5-2003

Size, Speed, and Power Analysis for Application-Specific Integrated Circuits Using Synthesis

Chung Ku
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Chung Ku entitled "Size, Speed, and Power Analysis for Application-Specific Integrated Circuits Using Synthesis." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald Bouldin, Major Professor

We have read this thesis and recommend its acceptance:

Paul Cilly, Chandra Tan

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Chung Ku entitled "Size, Speed, and Power Analysis for Application-Specific Integrated Circuits Using Synthesis." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald Bouldin

Major Professor

We have read this thesis and
recommend its acceptance:

Paul Crilly

Chandra Tan

Accepted for the Council:

Anne Mayhew

Vice Provost and Dean of Graduate Studies

Original signatures are on file with official student records.

Size, Speed, and Power Analysis for Application-Specific Integrated Circuits Using Synthesis

A Thesis
Presented for the
Master of Science
Degree

The University of Tennessee, Knoxville

Chung Ku
May 2003

Dedicated to my parents:
Kui-Din Ku and Chiu-Mei Chang,
and my lovely girl friend, Ya-wen Yang who
have motivated and supported me
throughout my college career.

Acknowledgements

I sincerely appreciate the many people who offered me guidance and support during the course of my Master's program. First of all, I would like to thank my thesis committee, Dr. Bouldin, Dr. Crilly, and Dr. Tan, for taking time to review and direct my thesis work. Special thanks to Dr. Bouldin for serving as my major professor and graduate advisor. In addition, I would like to thank each of my graduate professors for their excellent instruction and guidance.

Abstract

An application-specific integrated circuit (ASIC) must not only provide the required functionality at the desired speed but it must also be economical. In the past, minimizing the size of the ASIC was sufficient to accomplish this goal. Today it is increasingly necessary that the ASIC also achieve minimum power dissipation or an optimal combination of speed, size and power, especially in communication and portable electronic devices. The research reported in this thesis describes the implementation of a Huffman encoder and a finite impulse response (FIR) filter using a hardware description language (HDL) and the testing of the corresponding register transfer level (RTL) for functionality. The RTL was targeted for two different libraries, TSMC-0.18 CMOS and the Xilinx Virtex V1000EHQ240-6. The RTL was synthesized and optimized for different sizes, speeds, and power by using the Synopsys Design Compiler, FPGA Compiler II, and Mentor Graphics Spectrum. Cadence place and route tools optimized area, delay, and power of post-layout stages for TSMC-0.18. Xilinx place and route tools were used for the Virtex V1000EHQ240-6. The various ASICs were produced and compared over a range of speed, area, and power.

Table of Contents

List of Figures

Figure **Page**

List of Tables

Table **Page**

CHAPTER 1

Introduction

1.1 Overview

The first commercial discrete integrated circuit (IC) was introduced in the late 1950s. As predicted by Moore's Law in the 1960s, integrated circuit density has been doubling approximately every 18 months, and circuit speed has also simultaneously increased by the similar exponential [1]. Integrated circuit performance is typically characterized by the size of the chip, the speed of operation, the available circuit functionality, and the power consumption. The size of the chip not only affects the performance, but also influences the price of the chip and the number of potential sites and yield during fabrication. For example, reducing the area by factor of 4 increases the number of good dies by factor of 5 on a 5-inch wafer with 2-defects/sq. cm [6]. The other important factor for chip design is the time-to-market of the product. The earlier the product is brought into the market, the more money it is likely to produce. For this reason it is very important to find the most efficient way to optimize the size, delay, and power. This can be accomplished from different levels, such as the behavioral synthesis level, RTL/logic synthesis level, and the physical synthesis level.

1.1.1 Behavioral Synthesis Level

Behavioral synthesis is the process for synthesizing the circuit structure into RTL from the input behavioral descriptions written in a hardware description language such as

VHDL or Verilog. It describes how the system should behave in response to input signals but without having to specify the implementation. This level is the best level to debug the operation of the complete system and is also the level that provides the fastest emulation of the system [5].

Scheduling, allocation, and assignment (binding) are three phases in the behavioral synthesis process. Scheduling assigns operators into time slots without violating constraints. Allocation determines how many instances of each resource are needed in order to re-use hardware for different operators and to minimize registers and interconnections. The assignment decides which resources will be used by each operation of the behavioral description [3]. Behavioral synthesis attempts to minimize the number of resources to perform a task in a given time and tries to reduce the execution time for a given set of resources.

1.1.2 RTL/logic Synthesis Level

Conventional RTL synthesis or logic synthesis starts with a given RTL architecture where scheduling and allocation are already determined [2]. RTL/Logic synthesis is used to improve the logic to meet area or timing constraints. It provides a link between RTL architecture and a net-list. Generally, there are two stages in RTL/logic synthesis: the technology-independent stage and the technology-mapping stage [5].

A technology-independent stage uses algebraic and/or Boolean techniques for optimization. Most tools use algebraic techniques rather than Boolean techniques. These algebraic techniques use a series of factoring, substitution, and elimination steps to simplify the equation that represents the synthesized network. A technology-mapping stage matches pieces of the network with the logic cells that are available in a technology-dependent target cell library. During the mapping, the algorithms seek to minimize the area under consideration of any other constraints such as timing or power [4].

1.1.3 Physical Synthesis Level

Physical synthesis is very well developed and is accomplished primarily by the software. Once the gate net-list from the RTL/logic synthesis is available, and then it can be automatically converted to a layout. This step is normally divided into system partitioning, floorplanning, placement, and routing.

System partitioning, if needed, is the division of the microelectronics system into a number of ASICs in order to minimize the number of external connections between ASICs, thus keeping each ASIC small. Floorplanning consists of estimating sizes and assigning the location of all the blocks to keep the highly connected blocks physically close to each other. Placement involves defining the location of all logic cells within the flexible blocks and assigning the sets of the interconnect areas to minimize the layout area and interconnect density. Routing is the production of the connections between all the logic cells to minimize the total interconnection area and length used [4].

1.2 Research Objectives

Behavioral synthesis is the process that starts from the description of behavioral functionality and produces an architecture able to execute the design specification. The architecture is generally given as a RTL specification. The behavioral synthesis tool acts as a compiler that maps a high-level specification into an architecture. In order to modify the architecture, the behavioral description is simply changed and rerun through the behavioral synthesis tool. The high-level synthesis is the bridge between system design and CAD tools acting at the RTL/logic level [3].

The high-level of abstraction provides more powerful and complete methods to explore the design space within a much shorter turn-around time than the tasks performed by human designers. For example, software tools produce assembly code from a high-level language, such as C, by using the compiler. Previously, designers used assembly code and then translated them into machine code with an assembler. ***The goal of this research is to obtain a variety of size, speed and power solutions for a given HDL using several high-level synthesis tools***.

The first step of ASIC design process is to use a hardware description language to describe the desired functionality. The VHDL codes for this research work were provided by Honeywell Inc. for the Huffman encoder, and by the Boeing Company for the FIR filter. The code for the Huffman encoder was not completely error free in our CAD environment so some modification was necessary.

The second step in implementing an ASIC is to compile and simulate the VHDL code using a Mentor Graphics simulator, ModelSim. This step is also called pre-synthesis simulation. The next step is to change the design constraints so that a synthesis tool (Synopsys Design Compiler, Synopsys FPGA Compiler, and Mentor Graphics Spectrum) can produce multiple solutions within the area, delay, and power design space. The gate-level net-list is produced and made ready for the Cadence and Xilinx place and route tools. After this step, the different layouts are simulated again using ModelSim to verify the functionality and timing of the design. The final values of area, delay, and power are then determined.

1.3 Chapter Contents

The first part of Chapter Two is a brief tutorial of the Huffman encoder and FIR filter theory. It is followed by an introduction to the different types of ASICs. The brief explanation of how the parameters affect area, delay, and power will be given in the last part of Chapter Two.

A detailed explanation of ASIC design flow and design verification are illustrated in Chapter Three. Chapter Four presents the implementation of the Huffman encoder and FIR filter. Results and discussions are presented in Chapter Five and Chapter Six gives the summary and conclusion of this research work.

CHAPTER 2

Background

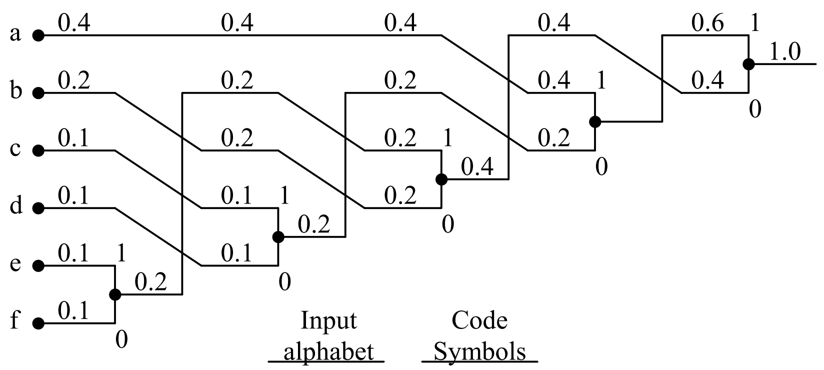
2.1 Huffman Coding Theory

The purpose of source coding is to form efficient descriptions of information sources. Source coding is used to either improve the signal-noise ratio (SNR) for a given bit rate or to reduce the bit rate for a given SNR. There are several different types of source coding, such as amplitude quantizing, differential pulse code modulation, synthesis/analysis coding, block coding, and redundancy-reducing coding [7]. Huffman coding is in the category of redundancy-reducing coding.

In Huffman coding, characters (or other data items) are represented as bit sequences of varying length, so that the most frequent character will have the shortest bit symbol. In written text, some characters are not used as frequently as others. The letter “e” appears quite often in English text, but rarely does one see a “z.” Even though the least frequent character usually will have more bit sequences than fixed-length bit sequences, Huffman coding will still achieve the shortest average code length compared to fixed-length bit sequences if the frequency distribution is appropriate for the input data. Huffman codes can be properly decoded because they obey the prefix property, which means that no code can be a prefix of another code. The complete set of codes can be represented as a binary tree, known as a Huffman tree.

As an example of the coding process, six input alphabets with different probabilities will form a Huffman tree as shown in Figure 2.1. The Huffman tree traces the tree path from right to left for each branch. The path contains the binary sequence, which will be the Huffman code.

A data compression technique varies the length of the encoded symbol in proportion to its information content. The more often a symbol or token is used, the shorter the binary string is used to represent it in the compressed stream. It is known that Huffman coding will achieve the shortest average code length. Table 2.1 shows how to calculate the average code length, $\bar{n}$.



Input alphabet	Code Symbols
a	11
b	00
c	101
d	100
e	011
f	010

Figure 2.1 Huffman coding tree for a six-character set [7].

Table 2.1 Calculation of average code length, $\bar{n}$ [7].

X_i	$P(X_i)$	Code	n_i	$n_i P(X_i)$
a	0.4	11	2	0.8
b	0.2	00	2	0.4
c	0.1	101	3	0.3
d	0.1	100	3	0.3
e	0.1	011	3	0.3
f	0.1	010	3	0.3

$$\bar{n} = \sum n_i P(X_i) = 2.4$$

$P(X_i)$ is probability y of input

n_i is number of data bits

The average code length is 2.4 bits per character. This means that on the average, 240 bits will be sent during the transmission of 100 input symbols. However, a fixed-length code of 3 bits representing a six-character input alphabet would require that 300 bits be transmitted for 100 symbols. Thus, the compression ratio will be 1.25 (3).

2.2 FIR Filter Theory

A filter is used to remove some component or modify some characteristic of a signal, but often the two terms are used interchangeably. A digital filter is simply a discrete-time, discrete-amplitude convolver. Basic Fourier transform theory states that the linear convolution of two sequences in the time domain is the same as multiplication of two corresponding spectral sequences in the frequency domain. Filtering is in essence the multiplication of the signal spectrum by the frequency domain impulse response of the filter.

A finite impulse response (FIR) filter performs a weighted average of a finite number of samples of the input sequence. The basic input-output structure of the FIR filter is a time-domain computation based on a feed-forward difference equation. Figure 2.2 shows a flow diagram of a standard 8-tap FIR filter. The filter has seven data registers. The FIR is often termed a transversal filter since the input data transverses through the data registers in shift register fashion. The output of each register (D1 to D7) is called a tap and is termed x[n], where n is the tap number. Each tap is multiplied by a coefficient c_k and the resulting products are summed. A general expression for the FIR filter's output can be derived in terms of the impulse response. Since the filter coefficients are identical to the impulse response values, the general form of a standard FIR filter can be represented as Equation 2.1 [15].

$$y[n] = \sum_{k=0}^{M} h[k]x[n-k] \tag{2.1}$$

When the relation between the input and the output of the FIR filter is expressed in terms of the input and the impulse response, it is called a finite convolution sum. We say that the output is obtained by convolving the sequences x[n] and h[n] [15]. There is a simple interpretation that leads to a better algorithm for achieving convolution. This algorithm can be implemented using the tableau that tracks the relative position of the signal values. The example in Figure 2.3 shows how to convolve x[n] with h[n].

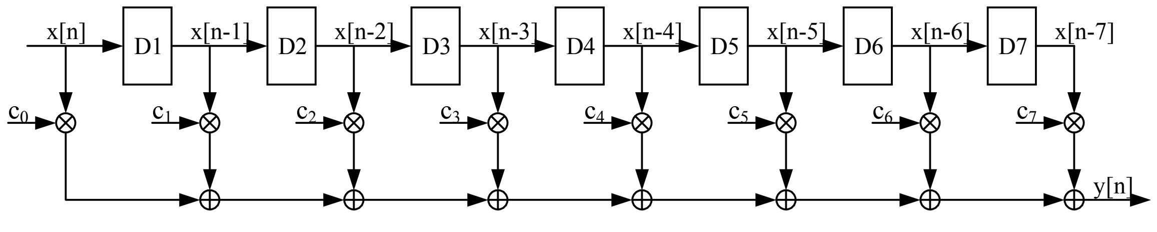


Figure 2.2 Block-diagram structure for the 8-tap FIR filter

n	0	1	2	3	4	5	6	7
x[n]	1	3	5	2	4			
h[n]	2	-1	3	1				
h[0]x[n-1]	2	6	10	4	8			
h[0]x[n-2]		-1	-3	-5	-2	-4		
h[0]x[n-3]			3	9	15	6	12	
h[0]x[n-4]				1	3	5	2	4
y[n]	2	5	10	9	24	7	14	4

Figure 2.3 Convolution of finite-length signals

The determination of filter coefficients controls the characteristic of the FIR filter. The most popular way to calculate coefficients is the window method. The coefficient function h(n) can be represented by the following equation.

$$h(n) = h_D(n)w(n) \tag{2.2}$$

$h_D(n)$ is ideal filter function in the frequency domain. For example, the ideal low-pass filter for $h_D(n)$ is the *sinc* function, and $w(n)$ is the window function. There are several common window methods such as Hamming window, Blackman window, and Kaiser window, which give stop-band attenuation less than 50db. The Hamming window is the most popular method due to the proper filter length and simple calculation. Equation 2.3 illustrates the function of a Hamming window.

$$w(n) = 0.54 + 0.46\cos\left(\frac{2\pi n}{N}\right) \tag{2.3}$$

where

N is the number of FIR coefficients.

2.3 Types of ASICs

An ASIC (application-specific integrated circuit) consists of a single chip or die. It is made on a thin silicon wafer, each containing hundreds of dice. Transistors and wiring are made from many layers on the top of a die [4]. Using CAD tools, the ASIC designer's task is to place and wire the transistors to perform logic functions that meet the design specification. There are several different approaches to designing an ASIC. The following will explain those design methods and give a brief explanation of each.

2.3.1 Full-Custom ASICs

In a full-custom design, a designer starts from the ground up using a layout editor to generate a physical layout description of the circuit. Full-custom ASICs are the most expensive to manufacture and design, but give full control to the designer to achieve a higher degree of optimization in both circuit area and performance. For instance, in this type of ASIC, the designer may change the width-to-length ratio of individual transistors to tune their performance [8]. On the other hand, this is a time-consuming and difficult task. The manufacturing lead-time is typically eight weeks [4]. The designer has to understand fully the characteristic and the rules of the physical layout in order to carry out each design. A full-custom approach is used when no existing cell libraries can meet the performance requirement, as for example, a microprocessor. The advantages are maximum circuit performance, minimum design size, and minimum high-volume production cost. The disadvantages are a long development cycle and manufacturing lead-time, less design change flexibility, and very high initial development cost [9].

2.3.2 Standard-Cell ASICs

The standard-cell is the logic block (AND gates and multiplexers, for example) which is optimized, tested for a particular function, and built into a standard-cell library. The designer places and interconnects multiple standard-cells to yield the desired circuit function. The range of gates in a standard-cell ASIC is usually from 10,000 to 10,000,000 gates (or more). The design of this type of ASIC requires a workstation-based development environment and costs about \$100K [6]. All the standard-cells have the same height and the cells are connected together horizontally to form rows. In order to fit standard-cells together, there are particular rules and formatting requirements for a standard-cell. The advantages of standard-cells are time efficiency and risk reduction by using a pre-optimized and pre-tested standard-cell library. The disadvantages include the time to develop a standard-cell library and long manufacturing lead-time, which is approximately eight weeks [4].

2.3.3 Gate Array ASICs

In a gate array ASIC, thousands of transistors are prefabricated on a silicon wafer in regular two-dimensional arrays, containing 10,000 to 10,000,000 gates (or more) [6]. Initially the transistors in the arrays are not connected to each other. That means the top few layers of the metal have not been fabricated. The designer will then define the connection between transistors to perform the desired logic function using custom masks. For this reason it is often called a mask gate array (MGA). The process of adding metal wires to a gate array is called *personalizing* the array [8]. Wafers of gate arrays can be stock-piled so that only the personalization of the top few layers of metal need to be performed, thereby reducing the manufacturing lead-time to 3–5 weeks [6]. The following are three different types of gate array ASICs [4]:

- Channeled gate array.
- Channelless gate array.
- Structured gate array (embedded gate array).

The differences among the gate array ASICs are slight. Channeled gate arrays have spaces between rows of base cells for interconnections. Channelless gate arrays, on the other hand, are completely filled with array base cells. Structured gate arrays like channelless gate arrays have no spaces between rows. The difference lies in the embedded block area which can be dedicated to a specific function, usually building memory cells. Figure 2.4 shows three different types of gate arrays. The advantages of gate array ASICs are shorter turnaround time due to prefabrication of most of the layers and lower cost. The major disadvantage is that all the transistors are the same size (width

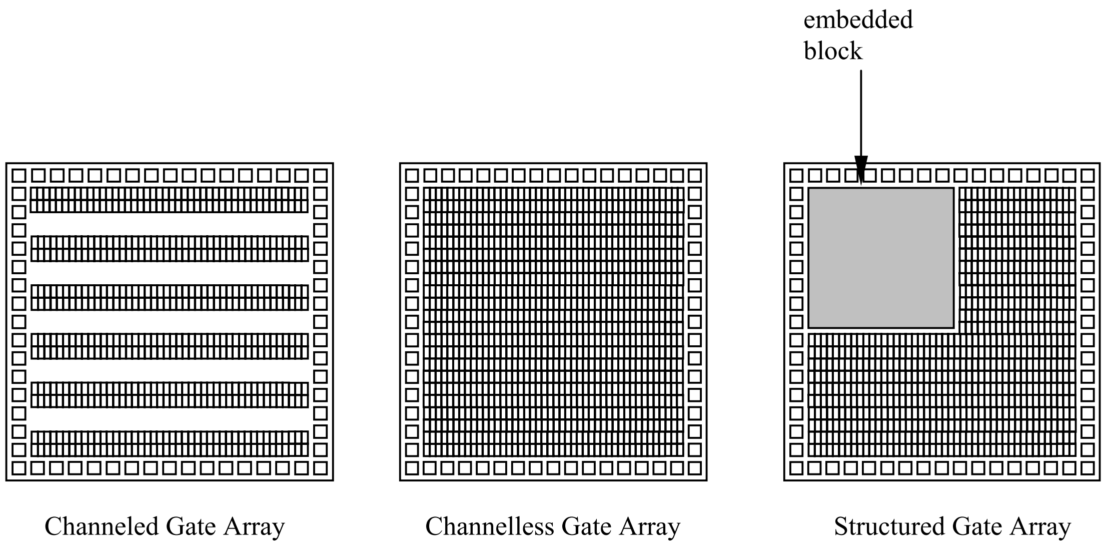


Figure 2.4   Different types of gate array ASICs [4].

and height). The cell library must be specially designed to meet the requirement [8]; otherwise, there will be wasted areas for all those unused transistors.

2.3.4 Programmable Logic Devices

A programmable logic device (PLD) consists of logic array blocks (LABs) which implement logic as two-level sum-of-product expressions. A LAB contains wide and programmable AND-gates, and narrow and fixed OR-gates. The AND plane implements the product terms and the OR plane implements the sums. PLD includes programmable array logic (PAL). A PLD can replace 300 to 8000 gates with a single package of 24-80 pins. However, it only contains up to 8000 gates, which limits the logic complexity. It is simple enough to use a PC-based development system costing about $5K-10K [6], can be programmed within few minutes, and has the shortest turnaround time to handle a simple design.

2.3.5 Field-programmable Gate Arrays

Field-programmable gate array (FPGA) is a step above the PLD in complexity. The difference between FPGA and PLD is very little. Both FPGA and PLD can be volatile or non-volatile. FPGA is just larger and more complex than a PLD. FPGA consists of a two-dimensional array of logic blocks. Each logic block is programmable to implement any logic function. Thus, they are also called configurable logic blocks (CLBs) [8]. Switchboxes or channels contain interconnection resources that can be programmed to connect CLBs to implement more complex logic functions. Designers can use existing CAD tools to convert HDL code in order to program FPGAs. An FPGA contains 2,000 to

2,000,000 gates (or more) [6]. Since the FPGA can be reprogrammed, the turnaround time is only a few minutes. The advantages of FPGAs are lower prototyping costs and shorter production lead times, which advances the time-to-market and in turn increases profitability. It can also ensure the reliability of the design on the board. The disadvantages include lower speed of operations and lower gate density, which has a larger area compared to a MGA. Thus, a typical FPGA may be 2x-10x slower and 2x-10x more expensive than an equivalent-gate MGA.

2.3.6 Summary

From the previous sections, we know the types of ASICs can be defined as two main categories: non-programmable and programmable ASICs. There are some advantages of programmable ASICs (PLDs and FPGAs) such as rapid prototyping, low risk, effective design verification, and low fixed costs. There are also some disadvantages of FPGAs such as bigger chip size, higher part cost, more power consumption, and lower speed. Because of those factors, FPGAs are better for smaller volume applications, reducing time-to-market, and prototyping [6]. In terms of prototyping, FPGAs can be easily tested and can always be reprogrammed if the design does not completely meet the specification. On the other hand, a MGA and a standard-cell are far better for high-volume and/or high-performance applications for which FPGAs cannot meet the specification.

2.4 Area, Delay, and Power Consideration

ASIC performance is typically characterized by the size of the chip, the speed of operation, and power consumption. The following sections will briefly describe how the parameters affect area, delay, and power.

2.4.1 Area

Chip area is determined by the logic blocks, interconnections and the I/O pads. Routing area, area of diffusion, transistor size, and parasitic transistor capacitance are some of the important factors that affect the area of the device. Routing area is the most demanding factor of all, taking up to 30% of the design time and a large percentage of the layout area [8]. Using the technology mapping approach, the routing area can be estimated by using two parameters available at the mapping stage; one is the fanout count of a gate, and the other is the "overlap of fanin level intervals" [11]. Minimizing switching capacitance can reduce the size of the transistors. Typically up to 70 to 80% of node parasitic capacitance is due to the interconnection routing. There are some techniques that can reduce the routing area such as the use of more metal layers routing interconnects and new technology to reduce λ size. Reducing λ can also reduce the area of diffusion and transistor size.

The area of a circuit has a direct influence on the yield of the manufacturing process. Yield is defined as the number of chips that are defect-free in a batch of manufactured chips. According to Stapper [12], the following is the yield formula to calculate the original yield of the memory array:

$$Y_0 = \left(1 + \frac{A\delta}{\alpha}\right)^{-\alpha} \tag{2.4}$$

where

δ is the defect density

A is the area of the RAM array

α is some clustering factor of the defects

From the equation above we know that the smaller the chip area, the higher the yield. A low yield would mean a high production cost, which in turn would increase the selling cost of the chip.

2.4.2 Delay

The time taken to charge and discharge the load capacitance C_L determines the switching speed of the CMOS gate. Rise time is defined during charging time from 10 % to 90% of its steady-state value; this is the same as the fall time. Delay time is defined by the time difference between 50% of charging time and 50% of discharging time. The rise time is described by Equation 2.5 [1].

$$t_r = \left(\frac{C_L}{\beta_p}\right)\frac{1}{V_{DD}(1-\gamma)}\left(2\frac{\gamma - 0.1}{1-\gamma} + \ln(19 - 20\gamma)\right) \tag{2.5}$$

The propagation delay of 50% of charging time is [1]:

$$t_{PLH} = \left(\frac{C_L}{\beta_p}\right)\frac{1}{V_{DD}(1-\gamma)}\left(\frac{2\gamma}{1-\gamma} + \ln(3 - 4\gamma)\right) \tag{2.6}$$

where

$$\gamma = -\frac{V_{tp}}{V_{DD}} \tag{2.7}$$

Due to the symmetry of the CMOS circuit, the fall time is [1]

$$t_f = \left(\frac{C_L}{\beta_n}\right)\frac{1}{V_{DD}(1+\eta)}\left(2\frac{\eta - 0.1}{1-\eta} + \ln(19 - 20\eta)\right) \tag{2.8}$$

and the propagation delay of 50%of discharging time is [1]

$$t_{PHL} = \left(\frac{C_L}{\beta_n}\right)\frac{1}{V_{DD}(1-\eta)}\left(\frac{2\eta}{1-\eta} + \ln(3 - 4\eta)\right) \tag{2.9}$$

where

$$\eta = \frac{V_{tn}}{V_{DD}} \tag{2.10}$$

Therefore, the delay time will be

$$t_{delay} = t_{PLH} + t_{PHL} \tag{2.11}$$

From the equations, in order to improve the individual gate delays, the load impedance C_L is reduced or the current gain of the transistors is increased. Increasing the current gain means higher β, approximately equal to the W/L of the transistor. Therefore, by increasing β, the transistor size will increase, thus affecting the size of the chip.

2.4.3 Power

There are three sources that cause power dissipation in a CMOS circuit:

- Dynamic power dissipation due to switching current
- Dynamic power dissipation due to short-circuit current
- Static power dissipation

2.4.3.1 Switching Current

When the p-channel transistor charges the output capacitive load, C_L, the current through the transistor is $C_L(dV/dt)$. Figure 2.5 shows an inverter with output capacitive load. The power dissipation is thus $C_L V(dV/dt)$ for one-half the period of the input. The power dissipated in the p-channel transistor is thus

$$\int_0^{1/(2f)} C_L V\left(\frac{dV}{dt}\right)dt = \int_0^{V_{DD}} C_L V\, dV = \frac{1}{2} C_L V_{DD}^2 \qquad 2.12$$

When the n-channel discharges the capacitor, the power dissipation is equal and that makes the total switching power dissipation [10]

$$P_D = C_L V_{DD}^2 f \qquad 2.13$$

From Equation 2.13, the best way to reduce power is to reduce V_{DD}, which has a squared term. That is why some of the TTL chips operate under 3.3V to reduce power dissipation.

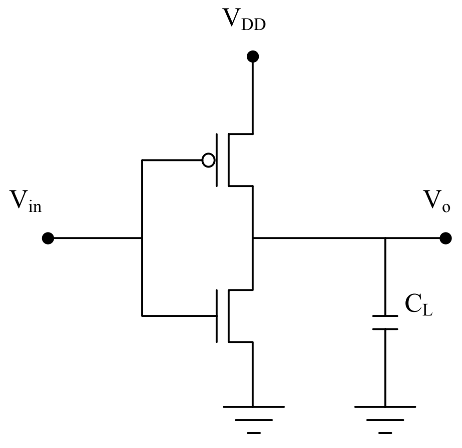


Figure 2.5 CMOS inverter with capacitive load, C_L

2.4.3.2 Short-Circuit Current

Another source of power dissipation is during the transition of "0" to "1" or "1" to "0", and both p-channel and n-channel transistors are on for a short period of time. This results in a short current pulse from V_{DD} to GND that causes a short-circuit power dissipation. The short-circuit power dissipation is given by

$$P_{SC} = I_{mean} \times V_{DD} \qquad 2.14$$

Assume that $V_{tn} = -V_{tn}$, $\beta_n = \beta_p = \beta$, and $t_r = t_f = t_{rf}$, thus for an inverter without load [4].

$$P_{sc} = \frac{\beta f t_{rf}}{12}\left(V_{DD} - 2V_t\right)^3 \qquad 2.15$$

In general, the transistor size of the p-channel and the n-channel are not the same to achieve the same rising time and falling time. The short-circuit current is also typically less than 20% of the switching current.

2.4.3.3 Static Power Dissipation

Considering a CMOS gate, as shown in Figure 2.6, when the p-channel is biased "ON," the n-channel will be "OFF". On the other hand, when the n-channel is "ON," the p-channel will be "OFF." Since one of the transistors is always "OFF," there should be no DC current from V_{DD} to GND. However, there is a small leakage current between the diffusion and the substrate to cause the static dissipation. The leakage current is described by Equation 2.16 [5].

$$I = I_s\left(e^{qV/kT} - 1\right) \qquad 2.16$$

where

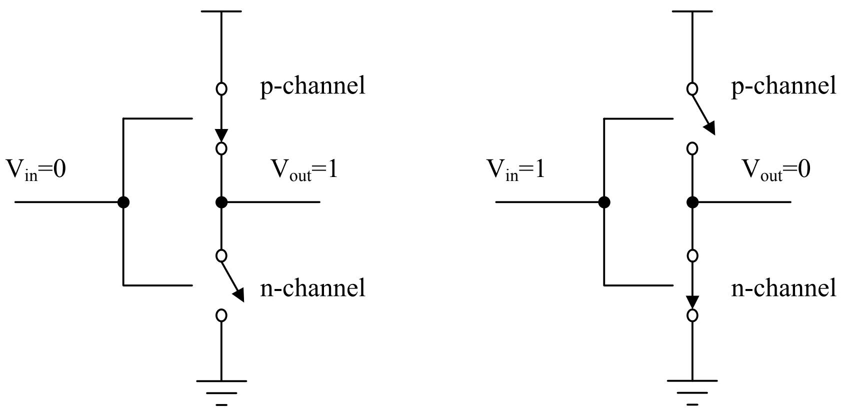


Figure 2.6 CMOS inverter represented as switch

I_s = reverse saturation current

V = diode voltage

q = electronic charge $\left(1.602 \times 10^{-19} C\right)$

k = Boltzmann's constant $\left(1.38 \times 10^{-23} J/K\right)$

T = temperature.

The static power dissipation is produced by the leakage current and the supply voltage. Then the total static power dissipation is obtained from

$$P_s = \sum_{1}^{n} \text{leakage current} \times \text{supply voltage} \qquad 2.17$$

where

n = number of devices.

Typical static power dissipation due to leakage for an inverter operating at 5 volts is between 1 and 2 nanowatts [5]. The static power dissipation is generally negligible due to the low range compared to dynamic power dissipation.

2.4.4 Summary

From the previous sections, the general factors on area, delay, and power are explained from the fundamental theory. There are some causes that have a more significant effect on area, delay, and power. The power consumption of interconnect wires and clock signals can be up to 40 and 50% of the total on-chop power consumption. More than 90% of the power dissipation of traditional FPGA devices has been attributable to the interconnection [13]. From Chapter One, we know that area, delay, and power optimization can be achieved from a behavioral synthesis level, RTL/logic level, and physical level. Each level has different controls and effects to the design, and most of the design optimizations are done by these levels. The importance of the fundamental theory is that most synthesis tools are based on techniques developed from the fundamental theory just presented.

CHAPTER 3

Design Flow and Design Verification

3.1 ASIC Design Flow Description

Figure 3.1 shows the programmable ASIC design flow, and Figure 3.2 shows the ASIC design flow of this research. The VHDL codes were obtained from Honeywell Inc. and the Boeing Company. In this research, the focus was on the Synopsys synthesis tools. The results from Mentor Graphics were also performed for comparison. The Synopsys synthesis can target different cell libraries to implement different types of ASICs. The four major steps of the design flow are pre-synthesis simulation, synthesis, placement and routing, and post-layout simulation.

- **Pre-synthesis Simulation**

Before the synthesis procedure, pre-synthesis simulation must be performed. Functionality tests are usually the first tests a designer might construct as part of the design process. The verification of functionality will be explained in Section 3.2. The pre-synthesis simulation procedure not only tests the functionality of the design, but also compiles the VHDL codes of the design. Before the simulation, all the VHDL codes for the design have to be complied without error, and then simulation can take place. Pre-synthesis simulation is technology-independent because it only simulates the functionality of the design without actually mapping to the target library. It ignores timing and includes unit-delay simulation, which sets delay to a fixed value.

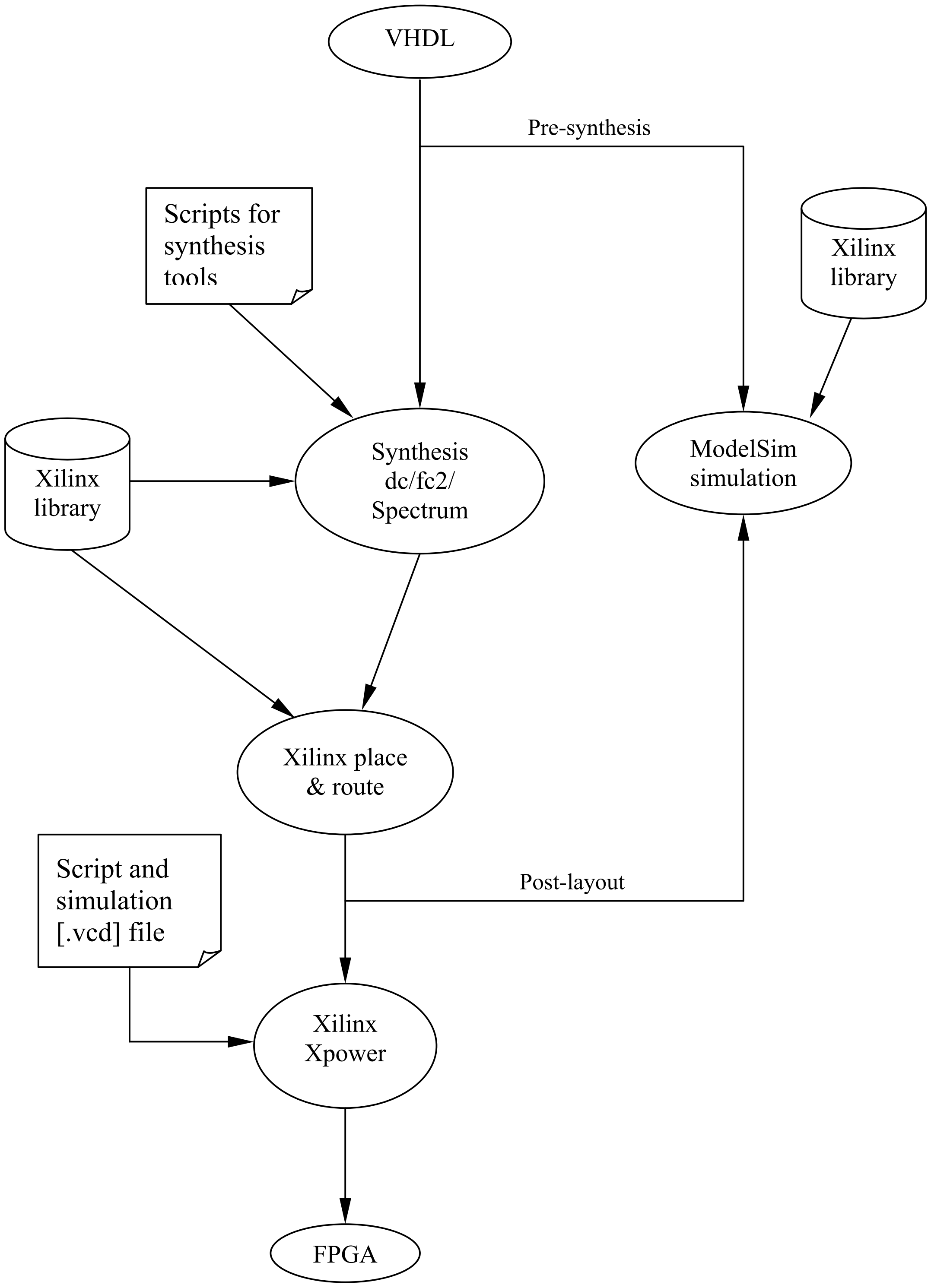


Figure 3.1 Programmable ASIC design flow

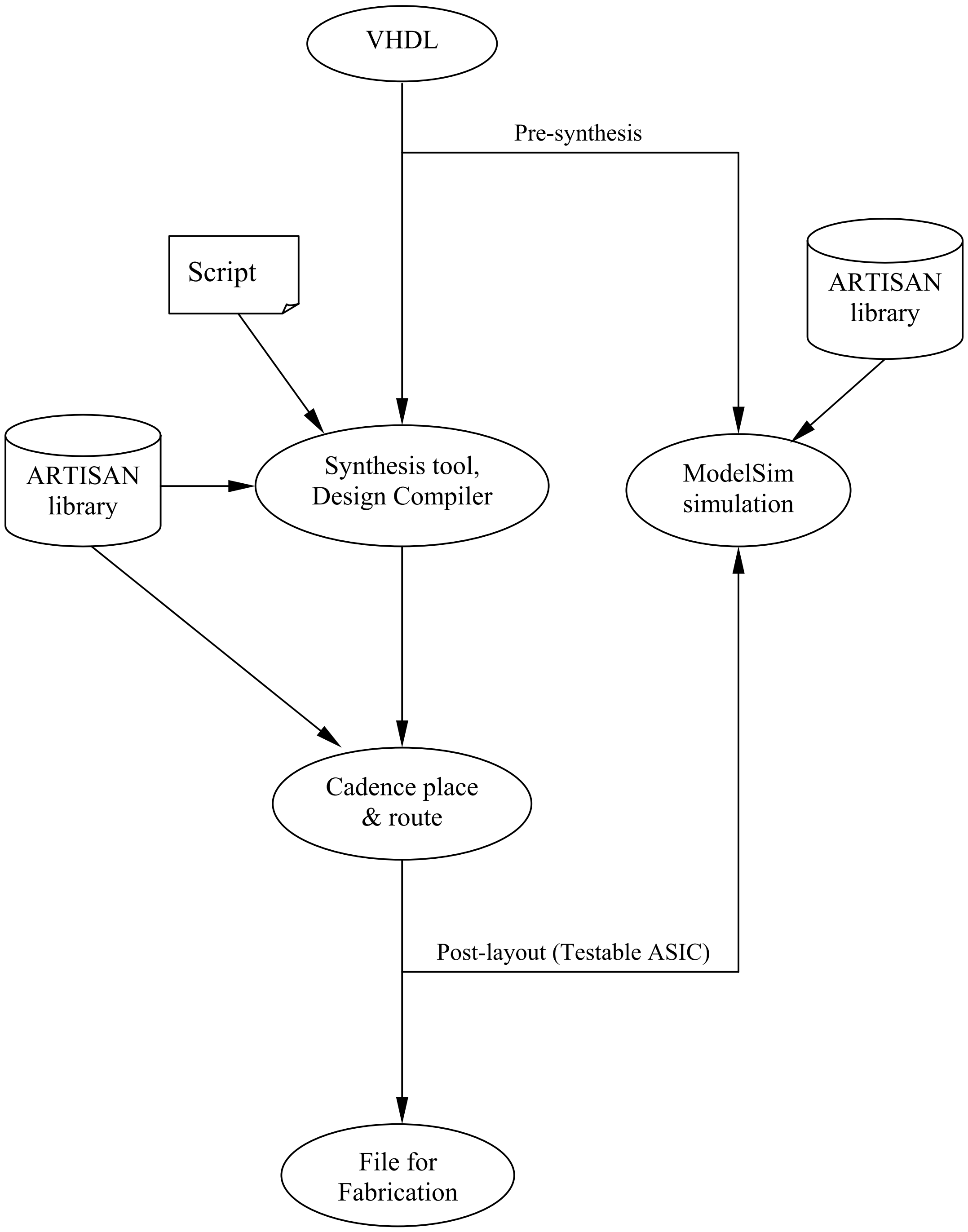


Figure 3.2 ASIC design flow

- Synthesis

The pre-synthesis simulation is followed by the synthesis procedure. In this research, a Synopsys synthesis tool; Design Compiler (DC), was used. Design Compiler synthesizes the high-level design description, written in VHDL or Verilog, into optimized gate-level designs. The Design Compiler generates ASIC designs by employing user-specified standard-cell or gate array libraries and translates designs from one technology to another. In this research, Xilinx libraries were used to implement the FPGA layout, and the Artisan library was used for Cadence place and route tools. The Design Compiler used the target library and optimized the design.

- Placement and Routing

After synthesis, the design is already mapped with the target library (Xilinx or Artisan) and the technology has been specified. Xilinx uses its own placement and routing tools to generate and optimize the layout. For the Artisan library, Cadence place and route tool links with the Artisan library to optimize and generate the layout of the design.

- Post-layout Simulation

After placement and routing, the physical layout of the design is generated. Unlike pre-synthesis simulation, post-layout simulation is technology-dependent and depending on the library, the delay will be different. The Standard Delay Format or “sdf” file has already been generated by the place and route tool, so the logic-cell delays and the interconnect delays have been calculated. Therefore, post-layout simulation will have an estimated delay time for the signal outputs for the design. After the post-layout simulation, if the simulation result is correct, the design is ready to be downloaded into the FPGA or sent to a foundry for fabrication.

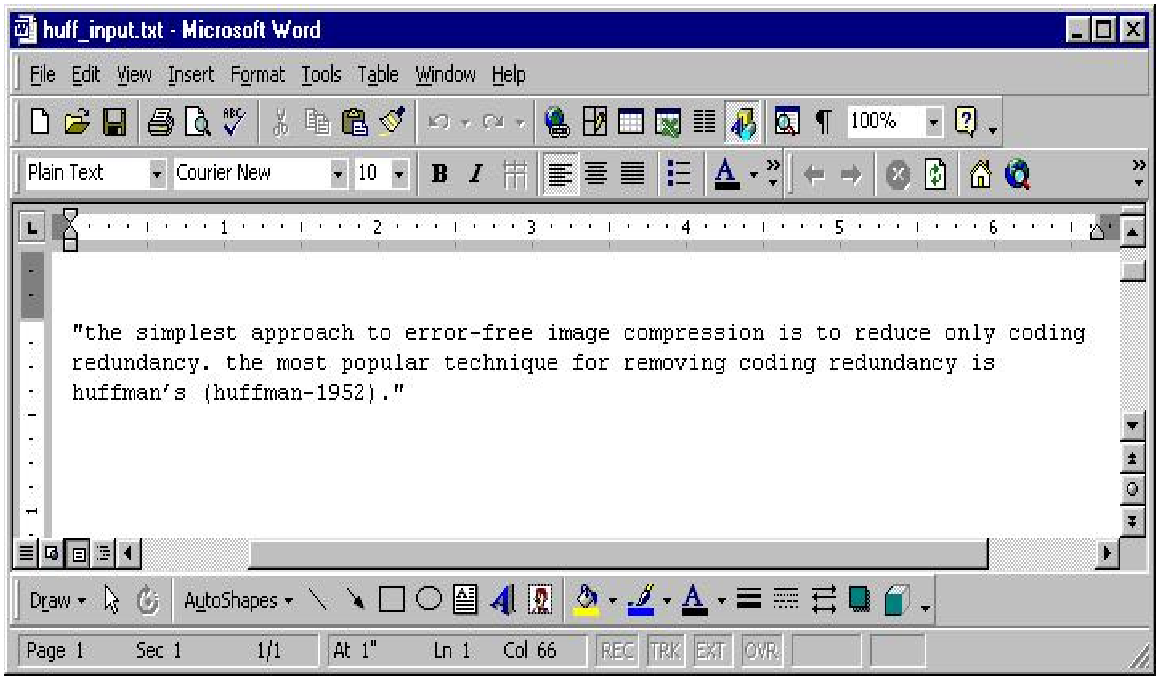
3.2 Design Verification

It is very important to know the correct result of the design. That is why we perform pre-synthesis simulation and post-layout simulation to verify the functionality of the design. The purpose of the simulation is to verify the result. However, the question is whether or not we know the result is correct in the first place. In order to know the correct result of the Huffman encoder, a C++ code (shown in Appendix B1) must be generated and compiled as the function of the Huffman encoder. An input data (shown in Figure 3.2) was made for testing the accuracy of the conversion from the program. Based on the input data, a Huffman look-up table (shown in Table 3.1) was generated with the probability of each character. The program read the input data from the file "huff_input.txt" and encoded the input data. Then it displayed the Huffman code on the screen (shown in Figure 3.3) and wrote out the Huffman code into file "huff_code.txt" (shown in Figure 3.4).

In the VHDL environment, the input signals to the entity under scrutiny are generated by means of another VHDL entity referred to as a Test-bench. The concept of a test-bench is an accepted VHDL standard means of simulating, testing, and documenting an entity.

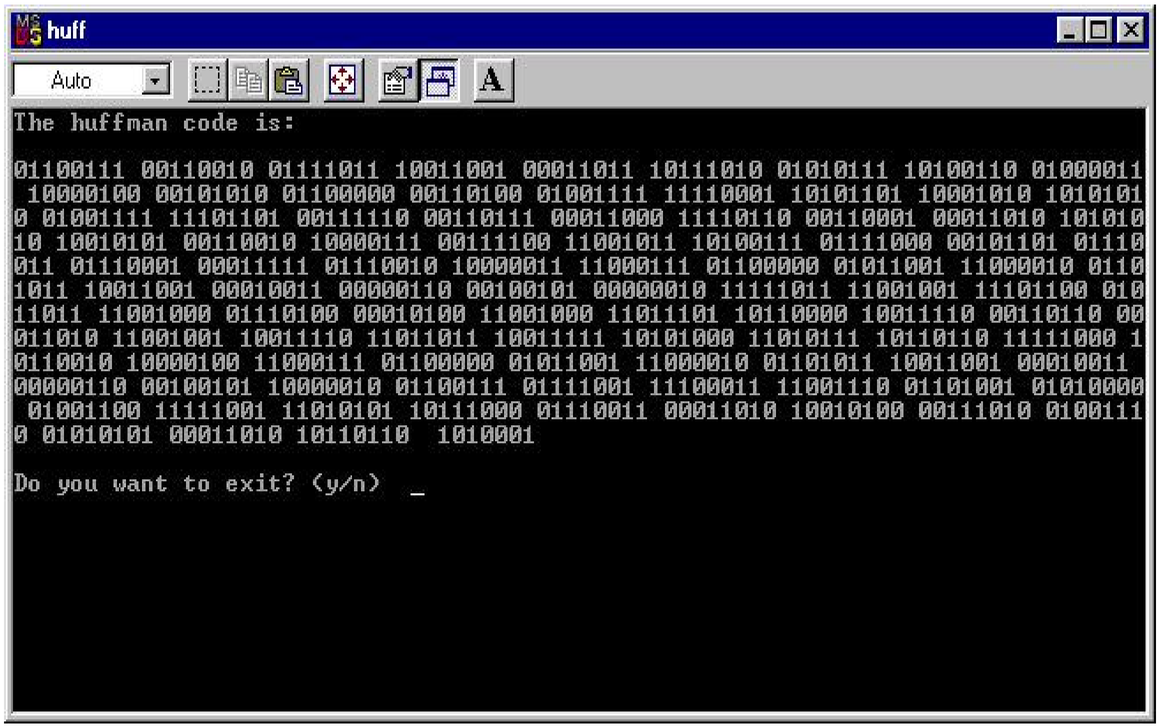
Table 3.1 Huffman look-up table

Symbol	Probability	Huffman Look-up number	Huffman Code
q	0.005714286	31	011110
v	0.005714286	30	011111
apostrophe	0.005714286	26	100110
begin quote	0.005714286	29	100111
end quote	0.005714286	27	101000
9	0.005714286	23	101001
5	0.005714286	21	101010
(	0.005714286	28	101011
)	0.005714286	25	110000
1	0.005714286	20	110001
2	0.005714286	24	110100
hyphen	0.011428571	19	110101
period	0.011428571	15	110110
l	0.017142857	22	110111
y	0.017142857	18	00000
g	0.022857143	17	00001
h	0.028571429	14	00110
f	0.034285714	13	00111
p	0.034285714	16	01000
t	0.034285714	11	01001
d	0.04	12	01100
m	0.04	10	01101
u	0.04	7	01110
a	0.045714286	9	10000
c	0.045714286	8	10001
s	0.045714286	6	10010
i	0.051428571	5	11001
o	0.062857143	4	0001
n	0.068571429	3	0010
r	0.068571429	2	0101
e	0.08	1	1011
space	0.137142857	0	111



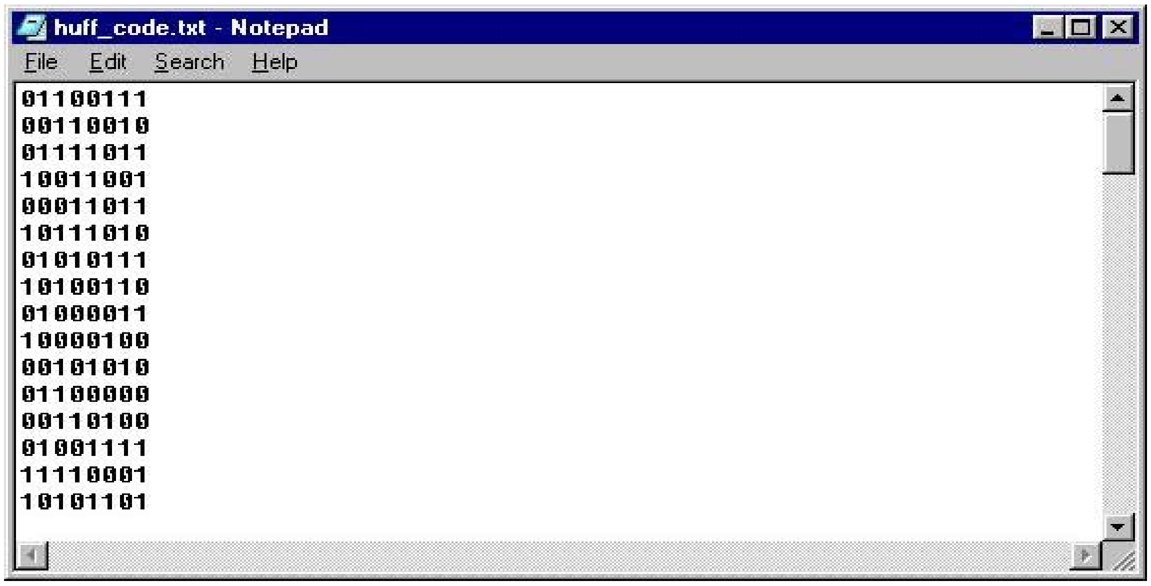
```
"the simplest approach to error-free image compression is to reduce only coding
redundancy. the most popular technique for removing coding redundancy is
huffman's (huffman-1952)."
```

Figure 3.2 Input file for C++ program



```
The huffman code is:

01100111 00110010 01111011 10011001 00011011 10111010 01010111 10100110 01000011
 10000100 00101010 01100000 00110100 01001111 11110001 10101101 10001010 1010101
0 01001111 11101101 00111110 00110111 00011000 11110110 00110001 00011010 101010
10 10010101 00110010 10000111 00111100 11001011 10100111 01111000 00101101 01110
011 01110001 00011111 01110010 10000011 11000111 01100000 01011001 11000010 0110
1011 10011001 00010011 00000110 00100101 00000010 11111011 11001001 11101100 010
11011 11001000 01110100 00010100 11001000 11011101 10110000 10011110 00110110 00
011010 11001001 10011110 11011011 10011111 10101000 11010111 10110110 11111000 1
0110010 10000100 11000111 01100000 01011001 11000010 01101011 10011001 00010011
00000110 00100101 10000010 01100111 01111001 11100011 11001110 01101001 01010000
 01001100 11111001 11010101 10111000 01110011 00011010 10010100 00111010 0100111
0 01010101 00011010 10110110  1010001

Do you want to exit? (y/n)  _
```

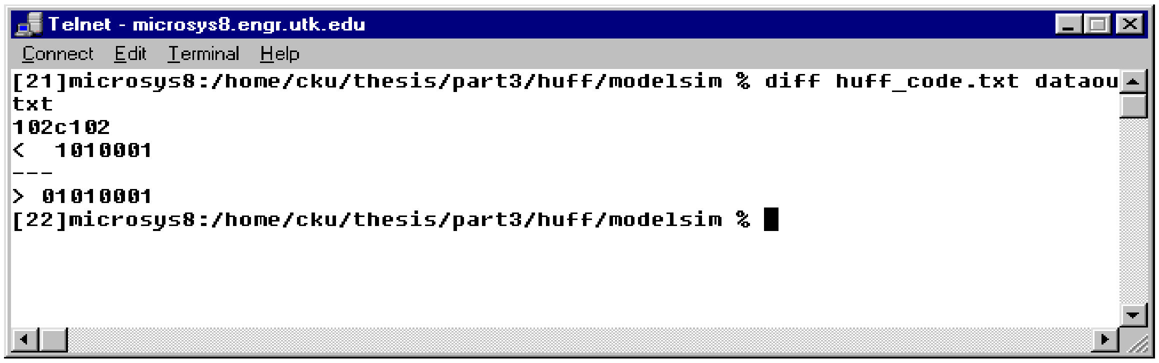
Figure 3.3 Output Huffman code from C++



```
01100111
00110010
01111011
10011001
00011011
10111010
01010111
10100110
01000011
10000100
00101010
01100000
00110100
01001111
11110001
10101101
```

Figure 3.4 Output Huffman code from C++ save as file.

The entity/test-bench pair can form the basis for executable specifications and documentation in a top-down design methodology [14]. In other words, the test-bench will generate the input signals for the design and, if necessary, it will also give the appropriate response from an output signal of the design. The test-bench of this design (shown in Appendix A1) will read the data input from a file "datain.txt". The data in datain.txt is the input sequence of corresponding values of input characters. After the simulation, the test-bench will write the output as "dataout.txt". The "diff" command under UNIX is used to compare the differences between "huff_code.txt" and "dataout.txt". The result is shown in Figure 3.5. There was only one difference between the two files. In the C++ program, if the last Huffman code does not fill up the last group of 8-bits, the software will fill it up with spaces, "_1010001" in this case. However, in the simulation output, it fills up by the next Huffman code, "01010001" in this case. This is the only the difference. Therefore, the output from simulation is accurate.



```
[21]microsys8:/home/cku/thesis/part3/huff/modelsim % diff huff_code.txt dataou
txt
102c102
<   1010001
---
> 01010001
[22]microsys8:/home/cku/thesis/part3/huff/modelsim %
```

Figure 3.5 Compare result of using "diff" Command

For the verification of the FIR, the test-bench of the FIR will produce the expected output that can compare with the actual data output to verify the accuracy of the results. Therefore, the simulation result can confirm verification. If the expected output and actual output do not match, an error is reported to the transcript.

CHAPTER 4

Implementation

This chapter describes the design methodology for this research. The Function description of the VHDL code used for this research is explained in detail, followed by the procedures for achieving delay, area, and power optimization.

4.1 Function Description of Huffman VHDL Code

The VHDL module huffman.vhd describes an algorithm for transforming data words (each 8-bits long) from a memory module and encoding them into a variable-length Huffman-coded sequence which is stored in another memory module (with 8-bit word width). The code consists of four modules: huffman.vhd (top module), huff.vhd, state.vhd, and control.vhd. A simplified description of each module's functionality is listed in Table 4.1.

Table 4.1 General description of modules

VHDL Filename	Description
huffman.vhd	Top-level module. Plus support code to interface with lower-level modules.
huff.vhd	Simple Huffman look-up table (only 32 symbols for an 8-bit input)
state.vhd	State machine which loads and compiles 8-bit sequences of Huffman codes. Writes output to memory.
control.vhd	Keeps track of memory address to use when reading and writing data to memory.

The huff.vhd module describes a simple, partial Huffman lookup table for an 8-bit input. Given an 8-bit input, the traditional Huffman encoding would generate a variable length output (i.e. if the input were an 8-bit value, for example “00010010”, the encoded output might only be a 3-bit value, say “110”). However, in the physical world, the number of data output lines is fixed and cannot be varied. Therefore, huff.vhd uses an 8-bit output “dataout” to describe the traditional Huffman code (with some additional zeros) plus an additional 3-bit output “encodelength” to describe the length. For example, if the Huffman code is to output is “011”, then the huff.vhd module would output: dataout => “00000**011**”, encodelength => “010”. The encodelength, which equals two, describes the desired length. The reason the encodelength equals two is that it counts 3 from “000” to “010”. Therefore, 3-bits of code length are present. Adjusting for this condition, a traditional Huffman table can be generated from huff.vhd. Table 3.1 from Chapter Three is the traditional Huffman table with the assign characters and probability for this research.

This huffman.vhd, state.vhd, and control.vhd module combines to create a six-level state machine. The combined state machine diagram is illustrated in Figure 4.1. The “Load” state is the first state occupied upon reset. It generates an address (“address” output), which is used to load the next data input from an external memory module. During the “Load” state, the data input is loaded and cycled through the huff.vhd module, resulting in an 8-bit “dataout” and a 3-bit “encodelength” value. Once the data input has been loaded and cycled through the huff.vhd module, the system moves into the “Shift” state.

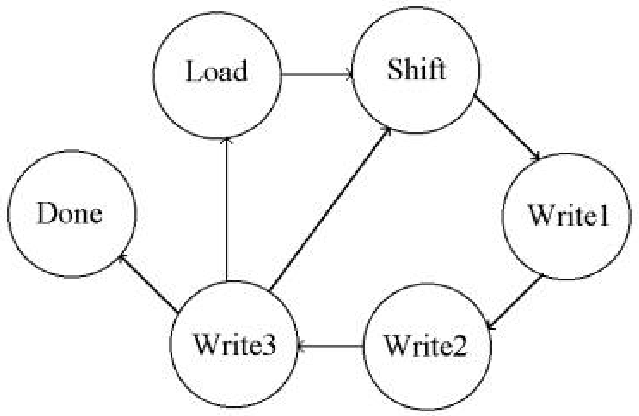


Figure 4.1 State diagram for Huffman.vhd

In the “Shift” state, the system will build a variable-length Huffman coded sequence. If the sequence is less than 8-bits, the “Shift” state will continue to load and encode data from memory until its sequence of coded data is 8-bits or greater. An 8-bit sequence must be maintained so that when the encoded sequence is written to a memory module (with word width equal to 8-bits), its width will match appropriately. The functionality of the “Shift” state is very important since it must be fully understood to properly simulate the Huffman code without use of an external memory module. The “Shift” state is a multi-clock cycle state. During this multi-cycle process, the “Shift” state assumes that the “dataout” input and the “encodelength” input (from huff.vhd) will not change unless the memory “address” output value changes. During a simulation it is the responsibility of the programmer to ensure that this requirement is met. This can be particularly difficult since the number of clock cycles required for various inputs is different. For example, if the input values (from huff.vhd) are dataout = “00011110” and encodelength = “101”, then the “Shift” state will require 6 clock cycles to process this data. During this time the simulated data input must not be changed (the input must remain static for exactly 6 clock cycles, no more and no less). By contrast, if the input values (from huff.vhd) are dataout = “00000111” and encodelength = “010”, then the “Shift” state will require only 3 clock cycles to process this data (again, the data must remain static for exactly 3 clock cycles). As a result, if the simulator inputs are not timed properly, they will not effectively simulate the response of a memory module and the output data will be nonsense. Assuming these conditions have been met, and an encoded 8-bit sequence properly constructed, the system moves to the “Write1” state.

The “Write1” state is merely a single clock latency state. This state pauses for one cycle to allow time for other operations to complete. Other operations include switching the “address” output to display the proper memory address to which the encoded data output will be written. After one clock cycle the module moves into the “Write2” state.

In the “Write2” state, the system write-enables (Wen => low) an external 8-bit memory module for the address described by the “address” output. The data written to memory is the 8-bit encoded sequence constructed during the “Shift” state. The varying length of Huffman codes will fill up 8-bit memories by sequence. Since Huffman code obeys prefix property, which means that no code can be a prefix of another code, the codes can be correctly decoded when the codes are read from memory. Only 8-bits of data are written to memory, so if a 9-bit sequence was constructed during the “Shift” state, the remaining 1-bit will be utilized in the next sequence generated by the “Shift” state. After one clock cycle the module moves into the “Write3” state.

The "Write3" state allows for a single clock cycle latency so that the memory address can again be switched back to the memory address for the data input. The "Write3" state also determines whether the system needs to return to the "Load" state or the "Shift" state. If the "Shift" state had constructed a sequence greater than 8-bits, (i.e., if a 9-bit sequence was constructed then there would be 1-bit remaining) then those remaining bits represent the start of a new sequence and the system would need to return to the "Shift" state. However, if there are no bits remaining, then the system needs a new data value and must return to the "Load" state. It is assumed that the maximum memory size for the "data in" memory module is 64 words. Therefore, if the memory address of the next data value is greater than 64, the module would instead move to the "Done" state, where it would remain disabled until reset.

4.2 Function Description of The FIR VHDL Code

The FIR filter implements a low-pass operation with a cutoff frequency of 17 MHz with a sampling frequency of 50 MHz. Figure 4.2 is the symbol of the FIR filter. The block diagram of FIR is similar to Figure 2.2 in Chapter Two, but instead of 7 registers, it has

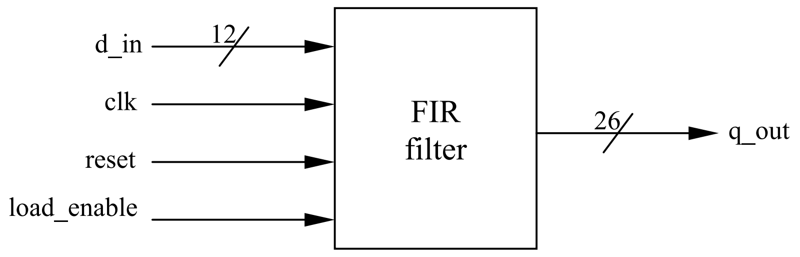


Figure 4.2 Symbol of FIR filter

31 registers; that means it has 32 coefficients. Equation 2.2 and 2.3 from Chapter Two explained how to calculate the coefficients. The coefficients are fixed and shown in Table 4.2. The registers are triggered on the rising edge of the clock when load-enable is active high. They are synchronously reset to zero when reset is low. This implementation had three pipeline stages. The first stage controlled the data shifting into the register. The second stage multiplied the data from registers with coefficients. The third stage was the added tree, which summed all the products from previous stage. The VHDL code used only adders to implement the added tree without any registers; in other words, it is a sequential process. Therefore, the delay of the third stage is quite long due to the serial of adding process. To improve this problem, we can change the implementation of the VHDL code to modify the structure of the added tree, which inserts some registers

Table 4.2 Coefficient table of FIR filter

Coefficient	Value	Binary	Coefficient	Value	Binary
0	0.013671875	0000001100	16	0.0576171875	0111111111
1	0.0078125	0000000111	17	-0.0527.34375	1111010001
2	-0.02734375	1111101000	18	-0.072265625	1111000000
3	-0.009765625	1111110111	19	0.0859375	0001001100
4	0.0078125	0000000111	20	-0.044924875	1111011000
5	-0.025390625	1111101001	21	-0.0078125	1111111001
6	0.009765625	0000001001	22	0.03515625	0000011111
7	0.0078125	0000000111	23	-0.03515625	1111100001
8	-0.03515625	1111100001	24	0.0078125	0000000111
9	0.03515625	0000011111	25	0.009765625	0000001001
10	-0.0078125	1111111001	26	-0.025390625	1111101001
11	-0.044924875	1111011000	27	0.0078125	0000000111
12	0.0859375	0001001100	28	-0.009765625	1111110111
13	-0.072265625	1111000000	29	-0.02734375	1111101000
14	-0.0527.34375	1111010001	30	0.0078125	0000000111
15	0.0576171875	0111111111	31	0.013671875	0000001100

between the adders. In this case, the registers will store the data after few of the adding process instead of all of the adding process; in other words, it becomes a pipeline process. Even though this requires more clock cycle to allow the data pass through the added tree, it will greatly increase the clock frequency to improve the overall performance.

4.3 Synthesis and Optimization

The basic design implementation flow (shown in Figure 4.3) involves defining the design goals, selecting a compilation strategy, optimizing the design, and analyzing the results to drive the place and route to achieve timing closure. In order to achieve delay, area, and power optimization, the design constraints for each must be defined, and then synthesis

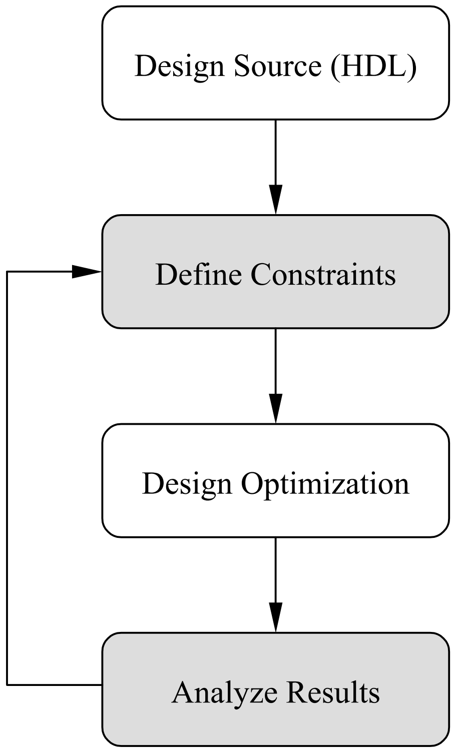


Figure 4.3 Basic design implementation flow

tools can optimize the design and try to meet the constraints. The result can be analyzed for verifying whether the design specification has been met, and then the constraints can be redefined, if necessary. For results of the FPGA implementation, the power information cannot be determined until using Xilinx Xpower after place and route. The results can then be analyzed and then constraints redefined, if necessary. In this research, Design Compiler was used for the high-level synthesis and the Xilinx and Cadence place and route tools were used for the physical level synthesis.

Design Compiler optimizes the RTL design performing both technology-independent optimization as well as technology-specific optimization. Design Compiler removes the existing gate structure from a design, and then rebuilds the design with the goal of improving the design's logic structure. The optimization of logic equations does not affect a particular part of a function; rather, it has a global effect on the overall area or speed characteristics of a design. The following are some of the key features in Design Compiler for using design optimization in this research.

- Flattening

Flattening is an optional logic optimization step that removes all intermediate levels and uses Boolean distributive laws to remove all parentheses. Thus, flattening removes all logic hierarchy from a design. Removing poor intermediate levels enables Design Compiler to choose more-efficient sub-functions. The result of flattening is a two-level, sum-of-products form. Flattening can result in a faster design because the two-level form generally has fewer levels of logic between inputs and outputs. The command "set_flatten true" is used to activate flattening.

- Structuring

Structuring is a logic optimization step that intermediates variables and logic structure to a design. During structuring, Design Compiler searches for sub-functions that can be factored out and then evaluates these factors based on the size of the factor and the number of times the factor appears in the design. Design Compiler structures a design during compilation by default.

- Timing-driven

Design Compiler offers timing-driven structuring to minimize delays and Boolean structuring to reduce area. Timing-driven structuring considers a design's timing constraints during local and global structuring and improves critical paths as necessary. Timing-driven structuring is on by default. When using timing-driven structuring, the accurate timing and clock constraints must be defined. Boolean optimization is used to reduce the area. Boolean optimization structuring uses Boolean algebra to capture non-essential information and reduce circuit equation input format.

- Delay optimization

During the delay optimization phase, Design Compiler creates an optimal net-list representation in the target technology library that meets the timing goals. Design Compiler takes design rules into account for optimizing the delay performance of critical paths in the design. When two circuit solutions offer the same delay performance, Design Compiler implements the solution that has the lower design rule cost.

- Area optimization

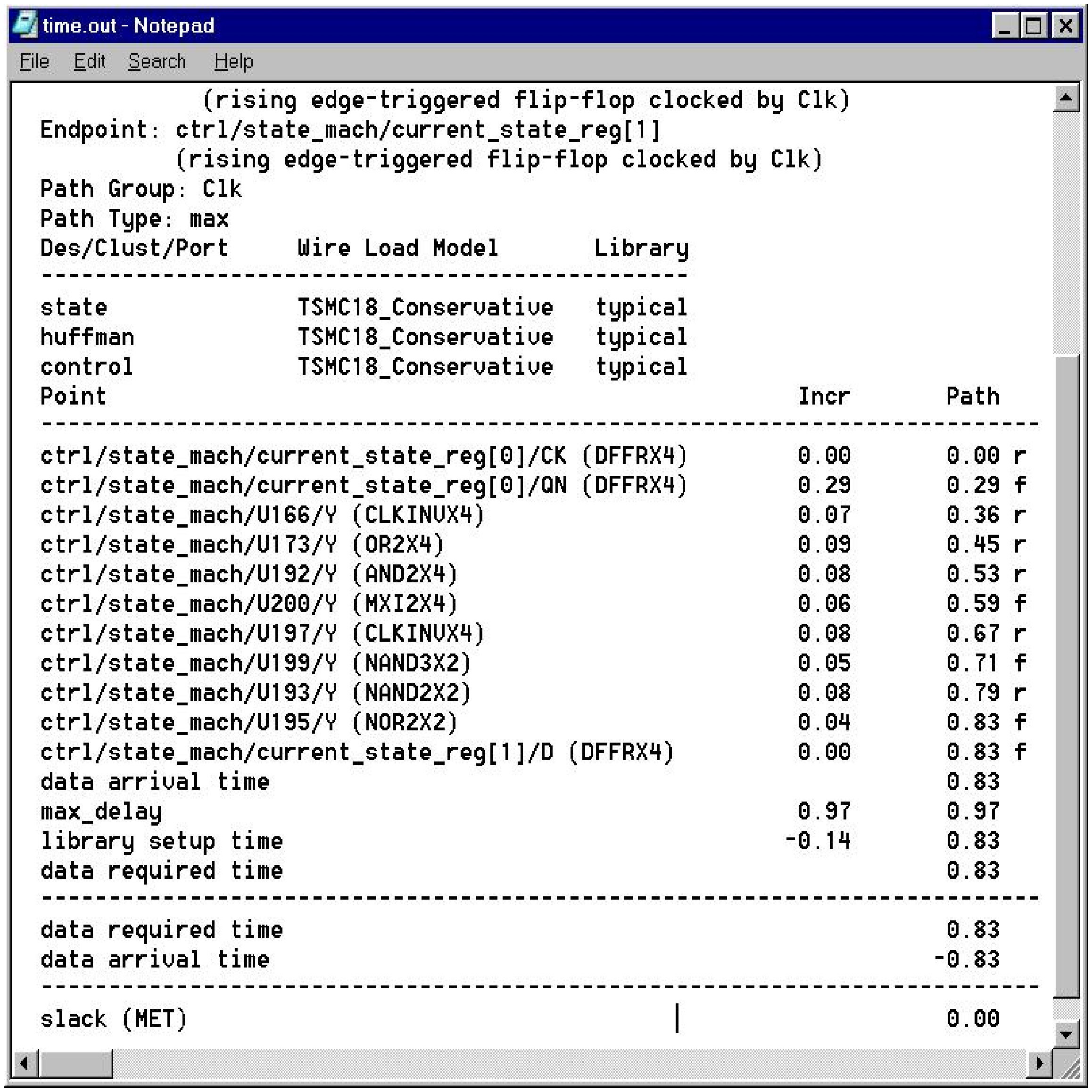
Assuming that the area constraints have been placed on the design, Design Compiler now attempts to minimize the number of gates in the design. The Design Compiler can be directed to put a low, medium, or high effort into area optimization. When low effort is set, Design Compiler performs gate sizing and buffer and inverter cleanup. Design Compiler allocates limited CPU time to this effort level. When medium effort is selected, Design Compiler adds phase assignment to gate sizing and buffer and inverter cleanup. Design Compiler allocates more CPU time to this effort than to a low-effort optimization. When high effort is set, Design Compiler performs additional gate minimization strategies and increases the number of iterations. The tool adds gate composition to the process and allocates even more CPU time.

- Power optimization

This constraint sets the maximum power attributes to the current design. In order to achieve the minimum power, the command is "set_max_dynamic_power 0". This command will set the target power as low as the design could achieve.

- Timing analysis

The purpose of timing analysis is to understand the performance and to ensure that the design meets the design goals after synthesis. Design Compiler automatically determines the maximum and minimum path delay requirements for the design. It examines the clock waveforms at each timing path startpoint and endpoint. A timing path is a path through logic along which signals can propagate. Timing paths normally start at primary inputs or clock pins of registers and end at primary outputs or data pins of registers. Figure 4.3 shows the timing analysis output file. The timing exception command such as “set_max_delay” is used to set the constraint. The timing exception commands operate on a “from” set, a “to” set, and several “through” sets. The -from option indicates a path startpoint, -to indicates a path endpoint, and -through enables control over a specific path or multiple paths between a given startpoint and endpoint. In the case shown in Figure 4.4, the command will be “set_max_delay 0.97 -from startpoint -to end point ”.


time.out - Notepad

File Edit Search Help

```
            (rising edge-triggered flip-flop clocked by Clk)
Endpoint: ctrl/state_mach/current_state_reg[1]
          (rising edge-triggered flip-flop clocked by Clk)
Path Group: Clk
Path Type: max
Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
state              TSMC18_Conservative   typical
huffman            TSMC18_Conservative   typical
control            TSMC18_Conservative   typical
Point                                                   Incr       Path
--------------------------------------------------------------------------
ctrl/state_mach/current_state_reg[0]/CK (DFFRX4)        0.00       0.00 r
ctrl/state_mach/current_state_reg[0]/QN (DFFRX4)        0.29       0.29 f
ctrl/state_mach/U166/Y (CLKINVX4)                       0.07       0.36 r
ctrl/state_mach/U173/Y (OR2X4)                          0.09       0.45 r
ctrl/state_mach/U192/Y (AND2X4)                         0.08       0.53 r
ctrl/state_mach/U200/Y (MXI2X4)                         0.06       0.59 f
ctrl/state_mach/U197/Y (CLKINVX4)                       0.08       0.67 r
ctrl/state_mach/U199/Y (NAND3X2)                        0.05       0.71 f
ctrl/state_mach/U193/Y (NAND2X2)                        0.08       0.79 r
ctrl/state_mach/U195/Y (NOR2X2)                         0.04       0.83 f
ctrl/state_mach/current_state_reg[1]/D (DFFRX4)         0.00       0.83 f
data arrival time                                                  0.83
max_delay                                               0.97       0.97
library setup time                                     -0.14       0.83
data required time                                                 0.83
--------------------------------------------------------------------------
data required time                                                 0.83
data arrival time                                                 -0.83
--------------------------------------------------------------------------
slack (MET)                                                        0.00
```

Figure 4.4   Timing analysis output

CHAPTER 5

Results and Discussions

This chapter presents the result of this research with discussions. The first section shows the result of Huffman encoder implementation and the second section illustrates the results of the FIR filter.

5.1 Huffman Encoder Implementation

In this section, the implementation of the Huffman encoder will be explained. The first part of the implementation is the comparison of different FPGA results using different synthesis tools. This is followed by the detailed explanation of implementation for the Artisan library with different results.

5.1.1 Results for FPGA Implementation

The Design Compiler, FPGA Compiler II, and Mentor Graphics Spectrum are used to synthesize the FPGA. From the design flow shown in Figure 3.1, the synthesis tool can target different libraries to implement different types of ASICs. The target library for FPGA implementation is Xilinx Virtex V1000EHQ240-6. The pre-synthesis simulation is shown in Figure 5.1. The functionality of the Huffman encoder was verified from Chapter Three. The following are the basic scripts for each synthesis tool and explanation.

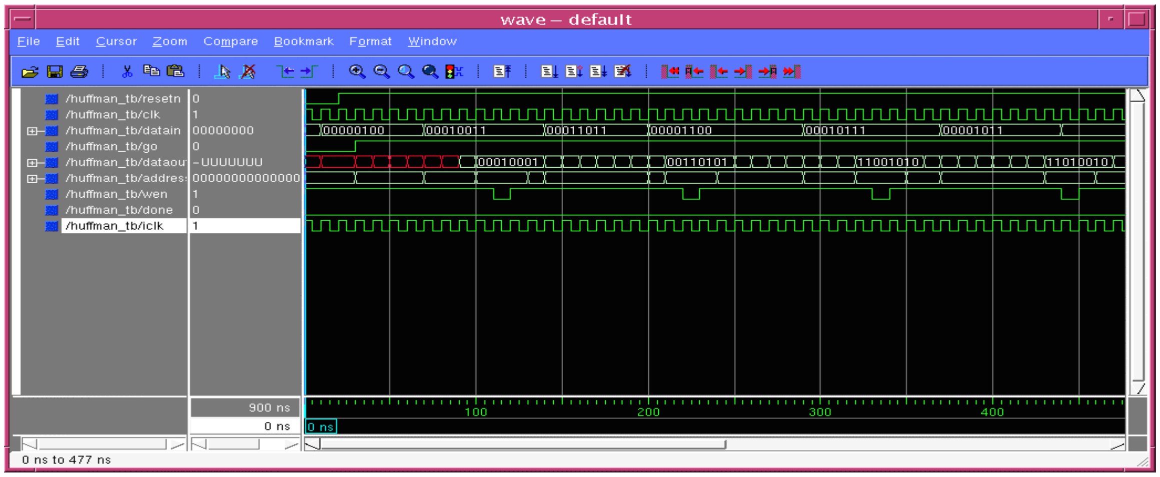


Figure 5.1 Pre-synthesis simulation of Virtex 1000e

- Mentor Graphics Spectrum

set part v1000ehq240
set process 6
set wire_table xcve1000-6_avg
set lut_max_fanout ""
load_library xcve
read -technology "xcve" {
/tnfs/home/cku/thesis/nrfir/new_fir/huff/state.vhd
/tnfs/home/cku/thesis/nrfir/new_fir/huff/control.vhd
/tnfs/home/cku/thesis/nrfir/new_fir/huff/huff.vhd
/tnfs/home/cku/thesis/nrfir/new_fir/huff/huffman.vhd }
pre_optimize -common_logic -unused_logic -boundary -xor_comparator_optimize
pre_optimize -extract
set register2register 20
set input2register 20
set register2output 20
optimize .work.huffman.huffman_arch -target xcve -chip -area -effort quick -hierarchy auto
auto_write -format EDIF /tnfs/home/cku/thesis/nrfir/new_fir/huff/result/huffleo1.edf

The script above is targeted for area optimization. If one wants to change to timing optimization, then -*area* should be changed to -*delay* in the *optimize* command. The other way is to find the maximum delay of the design, then change the time in *set register2register 20* to the time to be constrained, such as *set register2register 6* in this case, and add *optimize_timing .work.huffman.huffman_arch -force* after the *optimize* command. This will force the synthesis tool to optimize the longest path and attempt to meet the constraint.

- FPGA Compiler II

```
create_project test
add_file -format VHDL state.vhd
add_file -format VHDL control.vhd
add_file -format VHDL huff.vhd
add_file -format VHDL huffman.vhd
analyze_file -progress
create_chip -progress -name huffman -target VIRTEXE -device V1000EHQ240 -speed -6 -
frequency 50 -preserve huffman
current_chip huffman
set_chip_objective speed
set_chip_constraint_driven -enable
set_chip_effort high
optimize_chip -progress -name huff1
report_chip
current_chip Huffman
get_pathgroup
create_subpath -from_name p0 -to_name p1 -from_list /huffman/UncodeData_reg<0> -to_list
/huffman/ctrl/state_mach/current_state_reg<2> -maxd 15 (RC,Clk):(RC,Clk)
set_max_delay -path_group p0:p1 15
optimize_chip -progress -name huff2
report_chip
export_chip -pregress -dir result
quit
```

The *create_subpath* command is used to define the critical path of the chip. The command, *set_max_delay -path_group p0:p1 15*, is to set the timing constraint to 15ns. This step can be repeated after each optimization to change the timing constraint of the critical path.

- Design Compiler

```
analyze -f vhdl state.vhd
analyze -f vhdl control.vhd
analyze -f vhdl huff.vhd
analyze -f vhdl huffman.vhd
elaborate huffman
link
uniquify
current_design huffman
create_clock -name "Clk" -period 20 -waveform {0 10} {"Clk"}
set_wire_load_model -name xcv1000e-6_avg -library xdc_virtexe-6
set_port_is_pad "*"
insert_pads
set_cost_priority
set_flatten true -effort high
compile -map_effort high -boundary_optimization
report_timing -max_paths 5
set_max_delay 19 -from "UncodeData_reg<4>" -to "ctrl/state_mach/current_state_reg<0>"
compile -map_effort high -boundary_optimization
write -format edif -hierarchy -output ./result/huffdc2.edf
write_script > ./result/huffdc2.dc
cd result
sh dc2ncf huffdc2.dc
cd ..
quit
```

Before using Design Compiler, the ".synopsys_dc.setup" had to point to the right .db file. In order to find the correct .db file, simply type "synlibs virtexe 1000e-6", and the file will show on the screen. The command *set_wire_load_model -name xcv1000e-6_avg -library xdc_virtexe-6* had to be used to setup the correct wire load. As in the previous chapter, *set_max_delay 19 -from "starting point" -to "ending point"* is used to control the critical path. The *report_timing -max_paths 5* command is to list the 5 longest paths in the chip to help control the critical path.

Table 5.1 shows all the results from different synthesis tools. The delay and number of slices were determined during place and route. The power estimation used Xilinx Xpower to estimate the total consumption. The power results in the table are the values after

Table 5.1 Results from different synthesis tools

Synthesis tool	Delay (ns)	Slices	Power (mW)
Tool A	11.923	47	45
Tool A	7.684	46	45
Tool A	6.079	46	46
Tool B	11.829	63	46
Tool B	11.659	63	46
Tool C	16.462	74	48
Tool C	13.521	70	48
Tool C	12.343	64	48

subtracting the quiescent power consumption, which is 900 mW. Before using Xpower, "huff.vcd" had to be generated. "Huff.vcd" provides detailed design activity rates for all nets. It was generated during the post-layout-synthesis simulation. The following are the scripts for generate "huff.vcd".

- postsim

```
#!/bin/csh -f
source ~cad/.cshrc
mentor_tools
vlib work
vmap simprim /usr/cad/course/simprim
vcom -work work huff_tim_sim.vhd
vcom -work work huffman_tb.vhd
vsim -sdftyp top_inst=huffl_tim_sim.sdf huffman_tb -do stim.do
```

- stim.do

```
vcd file huff.vcd
vcd on
vcd add /*
view wave
add wave *
run 10000ns
```

Figure 5.2 show the flow to generate the "huff.vcd" file for Xpower to estimate power consumption. Figure 5.3 is the Xpower report file. It shows the total estimated power consumption and power consumption for each cell under the respective clock frequency. According to the table, the best result was implemented by Tool A from timing and area point of view. The best result from Tool A was used for post-layout simulation by using Modelsim. The delay of 7 ns is shown in the post-layout simulation (shown in Figure 5.4). The delay from Table 5.1 is 6.079, which is very close to the post-layout simulation result. The layout of the Virtex 1000e is shown in Figure 5.5.

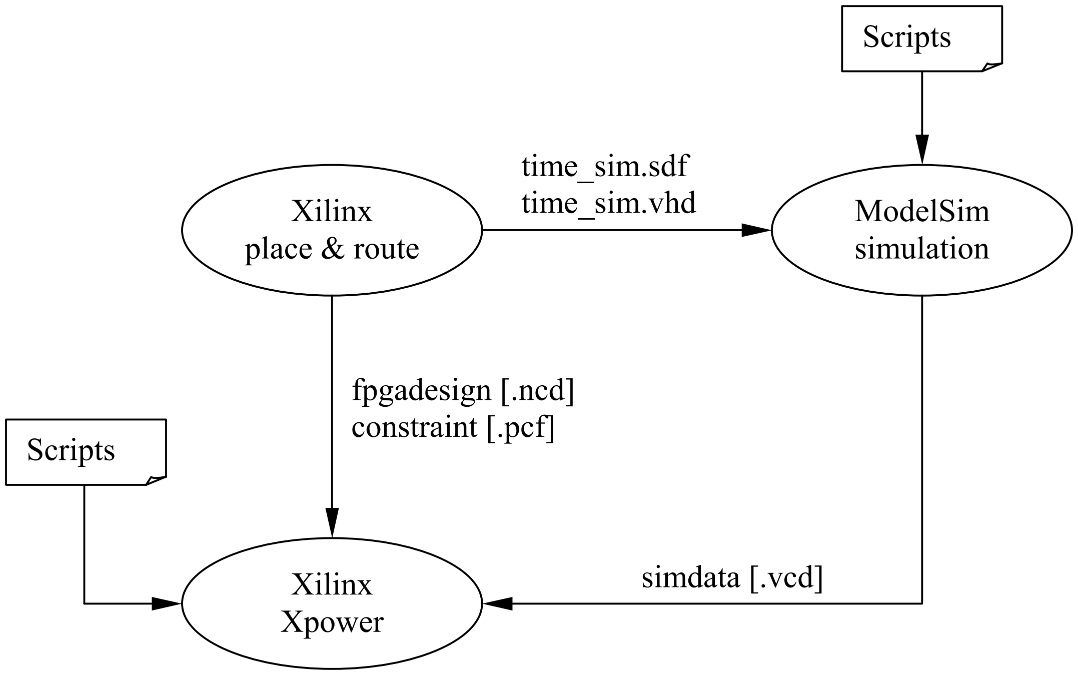
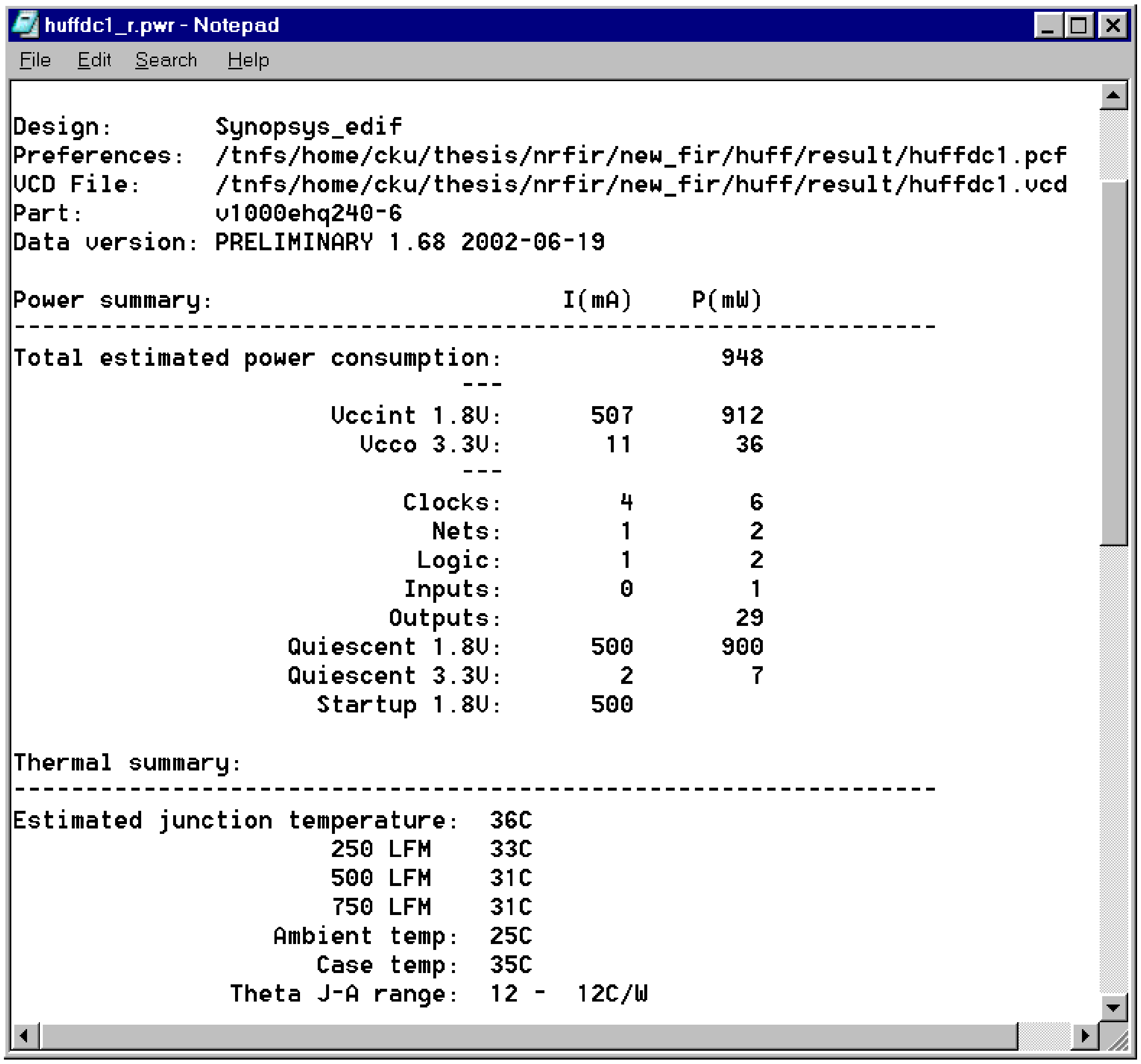
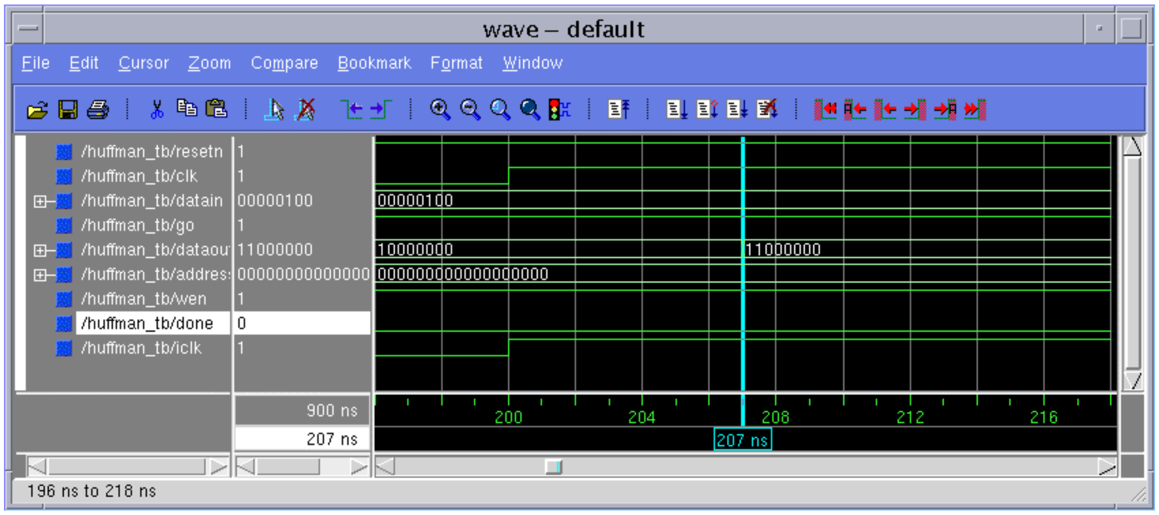


Figure 5.2 Design flow of generating [.vcd] file



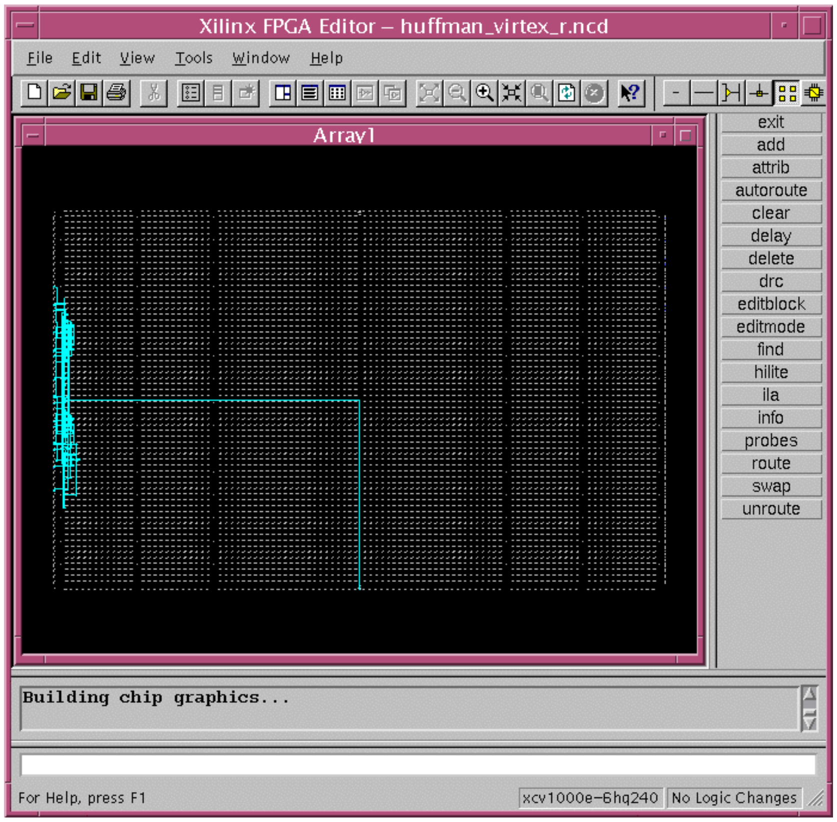
```
Design:       Synopsys_edif
Preferences:  /tnfs/home/cku/thesis/nrfir/new_fir/huff/result/huffdc1.pcf
VCD File:     /tnfs/home/cku/thesis/nrfir/new_fir/huff/result/huffdc1.vcd
Part:         v1000ehq240-6
Data version: PRELIMINARY 1.68 2002-06-19

Power summary:                         I(mA)    P(mW)
----------------------------------------------------------------
Total estimated power consumption:                948
                               ---
                      Vccint 1.8V:      507       912
                        Vcco 3.3V:       11        36
                               ---
                           Clocks:        4         6
                             Nets:        1         2
                            Logic:        1         2
                           Inputs:        0         1
                          Outputs:                 29
                   Quiescent 1.8V:      500       900
                   Quiescent 3.3V:        2         7
                     Startup 1.8V:      500

Thermal summary:
----------------------------------------------------------------
Estimated junction temperature:  36C
                      250 LFM    33C
                      500 LFM    31C
                      750 LFM    31C
                 Ambient temp:   25C
                    Case temp:   35C
               Theta J-A range:  12 -  12C/W
```

Figure 5.3 Output file of Xpower



```
/huffman_tb/resetn   1                 
/huffman_tb/clk      1
/huffman_tb/datain   00000100          00000100
/huffman_tb/go       1
/huffman_tb/dataou   11000000          10000000        11000000
/huffman_tb/addres   000000000000000   000000000000000000
/huffman_tb/wen      1
/huffman_tb/done     0
/huffman_tb/iclk     1
                     900 ns
                     207 ns
196 ns to 218 ns
```

Figure 5.4 Post layout simulation of Virtex 1000e



```
Building chip graphics...
```

For Help, press F1   xcv1000e-6hq240   No Logic Changes

Figure 5.5 Huffman encoder Layout of Virtex 1000e

5.1.2 Results for TSMC-0.18 Implementation

The following sections are the different optimization methods and results for Design Compiler to implement the Huffman encoder. The script for this is similar to the script for FPGA implementation using Design Compiler. Therefore, only the key commands will be discussed in this section. All the power estimations in this section are pre-layout values reported by Design Compiler without the use of switching activity. More accurate results could be obtained following layout using a switching activity and other tools.

5.1.2.1 Results for Timing Optimization

The delay optimization was performed by setting the timing constraint. The first step to optimize timing was to use the "-map_effort high" command. The first set of the delay, area, and power was found. The timing analyzer determined the actual path delays of the design and compared them with the required path delay. The timing analyzer computed each gate and interconnect delay, then traced critical paths, calculating minimum and maximum arrival time to points of interest. The timing analyzer used the critical path values to evaluate design constraints and created timing reports. Then, the timing report was checked to find out the critical path of the design. The new timing constraint was set and then the design was compiled with the new timing constraint using the "set_max_delay" command. The slack in the timing report had to be checked to ensure that it was met not violated after the design was compiled. The steps were repeated until the minimum delay was found. The minimum delay is 0.83 ns, so the highest internal operation frequency is 1.2 GHz. Table 5.2 presents the timing optimization results. Figure 5.6 gives the values in three dimensions.

Table 5.2 Timing optimization results

Timing Constraints (ns)	Delay (ns)	Area (sq.µm)	Power (µW)
-map_effort high	2.43	6885.276855	396.6453
2.23	1.97	6948.478516	399.8609
1.77	1.65	7204.621094	423.2546
1.45	1.31	8086.309082	469.6368
1.11	1	9926.186523	537.3382
0.97	0.83	10415.21777	550.435

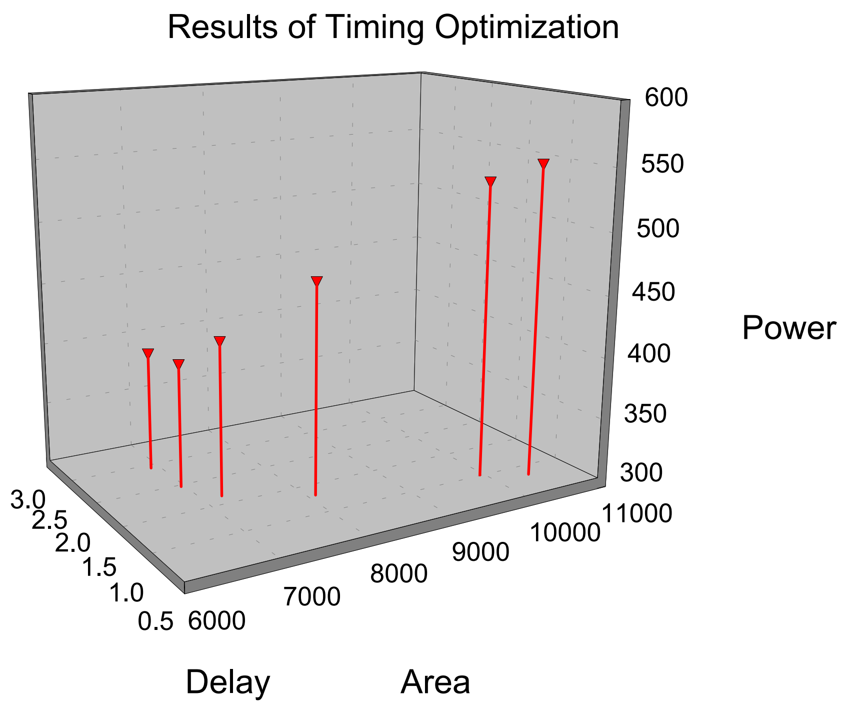


Figure 5.6 Plot results of timing optimization

5.1.2.2 Results for Area Optimization

The command "set_max_area 0" was used to optimize the area. The high, medium, and low effort was put into area optimization, and also combined with a different effort of the area and power optimization. Table 5.3 shown the results for area optimization. The smallest area for this research was 6379.51709 sq. µm. The areas were between 6300 and 6700 sq. µm for area optimization. The areas were between 6800 and 10500 sq. µm for timing optimization. The largest area from area optimization was still smaller than the smallest area from timing optimization. The difference between the smallest area for timing and area optimization is 3.8%. From the comparison of both optimization methods, we found that the timing optimization is critical to area optimization. Figure 5.7 is the 3-D plot of the area optimization.

Table 5.3 Area optimization results

Area Constraints set_max_area 0	Area (sq.µm)	Delay (ns)	Power (µW)
delay, power effort_low area effort_high	6379.51709	2.2	366.1497
delay, power effort_low area effort_medium	6399.48928	2.2	367.4032
delay, power effort_high area effort_high	6416.12305	2.2	392.945
delay, power effort_high area effort_medium	6432.76025	2.18	393.9166
delay, power effort_high area effort_low	6609.20986	2.47	371.0525
delay, power effort_low area effort_low	6639.04443	2.43	398.6882

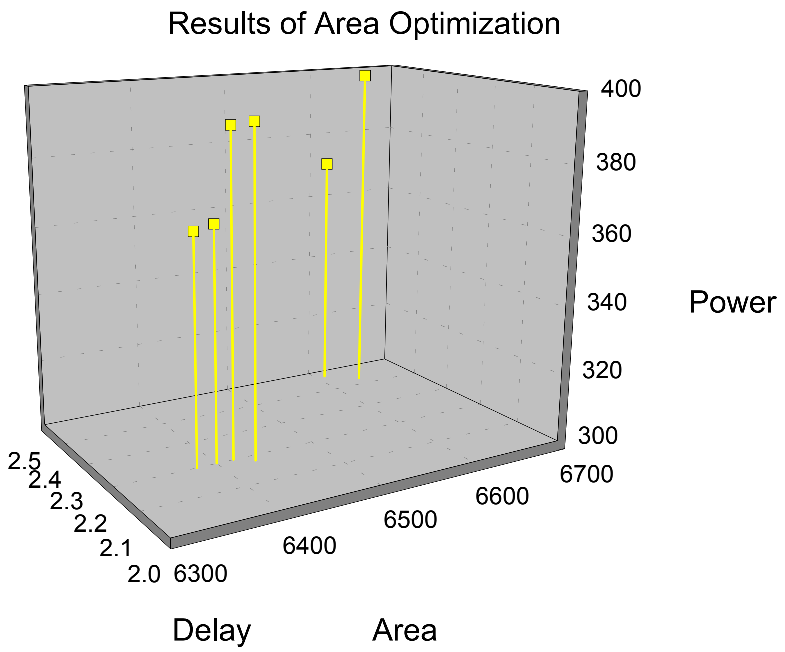


Figure 5.7 Plot results of area optimization

5.1.2.3 Results of Power Optimization

The method of power optimization was similar to that of the area optimization. The command “set_max_dynamic_power 0” was used to minimize the power. The tool was setup for minimum power and maximum area. From the previous section, reducing the area increased more power than the delay of the design. The cells were characterized for power in the target library. The tool selected the low-power equivalent of the device from library TMSC-0.18, which was the library used for this research work. During the power optimization, it also targeted the delay of the design. From Table 5.4, the minimum delay is 1.15 ns and has the power of 433.3379 µW. Comparing the result from timing optimization, the delay of 1.31 ns has the power of 469.6368 µW. The result from power optimization had less delay and used less power. Figure 5.8 is the plot of the results.

Table 5.4 Power optimization results

Power (µW)	Delay (ns)	Area (sq.µm)
326.9737	2.88	7161.495605
324.9893	2.53	7194.774902
339.7595	2.12	7244.647461
344.1441	1.85	7274.599609
371.7438	1.5	7577.355469
433.3379	1.15	8741.692383

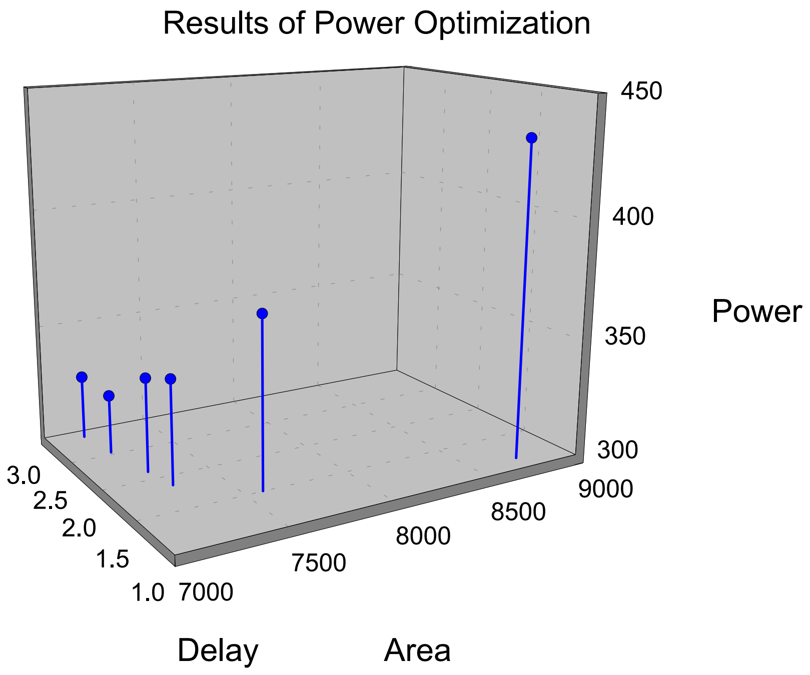


Figure 5.8 Plot results of power optimization

5.1.2.4 Discussions

From the previous sections, the minimum delay of Xilinx Virtex 1000e was 6.079 ns. The minimum delay from the timing optimization was 0.83 ns and the minimum delay between them was a factor of 7.32. The reason the Virtex required more delay time is because the programmable interconnect caused a decrease in speed. Another consideration is the gate capacity for the FPGA. Only 44 out of 12,288 slices were used for the Huffman encoder. It is less than 1% of the available gates. Therefore, understanding the design and then choosing the size of the FPGA for different designs is very important. Section 5.1.2.1 through 5.1.2.3 described the different methods to optimize the design. Figure 5.9 shows the result for each optimization method.

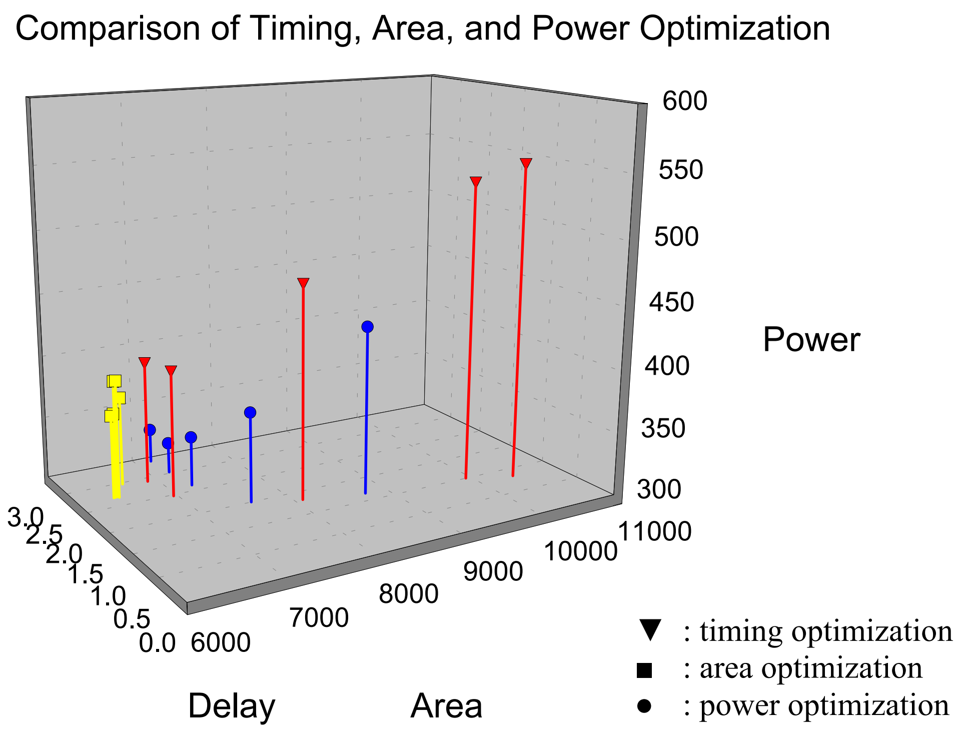


Figure 5.9 Comparison of timing, area, and power optimization

The three different shapes of the points represent three different optimization methods; the triangle is timing optimization, the square is area optimization, and the circle is power optimization. From Figure 5.9, the pattern of each optimization was obvious. The timing optimization only targeted delay, so the area and the power were large compared to the others. The area optimization had the largest area located at 6000 sq. μm, and the power and delay were in the middle range. The power optimization also targeted delay, thus the results were similar to timing optimization, only less power was used and more area consumed.

From Table 5.5, the minimum delay was 0.83 from timing optimization. It had a ratio of 2.63 and 1.39 in comparison with the smallest delay of area optimization and power optimization. If the timing is critical for design specification, the timing optimization will be used since timing optimization will cause large area and high power use.

Table 5.5 Delay for different methods of optimization

Timing	Area	Power
2.43 ns	2.2 ns	2.88 ns
1.97 ns	2.2 ns	2.53 ns
1.65 ns	2.2 ns	2.12 ns
1.31 ns	2.18 ns	1.85 ns
1 ns	2.47 ns	1.5 ns
0.83 ns	2.43 ns	1.15 ns

From Table 5.6, the area from timing optimization was about 1.6 times the area of optimization. Although the timing optimization had a delay ratio of 2.63 with area optimization, if the delay from area optimization had satisfied the design specification, the timing optimization would not have needed to be performed. Reducing the area size will increase the yield and reduce fabrication cost.

From Table 5.7, the maximum power was produced by the timing optimization and the minimum power was produced by the power optimization. Comparing the power dissipation between them, the timing optimization consumed 1.7 times more power than the power optimization. The power dissipation is critical when the chip is operating on portable electronic devices because the less power consumption, the longer the battery operating life.

The net-list was generated after the synthesis. The net-list was used by the Cadence place and route tool to generate the physical layout. The physical layout is shown in Figure 5.10.

Table 5.6 Area for different methods of optimization

Timing	Area	Power
6885.276855	6379.51709	7161.495605
6948.478516	6399.48928	7194.774902
7204.621094	6416.12305	7244.647461
8086.309082	6432.76025	7274.599609
9926.186523	6609.20986	7577.355469
10415.21777	6639.04443	8741.692383

Table 5.7 Power for different methods of optimization

Timing	Area	Power
396.6453	366.1497	326.9737
399.8609	367.4032	324.9893
423.2546	392.945	339.7595
469.6368	393.9166	344.1441
537.3382	371.0525	371.7438
550.435	398.6882	433.3379

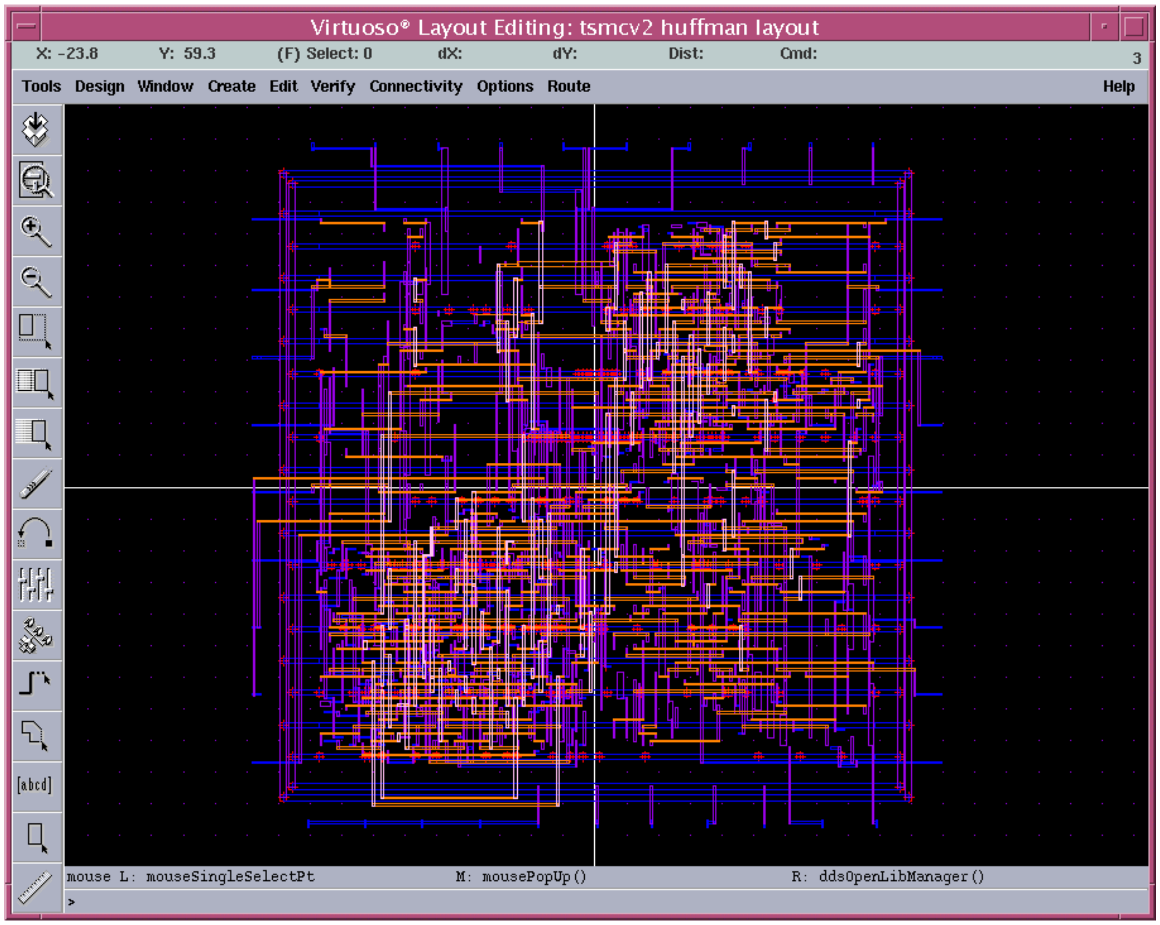


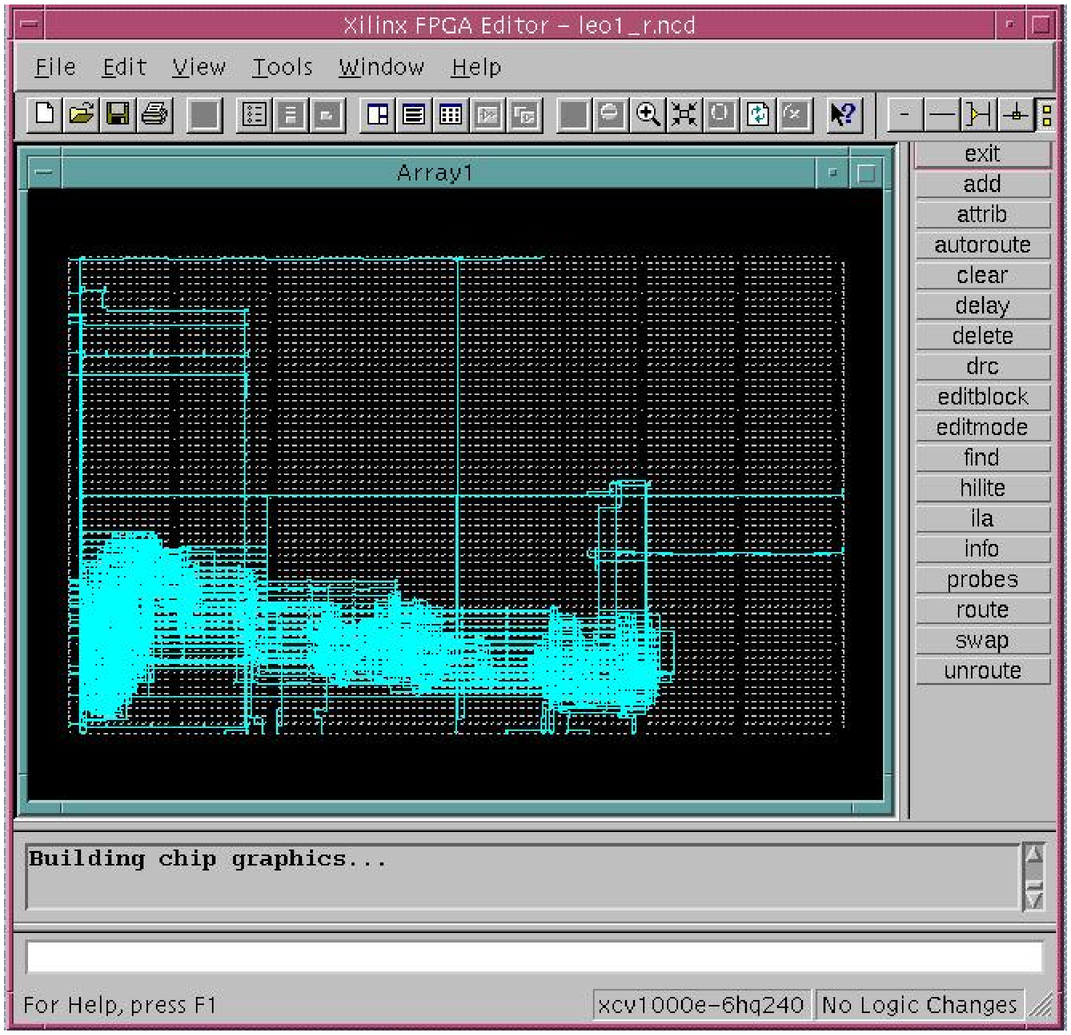
Figure 5.10 Layout of Huffman encoder

5.2 FIR Filter Implementation

The implementation of the FIR filter was targeted to the Xilinx Virtex only. The procedure was similar to the Huffman encoder FPGA implementation. The scripts for each synthesis tool were also similar. From Section 4.2, we knew the added tree for the FIR filter is a large sequential process. Since there are 32 taps in the FIR filter and the input signal is 12-bit, the fanout after the adder tree is more than 1000. The large sequential structural and the large fanout limited the timing constraint. Table 5.8 shows the results from different synthesis tools. As seen in Table 5.8, the Tool A still performed better than Tool B or Tool C. Tool C performed much better for TSMC-0.18 implementation, but not for FPGA implementation. Both results of Huffman and FIR implementation from Tool C, the delay and slices are the worst when compared to Tool A and Tool B, but Tool C had the most control of timing constraint. Figure 5.11 is the layout of Virtex 1000e.

Table 5.8 Results of FIR filter using different Synthesis tools

Synthesis tool	Delay (ns)	Slices	Power (mW)
Tool A	22.668	910	112
Tool A	20.192	910	114
Tool A	19.243	910	109
Tool B	22.382	1350	126
Tool B	21.237	1351	126
Tool C	33.762	2042	143
Tool C	32.611	2044	144
Tool C	30.000	2038	149
Tool C	28.784	2040	144



```
Building chip graphics...
```

Figure 5.11 FIR filter layout of Virtex 1000e

CHAPTER 6

Summary, Conclusion and Future Works

6.1 Summary

In this research work, the Huffman encoder and FIR filter were implemented using VHDL. The design was functionally tested, synthesized, optimized, and placed and routed. Testing of the RTL was performed using Synopsys and Mentor Graphics tool sets. The high-level synthesis and optimization were performed using the Mentor Graphics Spectrum, Synopsys Design Compiler and FPGA Compiler II. The physical level synthesis used Cadence and Xilinx for place and route. The VHDL code for this research work was obtained from Honeywell Inc. and the Boeing Company. A few modifications were made to the code, such as adding in Synopsys libraries, altering the modules, etc. This was done to make the code compatible to the Synopsys and Mentor Graphics tool during synthesis and optimization.

The functional testing of the Huffman encoder was done by using C++ and the input characters were transferred to corresponding binary codes. The corresponding binary codes were saved as “datain.txt”. A test-bench was made for the purpose of simulation. The test-bench read the file and wrote the output as “dataout.txt” during simulation. The result from simulation was compared with the C++ result from Chapter Three.

The three synthesis tools were used to optimize the FIR filter design, then Xilinx place and route tools implemented the synthesis results to the FPGA layout for Virtex 1000e.

The Synopsys synthesis tool, Design Compiler, was used to operate timing, area, and power optimization for different results. The target library for the synthesis tool and place and route tool is TSMC-0.18. The net-list from synthesis was used by Cadence to generate the physical layout.

6.2 Conclusion

Integrated circuit performance is typically characterized by the size of the chip, the speed of operation, the available circuit functionality, and the power consumption. Many market segments are driving the need for smaller, faster and less power-consuming ICs. With the increasing use of portable electronic devices, the emphasis is now on power optimization. The optimization of the timing, delay, and power has to be performed at all design phases for better results. There are three levels to design synthesis, but the RTL/logic level is by far the most efficient.

In this research, a VHDL synthesis methodology for ASIC was implemented. The overall ASIC design cycle can be significantly reduced by the use of VHDL synthesis partly due to the elimination of the schematic generation and maintenance tasks and partially due to the reduction in design error by using higher level VHDL simulation early in the cycle. VHDL synthesis also has the advantage of vendor and technology independence. Most of the synthesis tool vendors provide libraries for a number of different ASIC foundries and technologies.

The high-level synthesis and optimization for TSMC-0.18 was performed using the Synopsys Design Compiler in this research. The timing, area, and power optimization results were shown in Chapter Five. From the results, each method of optimization would affect the result of the other two methods. The maximum speed of the design also consumed maximum power. Using more power not only consumed more energy, but also generated heat. Therefore, the dissipation of heat became another problem. The designer has to focus on the design specification in order to get the best performance out of the chip. For example, if the design calls for critical power efficiency, the designer can concentrate on power optimization method to optimize the design.

From the results of the FPGA implementation, the faster designs did not use more slices and power as TSMC-0.18 did. The increasing of area and power consumption was not significant during timing optimization. The power consumption of the FPGA was much greater than TSMC-0.18.

The ASIC development also involves the “time-to-market” issues. The FPGA implementation has the shortest turn-around time. The result from Xilinx Virtex for the Huffman encoder had 6.067 ns of delay. It is more than 7 times slower than the minimum delay from timing optimization. Because of the shortest turn around time and the lower prototyping cost, the FPGA is still chosen for some designs. With the tremendous growth in technology today, FPGAs are becoming faster, and the number of gates is exponentially increasing annually.

6.3 Future Work

For the Huffman encoder, the layout needs to be put into a frame and the input and output pins need to be connected with pads before the fabrication process. A Smart Frame was developed from the previous class. The Smart Frame has a build-in-self-test (BIST) ability. The signature compressor produces a unique signal pattern to test the functionality of the chip. In addition, the design has to simulate the unique signal pattern in order to find out the corresponding output. The corresponding output will set up the comparator in BIST. The design could be sent to the foundry for fabrication. After receiving the actual chip, if the chip is set in the BIST mode, the design will read the data from the signature compressor and compare it with the simulation result. If there are no problems during the fabrication process, the comparator will send out a "Go" signal. The maximum speed can be tested and compared with the synthesis result.

For the FIR filter, the VHDL code can be modified to reduce the adder tree timing delay due to the sequential operation. That will also reduce the fanout of the adder tree. After modifying the code, the new implementation can be done. The results can be compared with the results of this research. The design can also be downloaded to the Virtex chip and tested to determine the highest operation frequency. The frequency can then be compared with this research.

References

[1] Ivan S. Kourtev, Eby G. Friedman, *Timing Optimization Through Clock Skew Scheduling*, Kluwer Academic Publishers, ISBN 0-7923-7796-6.

[2] Mike Tien-Chien Lee, *High-Level Test Synthesis of Digital VLSI Circuits*, Artech House, Inc. ISBN 8-89006-907-7.

[3] Ahmed Amine Jerraya, Hong Ding, Polen Kission, Maher Rahmouni, *Behavioral Synthesis and Component Reuse with VHDL*, Kluwer Academic Publishers, ISBN 0-7923-9827-0.

[4] Michael John Sebastain Smith, *Application-Specific Integrated Circuits*, Addison Wesley Longman, Inc. ISBN 0-201-50022-1.

[5] Neil H. E. Weste, Kamran Eshraghian, *Principles of CMOS VLSI Design, A Systems Perspective*, Second edition, Addison-Wesley Publishing Company, ISBN 0-201-53376-6.

[6] “Designing Application-Specific Integrated Circuits” http://vlsi1.engr.utk.edu/ece/bouldin_courses/551/overview_bw.pdf

[7] Bernard Sklar, *Digital communications, Fundamentals and Applications*, Prentice-Hall, Inc. ISBN 0-13-211939-0.

[8] Sadiq M. Sait, Habib Youssef, *VLSI Physical Design Automation, Theory and Practice*, IEEE Press, ISBN 0-7803-1411-8.

[9] Paul M. Brown, Jr. A *Guide to Analog ASICs*, Academic Press, Inc. ISBN 0-12-136970-6.

[10] Srinivasa R. Vemuru, Norman Scheinberg, “Short-Circuit Power Dissipation Estimation for CMOS Logic Gates”, *IEEE Transactions on Circuits and Systems*, Vol. 41, No. 11, Nov. 1994.

[11] A. Lu, G. Stenz, F.M. Johannes, “Technology Mapping for Minimizing Gate and Routing Area”, *IEEE Proceedings on Design, Automation and Test in Europe*, February 23-26, 1998, p664-p669.

[12] K. Chakraborty, S. Kulkami, M. Bhattacharya, P. Mazumder, A. Gupta, “A physical design tool for built-in self-repairable RAMs”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume: 9 Issue: 2, April 2001, p352 -p364.

[13] Kiat-Seng Yeo, Samir S. Rofail, Wang-Ling Goh, *CMOS/BiCMOS ULSI, Low Voltage, Low Power*, Prentice Hall PTR, ISBN 0-13-032162-1.

[14] Douglas E. Ott, Thomas J. Wilderotter, *A Designer's Guide to VHDL Synthesis*, Kluwer Academic Publishers, ISBN 0-7923-9472-0.

[15] James H. McClellan, Ronald W. Schafer, Mark A. Yoder, *DSP FIRST: A Multimedia Approach*, Prentice Hall, ISBN 0-13-243171-8.

Appendices

Appendix A: VHDL Code of Huffman Encoder

A1 Test-bench

```
-- Test-bench for huffman.vhd

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use STD.textio.all;
use IEEE.std_logic_textio.all;

entity huffman_tb is
end huffman_tb;

architecture structure of huffman_tb is

  signal Resetn       :    std_logic;
  signal Clk          :    std_logic;
  signal DataIn       :    std_logic_vector(7 downto 0);
  signal Go           :    std_logic;
  signal DataOut      :    std_logic_vector(7 downto 0);
  signal Address      :    std_logic_vector(17 downto 0);
  signal Wen          :    std_logic;
  signal Done         :    std_logic;

  COMPONENT huffman  PORT(
       Resetn      :  IN   std_logic;
       Clk         :  IN   std_logic;
       DataIn      :  IN   std_logic_vector(7 downto 0);
       Go          :  IN   std_logic;
       DataOut     :  OUT  std_logic_vector(7 downto 0);
       Address     :  OUT  std_logic_vector(17 downto 0);
       Wen         :  OUT  std_logic;
       Done        :  OUT  std_logic);
  END COMPONENT;

  constant Period : time := 20 ns;
  SIGNAL iclk : std_logic;

begin
   top_inst: huffman
   PORT MAP(Resetn, Clk, DataIn, Go, DataOut, Address, Wen, Done);

  ClockGen : PROCESS
  BEGIN
    iclk <= '1';
    WAIT FOR Period/2;
    iclk <= '0';
    WAIT FOR Period/2;
  END PROCESS ClockGen;
  Clk <= iclk;
  Resetn <= '0',
              '1' after 20 ns;
  Go <= '0',
      '1' after 60 ns;

Simu : PROCESS (Clk, Resetn, Go, Wen)

  file Fileout : text is out "dataout.txt";
  file Filein  : text is in  "datain.txt";

  variable Lineout : line;
  variable Linein : line;
  variable tmpout : std_logic_vector (7 downto 0);
  variable tmpin : std_logic_vector (7 downto 0);

  begin

  if Clk'event and Clk = '1' then
          if not(endfile(Filein)) then
                    readline ( Filein, Linein);
                    read (Linein, tmpin);
                    DataIn <= tmpin;
                    if Wen = '0' then
                              tmpout := DataOut;
                              write (Lineout, tmpout);
                          writeline(Fileout, Lineout);
                    end if;
          end if;
  end if;
END PROCESS Simu;
end structure;

configuration func_config of huffman_tb is
for structure
   for top_inst: huffman
      use entity WORK.huffman(BEHAVIORAL);
   end for;
end for;
end func_config;
```

Appendix B: C++ Code for Design Verification

B1 Huffman.cpp

```
/*-----------------------------------------------------*/
/*                                                     */
/*  Program for verification of Huffman encoder        */
/*                                                     */
/*  This program will take input of character          */
/*  string then use Huffman encode theory to           */
/*  convert the character string into binary           */
/*  code.                                              */
/*                                                     */
/*-----------------------------------------------------*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <fstream.h>

void main()
{
        int k;
        char inp[200];

        for ( k = 0 ; k < 200 ; k++ )
        {
                inp[k] = '\0';
        }

        ifstream File("huff_input.txt");
        File.read(inp,200);
        File.close();

        char huff[1200], temp[1200];
        int count=0;
        int flag=0;

        for ( k = 0 ; k < 200 && inp[k] != '\0'; k++ )

        {
                switch (inp[k])
                {
                        case 'q':
                                temp[flag+5]='0';
                temp[flag+4]='1';
                temp[flag+3]='1';
                temp[flag+2]='1';
                temp[flag+1]='1';
                temp[flag]='0';
                flag = flag + 6;
                break;
        case 'v':
                temp[flag+5]='0';
                temp[flag+4]='1';
                temp[flag+3]='1';
                temp[flag+2]='1';
                temp[flag+1]='1';
                temp[flag]='1';
                flag = flag + 6;
                break;
        case 39:
                temp[flag+5]='1';
                temp[flag+4]='0';
                temp[flag+3]='0';
                temp[flag+2]='1';
                temp[flag+1]='1';
                temp[flag]='0';
                flag = flag + 6;
                break;
        case '"':
                if (count ==0)  /* begin quote */
                  {
                        temp[flag+5]='1';
                        temp[flag+4]='0';
                        temp[flag+3]='0';
                        temp[flag+2]='1';
                        temp[flag+1]='1';
                        temp[flag]='1';
                        flag = flag + 6;
                        count = 1;
                }
                else  /* end quote */
                {
                        temp[flag+5]='1';
                        temp[flag+4]='0';
                        temp[flag+3]='1';
                        temp[flag+2]='0';
                        temp[flag+1]='0';
                        temp[flag]='0';
                flag = flag + 6;
                count = 0;
        }
        break;
case '9':
        temp[flag+5]='1';
        temp[flag+4]='0';
        temp[flag+3]='1';
        temp[flag+2]='0';
        temp[flag+1]='0';
        temp[flag]='1';
        flag = flag + 6;
        break;
case '5':
        temp[flag+5]='1';
        temp[flag+4]='0';
        temp[flag+3]='1';
        temp[flag+2]='0';
        temp[flag+1]='1';
        temp[flag]='0';
        flag = flag + 6;
        break;
case '(':
        temp[flag+5]='1';
        temp[flag+4]='0';
        temp[flag+3]='1';
        temp[flag+2]='0';
        temp[flag+1]='1';
        temp[flag]='1';
        flag = flag +6;
        break;
case ')':
        temp[flag+5]='1';
        temp[flag+4]='1';
        temp[flag+3]='0';
        temp[flag+2]='0';
        temp[flag+1]='0';
        temp[flag]='0';
        flag = flag + 6;
        break;
case '1':
        temp[flag+5]='1';
        temp[flag+4]='1';
        temp[flag+3]='0';
        temp[flag+2]='0';
```

```
            temp[flag+1]='0';
            temp[flag]='1';
            flag = flag + 6;
            break;
    case '2':
            temp[flag+5]='1';
            temp[flag+4]='1';
            temp[flag+3]='0';
            temp[flag+2]='1';
            temp[flag+1]='0';
            temp[flag]='0';
            flag = flag + 6;
            break;
    case '-':
            temp[flag+5]='1';
            temp[flag+4]='1';
            temp[flag+3]='0';
            temp[flag+2]='1';
            temp[flag+1]='0';
            temp[flag]='1';
            flag = flag + 6;
            break;
    case '.':
            temp[flag+5]='1';
            temp[flag+4]='1';
            temp[flag+3]='0';
            temp[flag+2]='1';
            temp[flag+1]='1';
            temp[flag]='0';
            flag = flag + 6;
            break;
    case 'l':
            temp[flag+5]='1';
            temp[flag+4]='1';
            temp[flag+3]='0';
            temp[flag+2]='1';
            temp[flag+1]='1';
            temp[flag]='1';
            flag = flag +6;
            break;
    case 'y':
            temp[flag+4]='0';
            temp[flag+3]='0';
            temp[flag+2]='0';
            temp[flag+1]='0';
            temp[flag]='0';
            flag = flag + 5;
            break;
    case 'g':
            temp[flag+4]='0';
            temp[flag+3]='0';
            temp[flag+2]='0';
            temp[flag+1]='0';
            temp[flag]='1';
            flag = flag + 5;
            break;
    case 'h':
            temp[flag+4]='0';
            temp[flag+3]='0';
            temp[flag+2]='1';
            temp[flag+1]='1';
            temp[flag]='0';
            flag = flag + 5;
            break;
    case 'f':
            temp[flag+4]='0';
            temp[flag+3]='0';
            temp[flag+2]='1';
            temp[flag+1]='1';
            temp[flag]='1';
            flag = flag + 5;
            break;
    case 'p':
            temp[flag+4]='0';
            temp[flag+3]='1';
            temp[flag+2]='0';
            temp[flag+1]='0';
            temp[flag]='0';
            flag = flag + 5;
            break;
    case 't':
            temp[flag+4]='0';
            temp[flag+3]='1';
            temp[flag+2]='0';
            temp[flag+1]='0';
            temp[flag]='1';
            flag = flag + 5;
            break;
    case 'd':
            temp[flag+4]='0';
            temp[flag+3]='1';
            temp[flag+2]='1';
            temp[flag+1]='0';
            temp[flag]='0';
            flag = flag + 5;
            break;
    case 'm':
            temp[flag+4]='0';
            temp[flag+3]='1';
            temp[flag+2]='1';
            temp[flag+1]='0';
            temp[flag]='1';
            flag = flag + 5;
            break;
    case 'u':
            temp[flag+4]='0';
            temp[flag+3]='1';
            temp[flag+2]='1';
            temp[flag+1]='1';
            temp[flag]='0';
            flag = flag + 5;
            break;
    case 'a':
            temp[flag+4]='1';
            temp[flag+3]='0';
            temp[flag+2]='0';
            temp[flag+1]='0';
            temp[flag]='0';
            flag = flag + 5;
            break;
    case 'c':
            temp[flag+4]='1';
            temp[flag+3]='0';
            temp[flag+2]='0';
            temp[flag+1]='0';
            temp[flag]='1';
            flag = flag + 5;
            break;
    case 's':
            temp[flag+4]='1';
            temp[flag+3]='0';
            temp[flag+2]='0';
            temp[flag+1]='1';
            temp[flag]='0';
            flag = flag + 5;
```

```
                break;
        case 'i':
                temp[flag+4]='1';
                temp[flag+3]='1';
                temp[flag+2]='0';
                temp[flag+1]='0';
                temp[flag]='1';
                flag = flag + 5;
                break;
        case 'o':
                temp[flag+3]='0';
                temp[flag+2]='0';
                temp[flag+1]='0';
                temp[flag]='1';
                flag = flag + 4;
                break;
        case 'n':
                temp[flag+3]='0';
                temp[flag+2]='0';
                temp[flag+1]='1';
                temp[flag]='0';
                flag = flag + 4;
                break;
        case 'r':
                temp[flag+3]='0';
                temp[flag+2]='1';
                temp[flag+1]='0';
                temp[flag]='1';
                flag = flag + 4;
                break;
        case 'e':
                temp[flag+3]='1';
                temp[flag+2]='0';
                temp[flag+1]='1';
                temp[flag]='1';
                flag = flag + 4;
                break;
        case ' ':
                temp[flag+2]='1';
                temp[flag+1]='1';
                temp[flag]='1';
                flag = flag + 3;
                break;
}
```

```
        }

        int l;
        cout << "The huffman code is:" << endl << endl;

        for (l = 0 ; l < flag ; l++ )
        {
                huff [7 + l - 2*(l%8)] = temp[l];
        }

        ofstream a_file("huff_code.txt");

        for ( l = 0 ; l < (flag + (8 - flag % 8)) ; l++ )
        {
                if (l >= (flag - (flag%8)) && l < (flag + 8 - 2*(flag % 8)))
                        a_file << " ";
                        cout << " ";
                else
                {
                        a_file << huff[l];
                        cout << huff[l];

                        if ((l % 8) == 7)
                        {
                                a_file << endl;
                                cout << " ";
                        }
                }
        }

        a_file.close();

        char yorn;
        cout << endl << endl;
        cout << "Do you want to exit? (y/n)  ";
        cin  >> yorn;
        cout << endl;
        if (yorn== 'y')
                exit (1);

}
```

Vita

Chung Ku was born in Pingtung, Taiwan in 1971. He graduated from Senior Industrial Vocational School, National Changhua University of Education in June 1989. He entered the Department of Mechanical Engineering of the National Chin-Yi Institute of Technology in August 1990. He finished his Associate degree in June 1992. After serving in the Navy for two years, Chung Ku completed his military duty with honor from the Department of Defense in August 1994.

Chung Ku came to the United States in January 1997 and started his undergraduate study at The University of Tennessee, Knoxville, majoring in Electrical Engineering. He was initiated into Eta Kappa Nu Honor Society. He also received the Engineering Academic Achievement Award from the department of Electrical Engineering in Spring 1999. Chung completed his undergraduate courses in the Spring of 1999 and received the Bachelor of Science degree in Electrical Engineering.

He continued his graduate education at The University of Tennessee, Knoxville. He joined Dr. Walker's group with a Graduate Research Assistantship in August 1999. After Dr. Walker left UTK, he received a Graduate Teaching Assistantship in May 2001.