g is
begin
y11<= l11 nor m11;
end nor2g;

entity nor3g is
port(l12,m12,n12: in bit ; Y12: out bit);
end nor3g;
architecture nor3g of nor3g is
begin
y12<= not (l12 or m12 or n12) ;
end nor3g;

entity nor4g is
port(l13,m13,n13,O13: in bit ; Y13: out bit);
end nor4g;
architecture nor4g of nor4g is
begin
y13 <= not (l13 or m13 or n13 or o13 );
end nor4g;

entity xorg  is
port(j1,k1 : in bit ; l1 : out bit);
end xorg ;
architecture xorg  of xorg  is
begin
l1 <= j1 XOR k1;
end xorg;

entity xnorg  is
port(j2,k2 : in bit ; l2 : out bit);
end xnorg ;
architecture xnorg  of xnorg  is
begin
```

```
l2 <= not (j2 XOR k2);
end xnorg;

entity notg is
port (a: in bit;b : out bit);
end notg;
architecture notg of notg is
begin
b <= not a;
end notg;
```

List of Publications

1. Anish Lal, Varinder Deepak, “Implementation of FSM using DNA”, National Conference on Bioinformatics Computing (NCBC’05), TIET, Patiala, Punjab, March 18-19, 2005 (Accepted).
2. Anish Lal, Varinder Deepak, “A Novel Design of High Speed Arithmetic Logic Unit using ROMs**”,** National Conference on Emerging Principles and Practices of Computer Science and Information Technology (EPPCSIT’06), GNDE, Ludhiana, Punjab, August 18-19, 2006 (Communicated).