**A High-Throughput VLSI Architecture Design of Canonical Huffman Encoder**

**Base paper Abstract :**

Abstract—In this brief, a high-throughput Huffman encoder VLSI architecture based on the Canonical Huffman method is proposed to improve the encoding throughput and decrease the encoding time required by the Huffman code word table construction process. We proposed parallel computing architectures for frequency-statistical sorting and code-size computational sorting. This architecture results in a process of building a tree and assigning symbols that can be completed by scanning the data only once. This solves the problem of the low efficiency of the traditional algorithm, which needs to scan the data twice. Consequently, in addition to the advantages of the high compression ratio inherited from the Canonical Huffman, the proposed architecture has overridden advantages for a high parallelism processing capacity. The experimental results showed that the proposed architecture decreased the encoding time by 26.30% compared to the available Huffman encoder using the standard algorithm when encoding 256 8-bit symbols. Furthermore, the VLSI architecture could further decrease the encoding time when encoding more 8-bit symbols. In particular, when encoding 212,642 8-bit symbols, the proposed VLSI architecture could reduce the encoding time by 87.40%. Thus, compared with the traditional Huffman encoders, this brief achieved the improvement of coding efficiency.

**Enhancement of this Project :**

* To Develop this design of High Throughput Hardware design of Canonical Huffman Machine will take 160 Data Bits and compressed data bits is 90 Data Bits with compared all the parameters in terms of area, delay and power.

**Proposed Title :**

**FPGA Implementation of High Throughput Lossless Canonical Huffman Machine Encoder**

**Proposed Abstract :**

In a recent technology of digital network will transfer and receive the data with more complexity due to number of data bits and number of memory operations, thus it will take more data losses and low throughputs. Therefore this proposed work of this paper present lossless data compression with less memory architecture using Canonical Huffman compression technique. In this case here, Huffman machine will present lossless and less memory configuration and its support multi bit operations on data compression. Here, this technique will present input as 640 Data Bits and compressed output as 90 Data bits using variable length with Canonical Huffman encoding method. Finally this work will present in Verilog HDL and synthesized in Vertex FPGA and get the results of area, delay and power.

**Existing System :**

HUFFMAN coding has many applications in the fields of data compression, image processing, audio compression, and data security. As a crucial component of the Huffman coding process, the “code word table” accurately reflects the data compressible space. To obtain a precise code word table, the input symbols have to be pre-scanned before the compression is started. This mechanism causes the input data to be processed twice, which results in a low coding speed and high hardware cost. Generally, a known code word table is adopted in the available commercial algorithms to remove the pre-scan procedure. However, the table has a high compression ratio only for the input data whose specific frequency distribution is suitable for the recommended code word table. Otherwise, it possesses a lower compression ratio. An efficient memory allocation scheme for Huffman coding using a known code word table was proposed. It can considerably reduce the computational burden for memory allocation for the Huffman table with little performance degradation. However, a large number of clock cycles is needed to search the Huffman code from the Huffman table. In a new data structure to improve the efficiency of Huffman coding was proposed. Nevertheless, the parameters of the data structure were generated through a series of complex calculations, leading to a low clock frequency. A PLA solution was proposed to achieve fast Huffman coding, but it typically needs a large amount of hardware to store the code word table. The authors of also proposed a CAM utilization method to store the code word table, in which the code word table is reconstructed with the currently encoded data and updated in real-time. However, the complexity is too high for hardware implementation. In this brief, a high-throughput Huffman encoder VLSI architecture based on the Canonical Huffman encoding method is proposed to overcome the shortcomings of the available designs. We propose parallel computing architectures for frequency statistical sorting and code-size computational sorting. This architecture allows the process of building a tree and assigning symbols to be completed by scanning the data only once, which greatly improves the coding efficiency. In contrast, the standard Canonical Huffman algorithm needs to scan the input data twice to complete the above process. Consequently, in addition to the advantages of the high compression ratio inherited from the Canonical Huffman algorithm, the proposed architecture has overridden advantages for a high parallelism processing capacity.

Due to its high compression ratio, Canonical Huffman encoding is widely used in compression algorithms. It mainly consists of two phases: creating a binary tree of nodes and coding based on this tree. This scheme causes the nodes to be scanned twice, which results in a longer encoding time. This means that if we want to achieve a higher compression rate, we must sacrifice coding efficiency. As is shown in Fig. 1, the tree building procedure is composed of three processes: frequency-statistics process used to determine the frequency of each node, the sorting process used to find two nodes with the minimum and sub-minimum frequencies, and the node creating process. In the node creating process, the two nodes produced by the sorting process will be selected as the children of a newly created node whose frequency is the summary of these two selected nodes. Meanwhile, the code-size of the two selected nodes will be increased by 1.

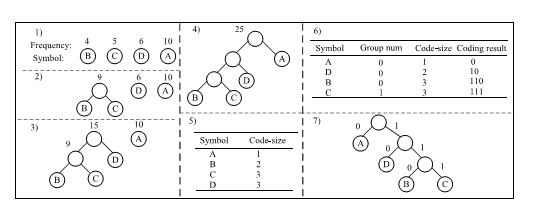
****

Figure 1 : Process of Canonical Huffman encoding

**Disadvantages :**

* Encoding efficiency is low regarding delay and power consumption.
* Less Throughput.
* More number of Memories.

**Proposed System :**

In a recent technology of digital network will transfer and receive the data with more complexity due to number of data bits and number of memory operations, thus it will take more data losses and low throughputs. Therefore this proposed work of this paper present lossless data compression with less memory architecture using Canonical Huffman compression technique. In this case here, Huffman machine will present lossless and less memory configuration and its support multi bit operations on data compression. Here, this technique will present input as 640 Data Bits and compressed output as 90 Data bits using variable length with Canonical Huffman encoding method. Finally this work will present in Verilog HDL and synthesized in Vertex FPGA and get the results of area, delay and power.

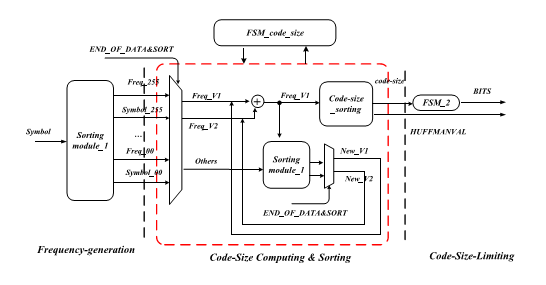


Figure 2 : System circuit block diagram.

As shown in Fig. 2, the proposed design is composed of three stages: Frequency-Generation, Code-Size Computing & Sorting, and Code-Size-Limiting. To improve the encoding efficiency and make up for the shortcomings of the Canonical Huffman encoder, in the first two stages, we propose two types of real-time frequency-sorting architectures that “eat” the input symbol in series schemes. These two architectures are given in detail in Sections III-B and III-C. Based on this hardware architecture, the code-size-sorting module generates a temporary sorted result of the code-size data queue at every clock cycle. Ultimately, once the code-size calculation process is completed, the HUFFMANVAL results will be given simultaneously. This brief also proposes an efficient VLSI architecture for the Code-Size-Limiting stage that is used to limit the bit length to improve the encoding speed. As the last stage is designed based on the standard algorithms, this biref optimizes the nesting of the standard algorithms to effectively reduce the circuit area and power consumption. As shown in Fig. 3, compared with the traditional Huffman encoder designs, the proposed architecture can reduce the required clock cycle effectively.

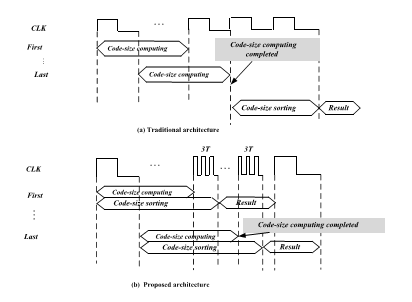


Figure 3 : Timing diagram comparison between the proposed architecture and the traditional designs.

**Advantages :**

* Encoding efficiency is high regarding delay and power consumption.
* High Throughput.
* Less number of Memories.

# **INTRODUCTION TO VLSI DESIGN:**

* 1. What is VLSI?

VLSI stands for "Very Large Scale Integration". This is the field which involves packing more and more logic devices into smaller and smaller areas. VLSI, circuits that would have taken board furls of space can now be put into a small space few millimeters across! VLSI circuits are everywhere ... your computer, your car, your brand new state-of-the-art digital camera, the cell-phones, and what have you. All this involves a lot of expertise on many fronts within the same field, which we will look at in later sections.

Dealing with VLSI Circuits

The way normal blocks like latches and gates are implemented is different from what students have seen so far, but the behavior remains the same. All the miniaturization involves new things to consider. A lot of thought has to go into actual implementations as well as design.

**Circuit Delays:** Large complicated circuits running at very high frequencies have one big problem to tackle - the problem of delays in propagation of signals through gates and wire even for areas a few micrometers across! The operation speed is so large that as the delays add up, they can actually become comparable to the clock speeds.

**Power:** Another effect of high operation frequencies is increased consumption of power. This has two-fold effect - devices consume batteries faster, and heat dissipation increases. Coupled with the fact that surface areas have decreased, heat poses a major threat to the Stability of the circuit itself.

**Layout:** Laying out the circuit components is task common to all branches of electronics. What’s so special in our case is that there are many possible ways to do this; there can be multiple layers of different materials on the same silicon, there can be different arrangements of the smaller parts for the same component and soon. The choice between the two is determined by the way we chose the layout the circuit components. Layout can also affect the fabrication of VLSI chips, making it either easy or difficult to implement the components on the silicon.

Introduction to VHDL

A digital system can be described at different levels of abstraction and from different points of view. An HDL should faithfully and accurately model and describe a circuit, whether already built or under development, from either the structural or behavioral views, at the desired level of abstraction. Because HDLs are modeled after hardware, their semantics and use are very different from those of traditional programming languages.

Limitations of traditional programming languages

There are wide varieties of computer programming languages, from Frontend to C to Java. Unfortunately, they are not adequate to model digital hardware. To understand their limitations, it is beneficial to examine the development of a language. A programming language is characterized by its syntax and semantics. The syntax comprises the grammatical rules used to write a program, and the semantics is the “meaning” associated with language constructs. When a new computer language is developed, the designers first study the characteristics of the underlying processes and then develop syntactic constructs and their associated semantics to model and express these characteristics.

Most traditional general-purpose programming languages, such as C, are modeled after a sequential process. In this process, operations are performed in sequential order, one operation at a time. Since an operation frequently depends on the result of an earlier operation, the order of execution cannot be altered at will. The sequential process model has two major benefits. At the abstract level, it helps the human thinking process to develop an algorithm step by step. At the implementation level, the sequential process resembles the operation of a basic computer model and thus allows efficient translation from an algorithm to machine instructions.

The characteristics of digital hardware, on the other hand, are very different from those of the sequential model. A typical digital system is normally built by smaller parts, with customized wiring that connects the input and output ports of these parts. When signal changes, the parts connected to the signal are activated and a set of new operations is initiated accordingly. These operations are performed concurrently, and each operation will take a specific amount of time, which represents the propagation delay of a particular part, to complete. After completion, each part updates the value of the corresponding output port. If the value is changed, the output signal will in turn activate all the connected parts and initiate another round of operations. This description shows several unique characteristics of digital systems, including the connections of parts, concurrent operations, and the concept of propagation delay and timing. The sequential model used in traditional programming languages cannot capture the characteristics of digital hardware, and there is a need for special languages (i.e., HDLs) that are designed to model digital hardware.

VHDL includes facilities for describing logical structure and function of digital systems at a number of levels of abstraction, from system level down to the gate level. It is intended, among other things, as a modeling language for specification and simulation. We can also use it for hardware synthesis if we restrict ourselves to a subset that can be automatically translated into hardware.

VHDL arose out of the United States government’s Very High Speed Integrated Circuits (VHSIC) program. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed. It was subsequently developed further under the auspices of the Institute of Electrical and Electronic Engineers (IEEE) and adopted in the form of the IEEE Standard 1076, Standard VHDL Language Reference Manual, in 1987. This first standard version of the language is often referred to as VHDL-87.

After the initial release, various extensions were developed to facilitate various design and modeling requirements. These extensions are documented in several IEEE standards:

1. IEEE standard 1076.1-1999, VHDL Analog and Mixed Signal Extensions (VHDL-AMS): defines the extension for analog and mixed-signal modeling.
2. IEEE standard 1076.2-1996, VHDL Mathematical Packages: defines extra mathematical functions for real and complex numbers.
3. IEEE standard 1076.3- 1997, Synthesis Packages: defines arithmetic operations over a collection of bits.
4. IEEE standard 1076.4-1995, VHDL Initiative towards ASK Libraries (VITAL): defines a mechanism to add detailed timing information to ASIC cells.
5. IEEE standard 1076.6-1999, VHDL Register Transfer Level (RTL) Synthesis: defines a subset that is suitable for synthesis.
6. IEEE standard 1 164- 1993 Multivalve Logic System for VHDL Model Interoperability (std-logicJl64): defines new data types to model multivalve logic.
7. IEEE standard 1029.1-1998, VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES): defines how to use VHDL to exchange information in a simulation environment.

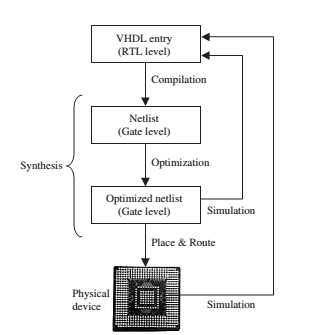


Figure 3:Summary of VHDL design flow

Electronic Design Automation

There are several EDA (Electronic Design Automation) tools available for circuit synthesis, implementation, and simulation using VHDL. Some tools (place and route, for example) are ordered as part of a vendor’s design suite (e.g., Altera’s Quartus II, which allows the synthesis of VHDL code onto Altera’s CPLD/FPGA chips, or Xilinx’s ISE suite, for Xilinx’s CPLD/FPGA chips). Other tools (synthesizers, for example), besides being ordered as part of the design suites, can also be provided by specialized EDA companies (Mentor Graphics, Synopsis, Simplicity, etc.). Examples of the latter group are Leonardo Spectrum (a synthesizer from Mentor Graphics), Simplify (a synthesizer from Simplicity), and Modelsim (a simulator from Model Technology, a Mentor Graphics company). The designs presented in the book were synthesized onto CPLD/FPGA devices (appendix A) either from Altera or Xilinx. The tools used were either ISE combined with ModelSim, MaxPlus II combined with Advanced

Synthesis Software or Quartus II. Leonardo Spectrum was also used occasionally. Although different EDA tools were used to implement and test the examples presented in the design, we decided to standardize the visual presentation of all simulation graphs. Due to its clean appearance, the waveform editor of MaxPlus II was employed. However, newer simulators like ISE þ ModelSim and Quartus II, over a much broader set of features, which allow, for example, a more refined timing analysis. For that reason, those tools were adopted when examining the fine details of each design.

Field-Programmable Gate Array

A field-programmable gate array (FPGA) is a [semiconductor](http://en.wikipedia.org/wiki/Semiconductor) device that can be configured by the customer or designer after manufacturing—hence the name "[field-programmable](http://en.wikipedia.org/wiki/Field-programmable)". To program an FPGA one must specify how they want the chip to work with a logic [circuit diagram](http://en.wikipedia.org/wiki/Circuit_diagram) or a [source code](http://en.wikipedia.org/wiki/Source_code) in a hardware description language (HDL). FPGAs can be used to implement any logical function that an [application-specific integrated circuit](http://en.wikipedia.org/wiki/Application-specific_integrated_circuit) (ASIC) could perform, but the ability to update the functionality after shipping offers advantages for many applications.

FPGAs contain [programmable logic](http://en.wikipedia.org/wiki/Programmable_logic_device) components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"—somewhat like a one-chip programmable [breadboard](http://en.wikipedia.org/wiki/Breadboard). Logic blocks can be configured to perform complex [combinational functions](http://en.wikipedia.org/wiki/Combinational_logic), or merely simple [logic gates](http://en.wikipedia.org/wiki/Logic_gate) like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple [flip-flops](http://en.wikipedia.org/wiki/Flip-flop_(electronics)) or more complete blocks of memory.

For any given [semiconductor process](http://en.wikipedia.org/wiki/Semiconductor_fabrication_plant), FPGAs are usually slower than their fixed [ASIC](http://en.wikipedia.org/wiki/ASIC) counterparts. They also draw more power, and generally achieve less functionality using a given amount of circuit complexity. But their advantages include a shorter [time to market](http://en.wikipedia.org/wiki/Time_to_market), ability to re-program in the field to fix bugs, and lower [non-recurring engineering](http://en.wikipedia.org/wiki/Non-recurring_engineering) costs. Vendors can also take a middle road by developing their hardware on ordinary FPGAs, but manufacture their final version so it can no longer be modified after the design has been committed.

Field Programmable Gate Array (FPGA) devices were introduced by Xilinx in the mid-1980s. They differ from CPLDs in architecture, storage technology, number of built-in features, and cost, and are aimed at the implementation of high performance, large-size circuits.

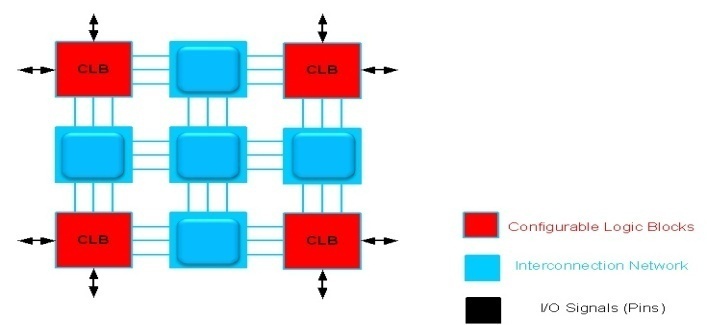


Figure 4: FPGA Architecture

The basic architecture of an FPGA is illustrated in figure 2. It consists of a matrix of CLBs (Configurable Logic Blocks), interconnected by an array of switch matrices.

The internal architecture of a CLB is different from that of a PLD First, instead of implementing SOP expressions with AND gates followed by OR gates (like in SPLDs), its operation is normally based on a LUT (lookup table). Moreover, in an FPGA the number of flip-flops is much more abundant than in a CPLD, thus allowing the construction of more sophisticated sequential circuits. Besides JTAG support and interface to diverse logic levels, other additional features are also included in FPGA chips, like SRAM memory, clock multiplication (PLL or DLL), PCI interface, etc. Some chips also include dedicated blocks, like multipliers, DSPs, and microprocessors.

Another fundamental difference between an FPGA and a CPLD refers to the storage of the interconnects. While CPLDs are non-volatile (that is, they make use of anti fuse, EEPROM, Flash, etc.), most FPGAs use SRAM, and are therefore volatile. This approach saves space and lowers the cost of the chip because FPGAs present a very large number of programmable interconnections, but requires an external ROM. There are, however, non-volatile FPGAs (with antifuse), which might be advantageous when reprogramming is not necessary.

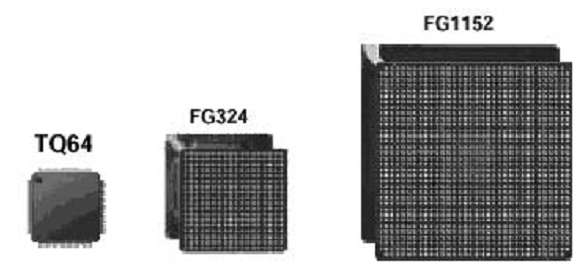


Figure 5: Examples of FPGA Packages

FPGAs can be very sophisticated. Chips manufactured with state-of-the-art0.09mmCMOS technology, with nine copper layers and over 1,000 I/O pins, are currently available. A few examples of FPGA packages are illustrated in figure A6, which shows one of the smallest FPGA packages on the left (64 pins), a medium-size package in the middle (324 pins), and a large package (1,152 pins) on the right. Several companies manufacture FPGAs, like Xilinx, Actel, Altera, and Quick Logic.

Notice that all Xilinx FPGAs use SRAM to store the interconnects, so are reprogrammable, but volatile (thus requiring external ROM). On the other hand, Actel FPGAs are non-volatile (they use anti fuse), but are non-reprogrammable (except one family, which uses Flash memory). Since each approach has its own advantages and disadvantages, the actual application will dictate which chip architecture is most appropriate.

* 1. MODELSIM - ALTRA

Assumptions

I assume that you are familiar with the use of your operating system. You should also be familiar with the window management functions of your graphic interface: Open Windows, OSF/Motif, CDE, KDE, GNOME, or Microsoft Windows 2000/XP. We also assume that you have a working knowledge of the language in which your design and/or test bench is written (i.e., VHDL, Verilog, etc.). Although ModelSim™ is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal.

Modelsim introduction

ModelSim is a verification and simulation tool for VHDL, Verilog, System Verilog, and mixed language designs. This lesson provides a brief conceptual overview of the ModelSim simulation environment. It is divided into four topics, which you will learn more about in subsequent lessons.

Basic Simulation Flow

The following diagram shows the basic steps for simulating a design in ModelSim.



Figure 6 : Basic Simulation Flow

Basic Simulation Flow - Overview Lab [Creating the Working Library]:

In ModelSim, all designs are compiled into a library. You typically start a new simulation in ModelSim by creating a working library called "work," which is the default library name used by the compiler as the default destination for compiled design units.

* **Compiling Your Design**

After creating the working library, and compile your design units into it. The ModelSim library format is compatible across all supported platforms. Its can simulate your design on any platform without having to recompile your design. Loading the Simulator with Your Design and Running the Simulation with the design compiled, load the simulator with your design by invoking the simulator on a top-level module (Verilog) or a configuration or entity/architecture pair (VHDL).

Assuming the design loads successfully, the simulation time is set to zero, and you enter a run command to begin simulation.

* **Debugging Your Results**

If you don’t get the results you expect, you can use ModelSim’s robust debugging environment to track down the cause of the problem.

Project Flow

A project is a collection mechanism for an HDL design under specification or test. Even thoughyou don’t have to use projects in ModelSim, they may ease interaction with the tool and are useful for organizing files and specifying simulation settings. The following diagram shows the basic steps for simulating a design within a ModelSim project.

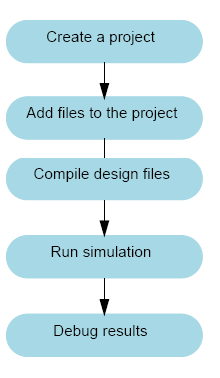


Figure 7 : Project flow

As you can see, the flow is similar to the basic simulation flow. However, there are two important differences:

* Do not have to create a working library in the project flow; it is done for you automatically.
* Projects are persistent. In other words, they will open every time you invoke ModelSim unless you specifically close them.

Multiple Library Flow

ModelSim uses libraries in two ways: 1) as a local working library that contains the compiled version of your design; 2) as a resource library. The contents of your working library will change as you update your design and recompile. A resource library is typically static and serves as a parts source for your design. It can create your own resource libraries, or they may be supplied by another design team or a third party (e.g., a silicon vendor). It specifies which resource libraries will be used when the design is compiled, and there are rules to specify in which order they are searched. A common example of using both a working library and a resource library is one where your gate-level design and test bench are compiled into the working library, and the design references gate-level models in a separate resource library. The diagram below shows the basic steps for simulating with multiple libraries.

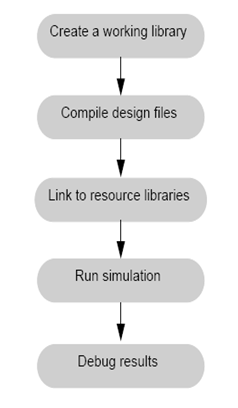


Figure 8 : Multiple Library flow

Debugging Tools

Model Sim offers numerous tools for debugging and analyzing your design. Several of these tools are covered in subsequent lessons, including:

* Using projects
* Working with multiple libraries
* Setting breakpoints and stepping through the source code
* Viewing waveforms and measuring time
* Viewing and initializing memories
* Creating stimulus with the Waveform Editor
* Automating simulation

Basic Simulation

Introduction

In this lesson you will go step-by-step through the basic simulation flow:

1. Create the Working Design Library

2. Compile the Design Units

3. Load the Design

4. Run the Simulation

Design Files for this Lesson

The sample design for this lesson is a simple 8-bit, binary up-counter with an associated test bench. The pathnames are as follows:

**Verilog** – *<install\_dir>/examples/tutorials/verilog/basicSimulation/counter.v* and t*counter.v*

**VHDL** – *<install\_dir>/examples/tutorials/vhdl/basicSimulation/counter.vhd* and *tcounter.vhd*

This lesson uses the Verilog files *counter.v* and *tcounter.v*. If you have a VHDL license, use

*Counter.vhd* and *tcounter.vhd* instead. Or, if you have a mixed license, feel free to use the

Verilog test bench with the VHDL counter or vice versa.

* 1. Xilinx ISE

The steps required to perform behavioral and post-route simulations using the Xilinx Integrated Software Environment (ISE) and Mentor Graphics ModelSim simulator. In the past, Xilinx bundled the ISE with a licensed edition of ModelSim, resulting in little or no issues between the two softwares. However, in recent years it appears as if the two companies are no longer on level terms. As a result, Mentor Graphics does not provide Xilinx with licenses for ModelSim and Xilinx for its part do not provide pre-compiled device libraries for ModelSim. This tutorial will show you how to resolve compatibility issues, so that you can continue to use the features that these software’s provide.

As with any software tutorial, there is always more than one way of doing things. If you come across better solutions or typos or have any suggestions for improving this document please email the author or contact Dr. Grantner.

Pre-requisite

Before reading any further you should have the following software's installed on your computer.

Simulation Libraries

* Simulation Library Compilation Wizard
* Compile HDL Libraries through Project Navigator

Setting up Environment Variables

Before we discuss the individual methods, let us first understand how ModelSim accesses these libraries. A list of the compiled libraries and their locations is stored in modelsim.ini file found in the installation directory. When ModelSim starts a simulation it searches for this ini file; first, in the local project folder, then in the installation directory of ModelSim. If a particular library is not found, the simulation stops with an error. Both the methods discussed above cannot access the modelsim.ini \_le directly, so it is important that we setup environment variables that point to the location of this \_le. Setting up environment variables requires administrative privileges on your computer. If you do not have these privileges skip to the next section.

1. The first step is to change the write permissions for the modelsim.ini file. Browse to the ModelSim install directory. In our setup this is c:\modeltech64 10.0c (this will be different for other versions). Right-click on modelsim.ini, select Properties and uncheck Read-only.
2. Make a backup of the modelsim.ini file for safety reasons.
3. The easiest way to view or modify the environment variables is by right-clicking on My ComputerProperties; click advanced system settingsEnvironment Variables.
4. Under System variables, click New; enter modelsim for the variable name. For the Variable value enterc: \modeltech64 10.0cnmodelsim.ini.
5. Similarly, add another variable named Path with the value c:\modeltech64 10.0cnwin64.For our setup; win64 is where the executable (modelsim.exe) is located. If a variable called path already exists, make sure you append to the existing values. Note that multiple values can be separated by semi-colon.
6. Click OK to save these settings.

Simulation Library Compilation Wizard

This method is highly recommended because library compilation can be performed for more than one device family and/or language.

1. Open the Simulation Library Compilation Wizard. This can be accessed from Start MenuProgramsXilinx ISE Design Suite 13.xISE Design Tools32-bit Tools

**Note:** If you are using 64-bit version of ModelSim use the Simulation Library Compilation Wizard from 64-bit Tools. ModelSim PE Student Version 10.x is only available in the 32-bit version.

1. The Select Simulator window opens up. Select the appropriate simulator (For Student Edition select ModelSim PE); enter c:\modeltech64 10.0cnwin64 for executable location, compxlib.cfg for Compxlib Configuration File and compxlib.log for Compxlib Log File.
2. Next select the HDL used for simulation. If you are unsure select Both VHDL and Verilog. However, this will increase the compilation time and the disk space required.
3. Then select all the device families that you will be working with. Again the more number of devices, more the compilation time and the disk space required. Remember that you can always run the compilation wizard at a later time for additional devices.
4. The next window is for selecting libraries for Functional and Timing Simulation. Different libraries are required for different types of simulation (behavioral, post-route, etc.). We suggest that you select All Libraries as the default option. Interested users can refer to Chapter 6 of the Xilinx Synthesis and Simulation Design Guide for additional information.
5. Finally the window for Output directory for compiled libraries is shown. We suggest leaving the default values that Xilinx picks. Then select Launch Compile Process.
6. Be patient as the compilation can take a long time depending on the options that you have chosen.
7. The compile process may have contained a lot of warnings but should be error-free. We have not explored the reasons behind these warnings, but they do not appear to affect the simulation of any of our designs.
8. Once the process is completed, open c:\modeltech64 10.0cnmodelsim.ini and verifies if there are libraries pointing to the output directory entered in step 6. This will happen only if you have set the environment variables.

Library compilation is now complete. If you have not set the environment variables then the wizard creates a modelsim.ini in the output directory entered in step 6. By default this location is c:\Xilinxn13.xnISE DSnISE. Open this file and verify that it contains the location of the libraries that were just compiled. This file should be copied into every project you create.

**Literature Survey :**

* Suzanne Rigler, William Bishop, Andrew Kennings, Department of Electrical and Computer Engineering University of Waterloo Waterloo, Ontario, Canada, N2L 3G1, 0840-7789/07/$25.00 ©2007 IEEE. FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms, Lossless data compression algorithms are widely used by data communication systems and data storage systems to reduce the amount of data transferred and stored. GZIP is a popular, patent-free compression program that delivers good compression ratios. This paper presents hardware implementations for the LZ77 encoders and Huffman encoders that form the basis for a full hardware implementation of a GZIP encoder. The designs have been implemented as state machines in VHDL in such a way that they are suitable for implementation using either FPGA or ASIC technologies. Performance metrics and resource utilization results obtained for a prototype implementation running on an Altera DE2 board are presented. Ultimately, the goal is to utilized the LZ77 encoders and Huffman encoders described in this paper to build a fully-functional, hardware design for a GZIP encoder that could be used in data communication systems and data storage systems to boost overall system performance1 .
* Yi Chen, Guo Chun Wan, Ling Yi Tang, and Mei Song Tong Department of Electronic Science and Technology, Tongji University, Shanghai, China, 2017 Progress In Electromagnetics Research Symposium — Fall (PIERS — FALL), Singapore, 19–22 November, Huffman Coding Method Based on Parallel Implementation of FPGA, With the rapid development of science and technology, the requirement of data compression speed is getting higher and higher. In recent years, software implementation of image compression technology in the general-purpose computer or DSP chip cannot meet the real-time processing speed requirements. Field Programmable Gate Array (FPGA) is a new type of digital circuit. Each logic gate in the FPGA chip performs some logical operation at the same time every clock cycle. Obviously, FPGA is essentially a large-scale parallel hardware device. In this paper, the parallel processing features of FPGA and parallel Huffman coding are utilized. Different from the traditional sorting algorithm, this paper uses the parallel characteristics of FPGA, using parallel full comparison algorithm. By using this algorithm, it will take only one clock and a small number of clock cycles to get all the data sorted. And the FPGA implementation exports the results through a three-stage pipeline.
* Shireesha Thummala1 ,Thrisul Kumar. J, Swarna latha. E, Vignan Institute of Technology and Aeronautical Engineering, Vignan Hills, Hyderabad, International Journal of Engineering Research & Technology (IJERT) Vol. 3 Issue 2, February - 2014, FPGA Implementation of Huffman Encoder and Decoder for High Performance Data Transmission, In computer science and information theory, Huffman coding is an entropy encoding algorithm used for lossless data compression. The purpose of this paper is to present and analyze HUFFMAN CODING ALGORITHM for the data compression and decompression. Huffman coding is a minimal variable character coding based on the frequency of each character. First, each character becomes a trivial binary tree, with the character as the only node. The character‘s frequency is the tree’s frequency. Two trees with the least frequencies are joined as the sub trees of a new root that is assigned the sum of their frequencies. This is repeated until all characters are in one tree. One code bit represents each level. Thus more frequent characters are near the root and are coded with few bits, and rare characters are far from the root and are coded with many bits. In this paper Huffman encoder and decoder are designed in VHDL. Huffman decoding is done by using a state diagram approach. ModelSim simulator tool from Mentor Graphics will be used for functional simulation and verification of the encoder & decoder modules. The Xilinx Synthesis Tools (XST) will be used to synthesize the complete design on Xilinx family FPGA.
* S. H. Kang, “Embedded STT-MRAM for energy-efficient and cost effective mobile systems,” in IEEE Symp. VLSI Circuits Dig. Tech. Papers, Jun. 2014, pp. 36–37-STT-MRAM is a logic-friendly non volatile memory that can realize a combination of high speed, low energy, and high endurance. Embedded STT-MRAM is positioned attractively not only for emerging low standby-power connectivity systems such as wearables, IOT (Internet-of-Things), and secure elements, but also for high-performance mobile SOC as an embedded non volatile working memory. With recent breakthroughs in CoFeB-based perpendicular magnetic tunnel junctions (MTJ), embedded STT-MRAM has become more energy-efficient and cost-effective in conjunction with robust data retention, scalable for advanced logic nodes.
* P. Amato, C. Laurent, M. Sforzin, S. Bellini, M. Ferrari, and A. Tomasoni, “Ultra fast, two-bit ECC for emerging memories,” in Proc. 6th IEEE Int. Memory Workshop (IMW), May 2014, pp. 79–82-Emerging Memories (EMs) could benefit from Error Correcting Codes (ECCs) able to correct a few errors in just a few nanoseconds; for example to cope with failure mechanisms that could arise in new storage physics. Fast ECCs are also desired for executed-in-Place (XiP) and DRAM applications. This paper shows the key elements to implement a BCH code able to correct 2 errors in a page of 256 data bits in no more than 10ns with 180nm-CMOS logic, and with low energy consumption. The decoding time can be further reduced to few ns using smaller gate length logics. Moreover, the proposed solution is soundly rooted in BCH theory, and can be applied to any user data size. Basically the ideas are to avoid the division in the computation of the coefficients of the Error Locator Polynomial (ELP) of the BCH code, to optimize the implementation of the multiplication in the Galois Fields (GF) and to fully implement the decoder in a parallel combinatorial architecture. Such a BCH code has been embedded in a 45nm 1Gbit Phase Change Memory (PCM) device.

**Reference :**

[1] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” Proc. IRE, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[2] S. J. Sarkar, N. K. Sarkar, and A. Banerjee, “A novel Huffman coding based approach to reduce the size of large data array,” in Proc. Int. Conf. Circuit Power Comput. Technol. (ICCPCT), Nagercoil, India, 2016, pp. 1–5.

[3] Y. Liu and L. Luo, “Lossless compression of full-surface solar magnetic field image based on Huffman coding,” in Proc. IEEE 2nd Inf. Technol. Netw. Electron. Autom. Control Conf. (ITNEC), Chengdu, China, 2017, pp. 899–903.

[4] N. Markandeya and S. Patil, “Improve information rate in Thien and Lin’s image secret sharing scheme using Huffman coding technique,” in Proc. Int. Conf. Comput. Commun. Control Autom. (ICCUBEA), Pune, India, 2017, pp. 1–5.

[5] R. B. Patil and K. D. Kulat, “Audio compression using dynamic Huffman and RLE coding,” in Proc. 2nd Int. Conf. Commun. Electron. Syst. (ICCES), Coimbatore, India, 2017, pp. 160–162.

[6] N. H. Kumar, R. M. Patil, G. Deepak, and B. M. Murthy, “A novel approach for securing data in IoTcloud using DNA cryptography and Huffman coding algorithm,” in Proc. Int. Conf. Innovat. Inf. Embedded Commun. Syst. (ICIIECS), Coimbatore, India, 2017, pp. 1–4.

[7] S. V. Keerthy, T. K. C. R. Kishore, B. Karthikeyan, V. Vaithiyanathan, and M. M. A. Raj, “A hybrid technique for quadrant based data hiding using Huffman coding,” in Proc. Int. Conf. Innovat. Inf. Embedded Commun. Systems (ICIIECS), Coimbatore, India, 2015, pp. 1–6.

[8] Information Technology—Digital Compression and Coding of Continuous-Tone Still Images: Requirements and Guidelines, Standard ISO/IEC 10918-1:1994, 2017.

[9] Information Technology—Generic Coding of Moving Pictures and Associated Audio Information—Part 2: Video, Standard ISO/IEC 13818- 2:2013, 2019.

[10] S. J. Lee, K. H. Yang, J. S. Song, and C. W. Lee, “An efficient memory allocation scheme for Huffman coding of multiple sources,” Signal Process. Image Commun., vol. 14, pp. 311–323, Jan. 1999.