

Design of a Huffman data encoder architecture

Nor Alina Khairi^{1a}, Asral Bahari Jambek^{1b}, Nor Asilah Khairi^{1c}

¹School of Microelectronic Engineering, Universiti Malaysia Perlis, Pauh Putra Campus, 02600 Arau, Perlis, Malaysia

^anoralina89@gmail.com, ^basral@unimap.edu.my, ^casilah.khairi91@gmail.com

Abstract. Wireless sensor networks (WSNs) are important in today's technology for helping to monitor our environment. WSNs are widely used in military, medical and industrial environments. It is capable of monitoring, collecting and transmitting data to a primary server wirelessly. Wireless sensor nodes are powered by a limited energy supply such as a small battery or an energy harvester that generally produces a small amount of energy. To extend the lifetime of the device, energy consumption must be reduced. Data transmission is known to consume the largest percentage of energy in a sensor node. One method for reducing energy is by compressing the data prior to transmitting it. This study analyses the performance of the Huffman architecture in terms of compressing data that are commonly used in wireless sensor nodes. The primary module in the architecture comprises a data retriever, frequency calculator, probability calculator, Huffman tree generator and Huffman code generator. From the experimental results, the Huffman circuit architecture simulation consumed 51394 clock cycles to completely compress 366 data samples, using 3.729mW of power consumption. Based on a 20MHz clock frequency, this is equivalent to 9.5824 μ J of energy consumption. Based on our analyses, the Huffman tree generator consumed the most power at 1.184mW, equivalent to 31.75% of overall power consumption.

Keywords: Wireless Sensor Node; Huffman architecture; reduce energy; power consumption

Introduction

The use of wireless communication devices have been increasing due to its low maintenance and portable features [1]. This has resulted in the rapid development of wireless sensor nodes. The function of this device is to monitor and collect data that will later be transmitted wirelessly to a main server or station [2]. Such a system can be implemented in many different environments due to its wireless characteristics, e.g., in the contexts of agriculture and food industry [3].

Due to the small size of these nodes, energy and storage capacities are limited. Thus, energy consumption is one of the problems that occur when using wireless sensor node devices [1]. Researchers need to find approaches for reducing power consumption in order to increase the device's lifetime and without frequently replacing their batteries.

It has been established that transmission modules consume the most energy within sensor nodes [1]. This is due to the significant energy needed for powering up the wireless transmitter or receiver in order to transmit or receive data. One approach for reducing energy usage is to compress data prior to transmitting it. The amount of data required to be

transmitted is in this way reduced, which will reduce the energy used in the transmission process. The higher the compression ratio of the data, the more energy can be saved.

This study is written as follows. Section 2 discussed on work done related to a number of data compression algorithm and Huffman algorithm architectures in the present literature. The method used in this study is described in section 3. In section 4, it presents the results and discussions upon the conducted research, which will be presented in detail. Section 5 concludes this work.

Literature Review

A survey was conducted on various data compression algorithms that compress different types of data formats and included the Huffman and Lempel-Ziv-Welch (LZW) algorithms [4]. This survey was conducted in order to suggest an efficient algorithm with respect to its corresponding data types. The data used in this work included .DOC, .TXT, .BMP, .TIF, .GIF and .JPG files. For text file data, the results for both algorithms were nearly the same. The results for the LZW algorithm were better than for the Huffman algorithm in the case of .BMP image file compression. For .GIF and .JPG data types, the results after compression were larger than the original file size.

Experiments including various types of data compression for compressing text data were conducted [5]. The purpose of these experiments was to conclude the algorithm best suited to text data. In this experiment, run length encoding (RLE), static Huffman, adaptive Huffman, Shannon Fano, arithmetic and LZW data compression algorithms were used to compress text data. The LZW algorithm achieved the highest saving percentages among all algorithms. Both types of Huffman algorithms and the Shannon Fano algorithm yielded an average saving percentage, whereas the RLE algorithm had the lowest saving percentage. Although LZW obtained the highest saving percentage, a problem occurred; as the file's size increased, the dictionary size also increased; this as due to an increase in entries for the input data.

A study was conducted on various types of data compression using text data as an input [6]. The study primarily focused on the bit per character (bpc) of each type of data compression. The study was divided into two categories: statistical compression technique and dictionary based compression technique. For statistical compression technique, a RLE algorithm achieved an average of 7.93bpc, rendering it inefficient for compressing the data, as the bits for a character was the longest among all algorithms. The Shannon Fano algorithm achieved 5.50bpc, which was the second longest. Huffman and adaptive Huffman coding achieved 5.27bpc and 5.21bpc, respectively, which was considered to be moderate. The lowest bpc achieved was for arithmetic coding, which was 5.51bpc. For the Dictionary-based compression technique, Lempel-Ziv-77 (LZ-77) achieved 3.88bpc, which was the highest among the other three algorithms. Lempel-Ziv-Storer-Szymanski (LZSS), Lempel-Ziv-Huffman (LZH) and Lempel-Ziv-Bell (LZB) each achieved 3.32bpc, 3.22bpc and 3.11bpc, respectively. Lempel-Ziv-78 (LZ-78) achieved 4.26bpc, while the highest bpc was achieved by LZW at 4.90bpc. Lempel-Ziv-Fiala-Green (LZFG) achieved the lowest bpc at 2.89bpc.

A study of hardware architectures on two stage lossless data compression and a decompression algorithm was conducted [7]. The purpose of the study was to increase the

performance of the algorithm and to reduce the size of the designed architecture. The chosen algorithms used in the work were a combination of the Parallel Dictionary Lempel-Ziv-Welch (PDLZW) algorithm and an approximated adaptive Huffman dynamic-block (AHDB) algorithm. This study is an update to the previous study conducted by Lin, Lee & Jan (2006) [8], using a content-addressable memory (CAM) dictionary in PDLZW changed to CAM-tag-based and CAM-based ordered list in AHDB changed to memory-inter-reference (MIR) that uses two static random access memory (SRAM). Improvements were made to reduce hardware cost and to increase the performance of the algorithm.

A simple and area-efficient very large-scale integration (VLSI) architecture was proposed for Huffman coding [9]. The memory storage supports real-time encoding and decoding. A few simple arithmetic operations were also performed on the chip for encoding and decoding processes. In the work, a simple scheme that maps the Huffman tree onto memory was proposed; $2n \times (\log_2 n + 1)$ bits of memory for $\log_2 n$ -bit symbols were required for the scheme where n denoted the number of leaves in the Huffman tree and where n is a power of 2. An average of $O(\log n)$ time units was required to encode or decode a symbol.

Methodology

This section describes the methods and data used in this work. The Huffman algorithm is firstly presented, followed by discussions of the block diagrams of the Huffman architecture.

A Huffman algorithm is a technique where the symbols or characters in a sample are converted into binary code. The high occurrence of symbols or characters is converted into shorter binary codes, whereas the less frequent occurrences are converted into longer binary code. Figure 1 shows the flow chart of the Huffman encoder. Before Huffman code can be generated, all symbols in the sample were initially read. Then, the probabilities for each symbol were calculated using the following equation,

$$\text{Probability of a symbol } (i) = \text{Frequency of symbol } (i) \div \text{Total symbols in a sample} \quad (1)$$

where i is the symbol in a data sample. Next, the number of calculated probabilities was checked. If it only contained one probability, the next step followed; where there was more than one probability, these were sorted from high value to low value probabilities. Two lowest probabilities were then added to create a new probability. The process was repeated until there was only one probability left. In the next part, binary code was assigned between the connectors of the two probabilities. A higher value probability is assigned as high (1) and a lower probability value as low (0). This began from the highest, most likely probability, which was 1.0. From here, both probabilities were checked in terms of whether they included more probabilities. If there were probabilities under its branch, the next lower probabilities were readied for the following assigned binary bit. The process was repeated until there were no probabilities left. The results formed a Huffman tree or binary tree that contained branches and leaves. The leaves represented the probabilities for symbols, while the branches signalled the connection between the leaves. The next process was to read the binary bits in order to create Huffman code. The bits were read from the connectors that linked from the highest,

most likely probability to the branch towards the end of the connectors, which were the leaves.

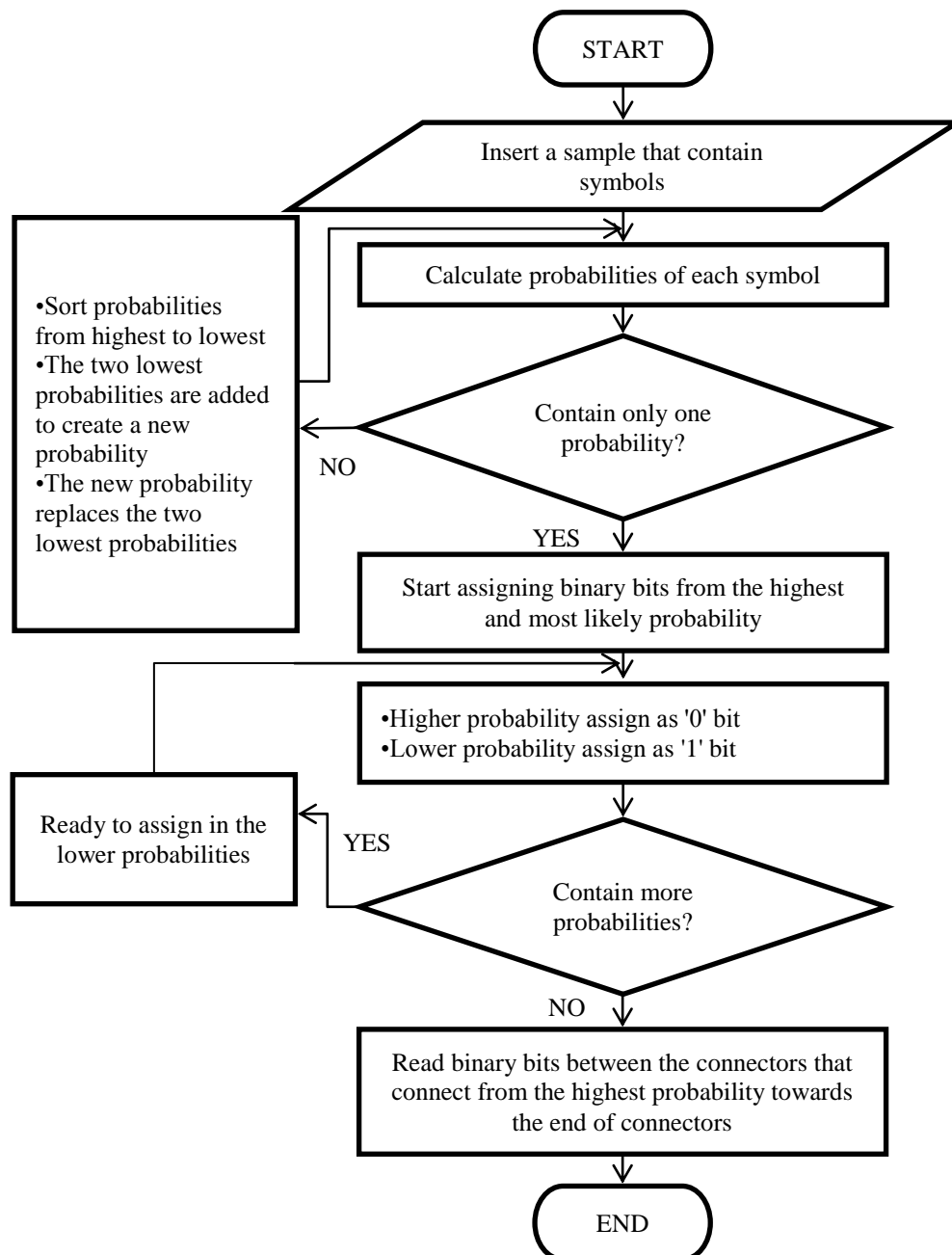


Figure 1. Flow chart for the Huffman encoder.

Figure 2 shows an example of how a Huffman tree is created. There are three symbols, i.e., A, B and C, each with their respective probabilities, which are 0.6, 0.1 and 0.3, sorted from low probability to high probability. The two lowest probabilities are added together to create a new probability, i.e., D. Then, the new probability is sorted with another probability, i.e., A, from low probability to high probability. From there, both probabilities are added to a sum of 1. The dotted boxes are the leaves of the Huffman tree, while the line arrows are the branches of the Huffman tree that connects between leaves.

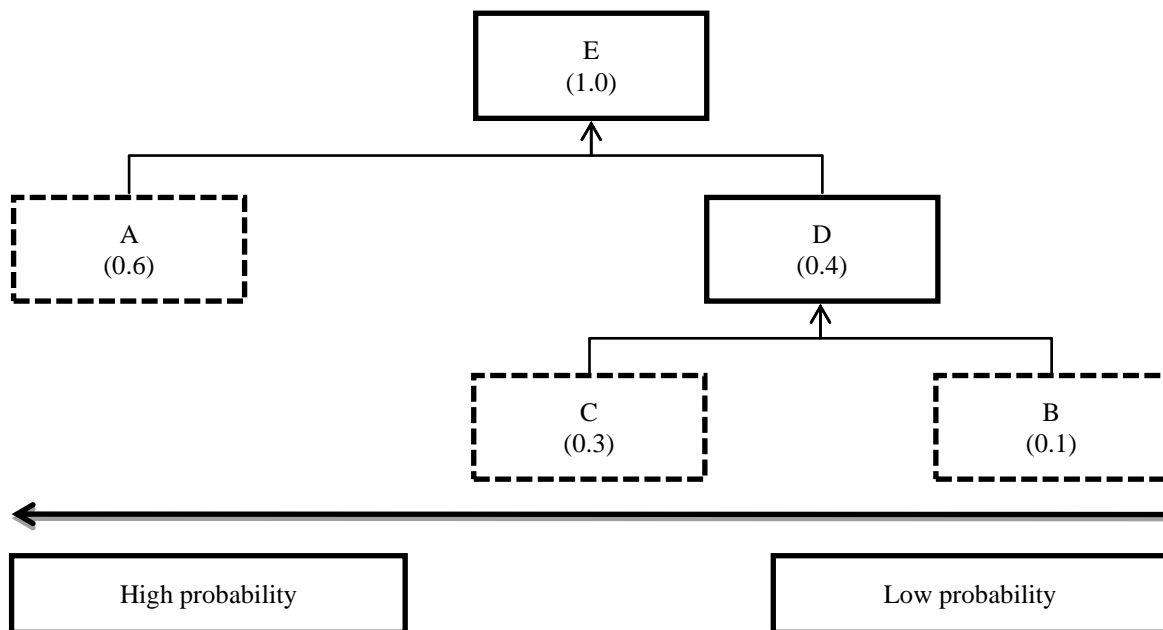


Figure 3 illustrates the flow chart of the Huffman decoder. Firstly, the sequence of binary bits that contain Huffman code was read. Then, the first bit was read. Next, a bit by bit

Figure 2. Example of a Huffman tree.

comparison was made using the existing Huffman code. If there was a match, the symbol of the matched code was outputted. If no match was found for the corresponding Huffman code, the next bit was added until the Huffman code was found. When the corresponding bit and code were matched, the binary bits in the sequence were deleted. The process was repeated until all binary bits in the sequence were deleted.

Based on the analysis using Matlab algorithms, five main computational modules were required for the Huffman module. They were a data retriever (DR), frequency calculator (FC), probability calculator (PC), Huffman tree generator (HTG) and Huffman code generator (HCG). These computational modules were important for Huffman operation. Figure 4 shows a block diagram for the top level of the architecture. The DR module retrieved samples from an input port. The width of the port was 32 bits. The module saved the sample data to memory. Next, the FC module calculated the frequency of the symbols or characters in the sample. The symbols were calculated using the summation and division logic of an arithmetic logic unit (ALU) module inside the main FC module. The calculated values were saved to memory. The PC module's function was to calculate the probability of the symbols of the sample. The probability calculation was performed using the saved frequency values in the memory and the ALU module. The calculated probabilities were saved to memory. After the probabilities were calculated, the next module, which was HTG module, sorted the probabilities as described in the previous section. The sorting of probabilities used a number of registers and memories in order to save its binary codes and its locations within the generated Huffman tree. Finally, the last module, the HCG, generated the Huffman code that had been saved in the previously mentioned memories. A number of registers were used to arrange the binary codes. The arranged binary code, which was the

Huffman code, was later saved to memory. The detail operation of these modules will be described in the following paragraph.

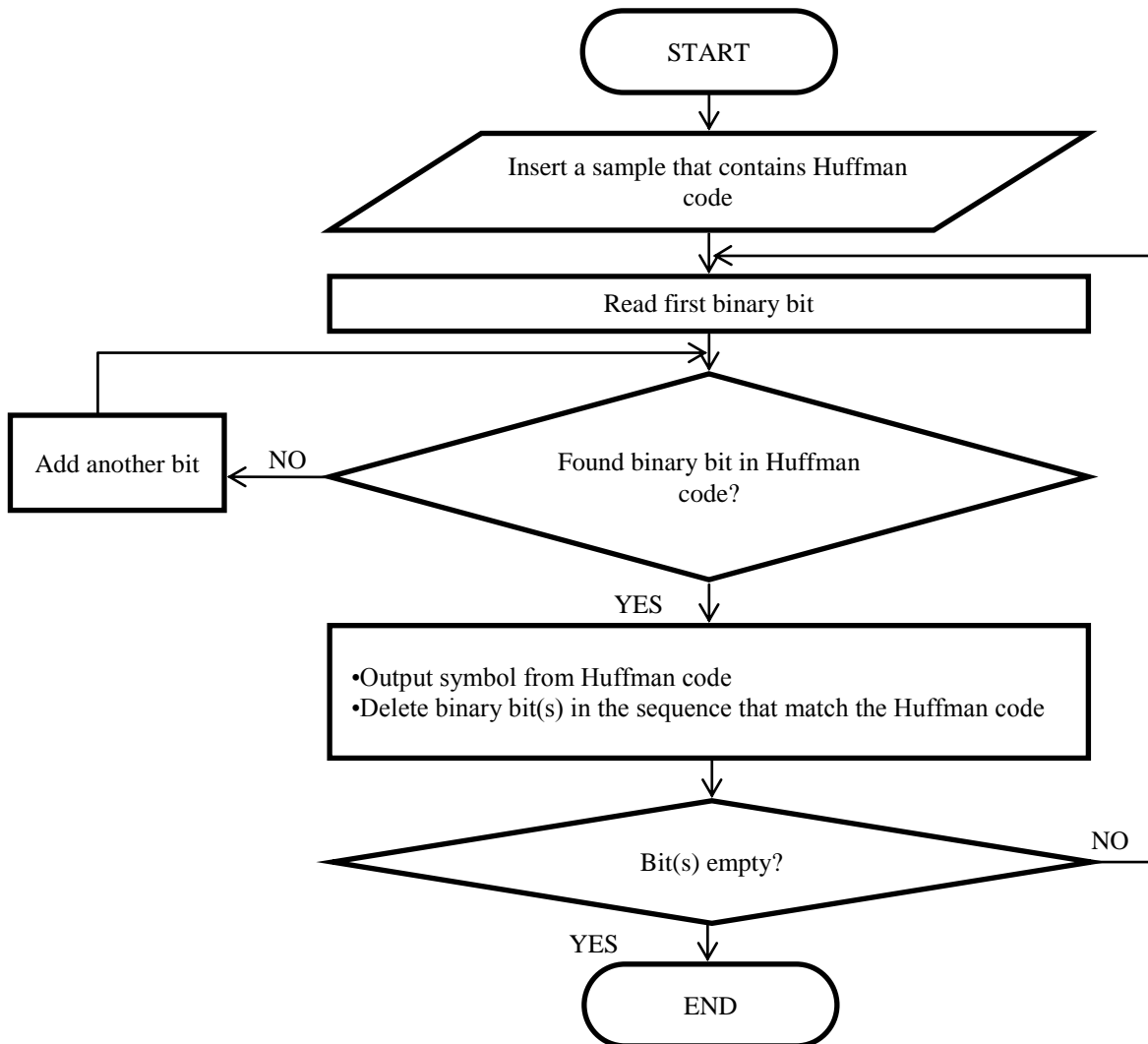
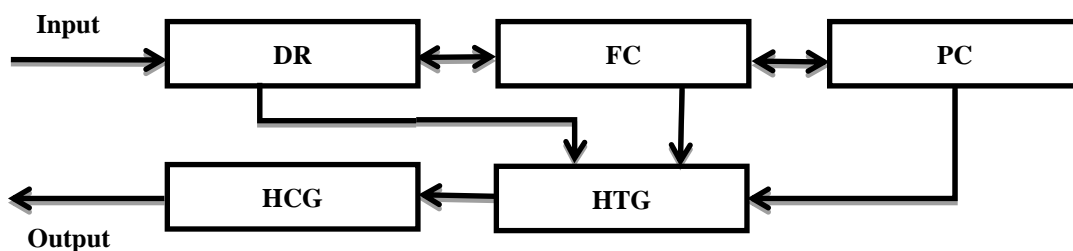


Figure 3. Flow chart for the Huffman decoder.

Figure 5 shows the block diagram for the data retriever module. The function of the module is to save input data into a memory. The module consists of 32-bit 4kbyte data insert memory (DIM), an identifier and an address counter. The data will be inserted using an input port that is 32-bit wide. An identifier will identify the input data, where an incoming input datum will increase the address counter; no incoming data will keep the address counter in a holding state. The address counter is used to set the address of the memory for the datum to be written in its individual address. The output port will transfer the saved data into the next



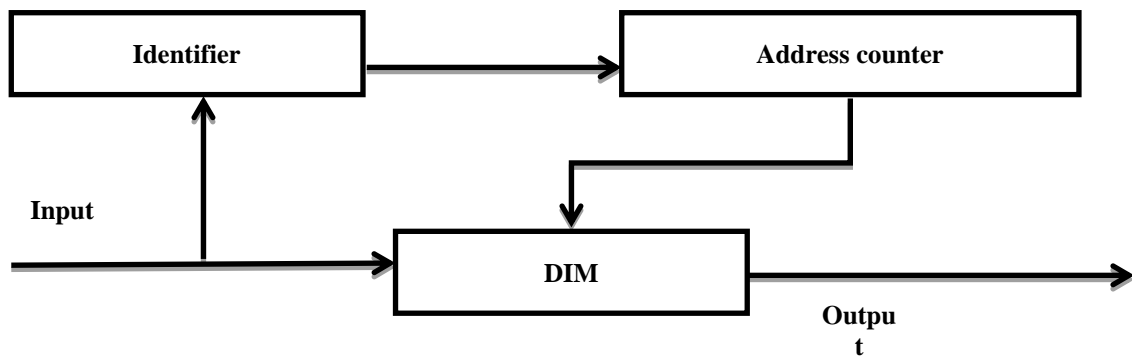


Figure 5. Block diagram for data retriever.

module, which is the FC module.

The block diagram for the frequency calculator module is shown in Figure 6. The function of the module is to calculate the frequency of the saved input data. The module contains 32-bit 2kbyte arranged data memory (ADM), 16-bit 2kbyte frequency arranged data memory (FADM), an identifier, an address counter, data counter and a data comparator. A series of saved data are inserted through the input. The first datum from the series is saved into the data comparator. The purpose of the data comparator is to compare the saved datum with the series of saved data. When there is a match, the identifier will be triggered. The identifier has two functions: to increase the data counter and to increase the address counter when the entire series of data has been read. To save the next datum's frequency, the address counter is increased. Thus, for each datum, the respective frequency will be saved to the same address as in ADM.

The next module in the Huffman algorithm was probability calculator, as shown in Figure 7, which calculated the probabilities of the frequencies of the data. It included an address counter, ALU and probability register. The frequencies from the previous module were sent through the input port towards the ALU. Using the previously mentioned equation, the frequency of a symbol was divided with the total symbols in ADM. The result was saved to the probability register and sent to FADM to be overwritten at its same address location. The address counter in the PC module was increased in order to read the next value. The probabilities in the FADM were sent to the next module as an output.

The main module of the Huffman encoder algorithm is the Huffman tree generator module, as shown in Figure 8, which adds and connects probabilities that will subsequently

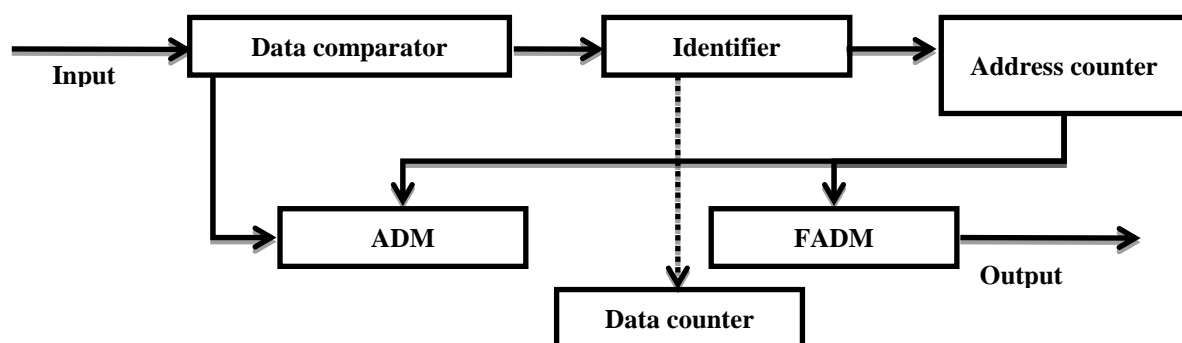


Figure 6. Block diagram for the frequency calculator.

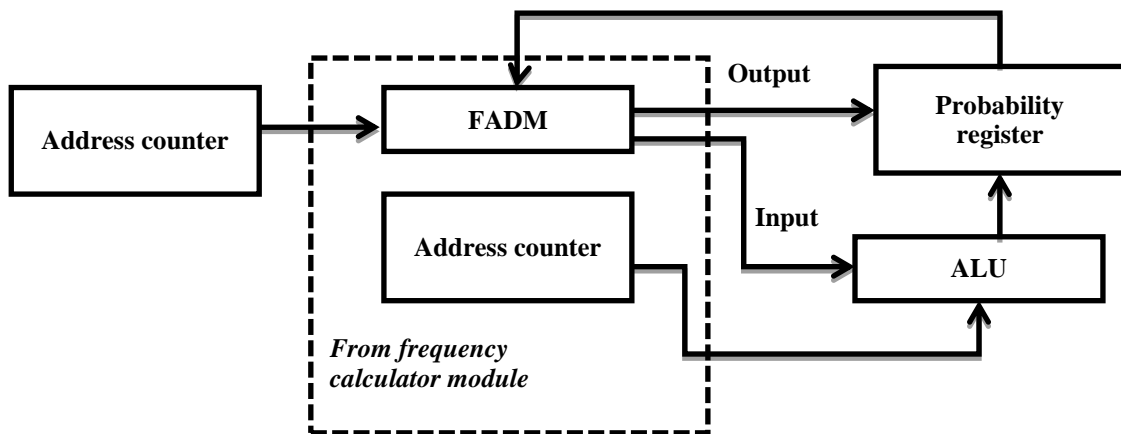


Figure 7. Block diagram for the probability calculator.

become long leaves-branches probabilities called the Huffman tree. In this work, the approach in designing the Huffman tree was to use a memory mapping method. The probabilities from FADM in the FC module are inserted through the probabilities comparator. The address counter acts as a counter for increasing addresses in the memory, while the probabilities comparator has two functions. First, the probability will be written directly into the (arranged probabilities memory) APM if there is no repeated probability; the second function is to trigger the identifier if there is a repeated probability. The identifier also has two functions. The identifier will increase the data counter if there is a repeated probability, count the frequency of the probabilities that appears and writes it in (frequency arranged probabilities memory) FAPM. The second function of the identifier is to trigger the address

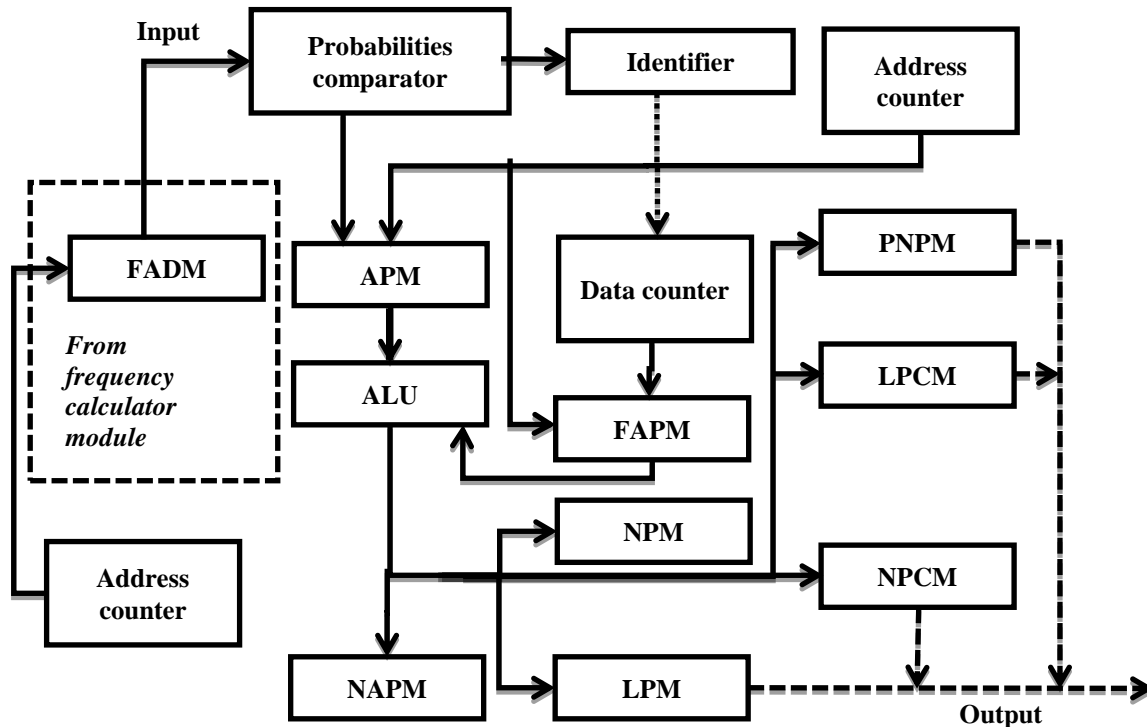


Figure 8. Block diagram of the Huffman tree generator.

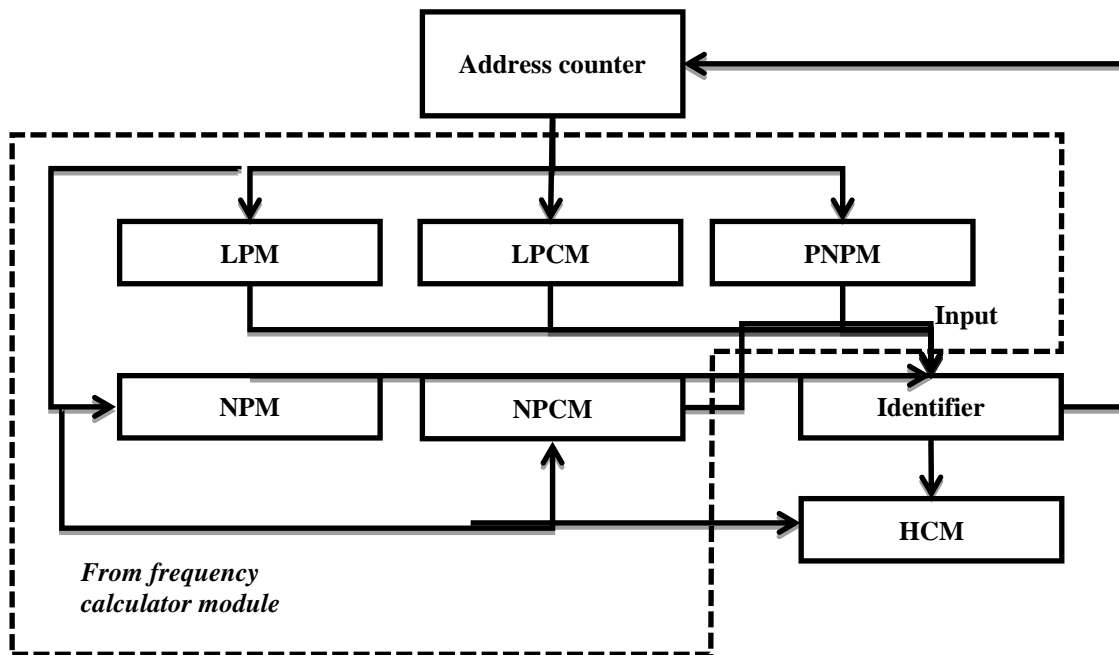


Figure 9. Block diagram of the Huffman code generator.

counter, which sorts from lower to higher probability values. After all probabilities have been sorted, the two lowest probabilities will be added using the ALU module (see Figure 8). The newly added probability will be saved in (new added probabilities memory) NAPM. The binary codes for the Huffman tree will be written in leaves probabilities memory (LPM), leaves probabilities connectors memory (LPCM), nodes probabilities memory (NPM), nodes probabilities connectors memory (NPCM) and parent nodes probabilities memory (PNPM) using the memory mapping technique. The output from LPM, LPCM, NPM, NPCM and PNPM will be sent to the final module, which is the HCG.

The final part for the Huffman encoder architecture is the Huffman code generator, as shown in Figure 9, which generates Huffman code to the respective data using a memory mapping method. It includes an address counter, an identifier and Huffman codes memory (HCM). The size of the memory is 32-bit 2kbyte. The data from LPM, LPCM, NPM, NPCM and PNPM are controlled by the address counter. The address counter acts as an address pointer that points out the data to be read. The identifier acts as a module that follows the memory mapping method. From the identifier, the binary data will be written in the HCM, thus producing the Huffman code.

Results and Discussion

This section discusses the architecture results, which were obtained using SYNOPSIS Design Compiler and IC Compiler. For the architecture results, a sample of 366 temperature data of 32-bit wide was used. A timescale of 20ps and a clock period of 50ns (20MHz) was set. Table 1 shows the results for the clock cycles of the Huffman encoder. The highest time was consumed by the HTG module, with 24472 clock cycles (47.62% of the entire process). This was followed by DR with 32.60%, FC with 18.72%, HCG with 1% and PC with 0.06%.

HTG took the longest due to the search for the two lowest probabilities. The HTG module also consumed a significant amount of time creating the Huffman tree and assigning binary bits.

Table 1. Breakdown percentages in the Huffman algorithm

Module	Clock Cycles	Percentage
Data retriever	16756	32.60%
Frequency calculator	9623	18.72%
Probability calculator	27	0.06%
Huffman tree generator	24472	47.62%
Huffman code generator	516	1.00%
Total	51394	100%

Table 2 shows the power analysis of the Huffman encoder. In the layout level, HTG consumed the most power, that is, 1.184mW, which is 31.75%. The second most power consumed by a module is the others module which is 23.47%, followed by HCG with 13.97%, FC with 13.65%, PC with 10.16% and finally, DR with 7.00%. HTG consumed the most power due to the complexity of the module, which employed most of its power use for compressing data.

Table 2. Power analysis for the Huffman encoder according to part

Module	Layout	
	Total Power	Percentage
Data retriever	0.261mW	7.00%
Frequency calculator	0.509mW	13.65%
Probability calculator	0.379mW	10.16%
Huffman tree generator	1.184mW	31.75%
Huffman code generator	0.521mW	13.97%
Others	0.8750mW	23.47%
Total	3.7290mW	100%

Table 3 shows the area analysis for the Huffman encoder. Again, the HTG module had the largest area at 12568.3327 μm^2 , i.e., 48.1%. The FC module used 17.3%. The third largest module was the HCG, which comprised 15.5%. Next was the PC module at 10.7%, followed by the DR module with 8.0% and the rest with 0.4%. Due to the complexity of the module, HTG employed the largest area in the module.

Table 3. Area analysis for modules in the Huffman encoder

Module	Area (μm^2)	Percentage
Data retriever	2089.8192 μm^2	8.0%
Frequency calculator	4517.3513 μm^2	17.3%
Probability calculator	2808.7829 μm^2	10.7%
Huffman tree generator	12568.3327 μm^2	48.1%
Huffman code generator	4063.1451 μm^2	15.5%
Others	105.9030 μm^2	0.4%

Total	26153.3342 μm^2	100%
-------	----------------------------	------

Table 4 shows the energy analysis for the Huffman encoder. The time taken to completely compress 366 data was 2.5679ms. The clock cycle for the compression process was 51394 clock cycles. It took 2.5697ms to compress all the data. Using the $energy = p \times t$ equation, where p is power and t is time, the energy consumed by the architecture was 9.5824nJ.

Table 4. Results for Huffman algorithm energy analysis

	Huffman encoder
Total Dynamic (mW)	3.7290mW
Clock Cycle	51394
Time (ms)	2.5697ms
Energy (μJ)	9.5824 μJ

Table 5 shows comparisons to previous research conducted by other authors and the present study. It shows that the library cell used for the present study was smaller than others, resulting in a small chip size, which was $208.35 \times 207.94 \mu\text{m}^2$. The power used in this work, i.e., 3.7290mW, was lower compared to other research, where it was 100-times higher. The frequency (20 MHz) used was lower than in other studies. Figure10 shows the layout of the Huffman encoder of the current study, i.e., $208.35 \times 207.94 \mu\text{m}^2$. As seen in Table 5, the comparison between all the works cannot be accepted 100% since each work used different technology process.

Table 5. Comparisons between existing research and the present study

	Process	Core size	Power dissipation	Operating frequency
Park & Prasanna, 1993 [9]	2 μm	$5.8 \times 5.8 \text{ mm}^2$	-	-
Lin & Chang, 2009 [7]	0.18 μm	$1.2 \times 1.2 \text{ mm}^2$	288mW	200 MHz
	0.35 μm	$2.2 \times 2.2 \text{ mm}^2$	394mW	100 MHz
Current study	0.13 μm	$208.35 \times 207.94 \mu\text{m}^2$	3.290mW	20 MHz

Conclusion

This paper discussed the design of Huffman encoder architecture. In this design, the architecture consists of a data retriever, frequency calculator, probability calculator, Huffman tree generator and Huffman code generator. The architecture was simulated and verified using the SYNOPSIS design compiler and SYNOPSIS IC compiler. The experimental results showed that the time taken to compress a sample of 366 data of 32-bit wide was 51394 clock cycles, or 2.5697ms at a clock frequency of 20 MHz. The Huffman chip, with a size of $208.35 \times 207.94 \mu\text{m}^2$, only used 3.7290mW of power, or 9.5824 μJ of energy, to compress this data. The highest power consumption in the architecture was recorded for the

Huffman tree generator, which consumed 31.75% of the total power. This was followed by the Huffman code generator and frequency calculator, which took 13.97% and 13.65%, respectively.

References

- [1] F. Chen, A. P. Chandrakasan, and V. M. Stojanovic, "Design and Analysis of a Hardware-Efficient Compressed Sensing Architecture for Data Compression in Wireless Sensors," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 3, pp. 744-756, Mar. 2012.
- [2] F. Duarte, J. Hulzink, J. Zhou, J. Stuijt, J. Huisken, and H. D. Groot, "A 36 μ w Heartbeat-Detection Processor For A Wireless Sensor Node," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 16, no. 4, pp. 51,1-19, Oct. 2011.
- [3] L. Ruiz-Garcia, L. Lunadei, P. Barreiro, and I. Robla, "A Review of Wireless Sensor Technologies and Applications in Agriculture and Food Industry," *State of the Art and Current Trends. Sensors*, vol. 9, no. 6, pp. 4728-4750, Jun. 2009.
- [4] M. Al-Laham, and I. M. M. El-Emary, "Comparative Study between Various Algorithms of Data Compression Techniques," *IJCSNS International Journal of Computer Science and Network Security*, vol. 7, no. 4, pp. 281-291, Oct. 2007.
- [5] S. R. Kodituwakku, and U. S. Amarasungha, "Comparison of Lossless Data Compression Algorithms for Text Data," *Indian Journal of Computer Science and Engineering*, vol. 1, no. 4, pp. 416-425, Dec. 2010.
- [6] S. Shanmugasundaram, and R. Lourdasamy, "A Comparative Study of Text Compression Algorithms," *International Journal of Wisdom Based Computing*, vol. 1, no. 3, pp. 68-76, Dec. 2011.
- [7] M. B. Lin, and Y. Y. Chang, "A New Architecture of a Two-Stage Lossless Data Compression and Decompression Algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 9, pp. 1297-1303, Sept. 2009.
- [8] M. B. Lin, J. F. Lee, and G. E. Jan, "A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, pp. 925-936, Sept. 2006.
- [9] H. Park, and V. K. Prasanna, "Area Efficient VLSI Architectures for Huffman Coding," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 40, no. 9, pp. 568-575, Sept. 1993.