UNIVERSITY OF THESSALY

ELECTRICAL & COMPUTER ENGINEERING DEPARTMENT

DIPLOMA THESIS

Author: Dimitrios Tychalas

A HARDWARE IMPLEMENTATION OF THE ENTROPY ENCODING TECHNIQUE CABAC ON RECONFIGURABLE ARCHITECTURES

SUPERVISING PROFESSORS :

NIKOLAOS BELLAS, Associate Professor

CHRISTOS SOTIRIOU, Associate Professor

VOLOS 2015

HARDWARE IMPLEMENTATION OF THE ENTROPY ENCODING TECHNIQUE CABAC OR RECONFIGURABLE ARCHITECTURES

ΥΛΟΠΟΙΗΣΗ ΤΗΣ ΤΕΧΝΙΚΗΣ ΚΩΔΙΚΟΠΟΙΗΣΗΣ CABAC ΣΕ ΕΠΑΝΑΔΙΑΤΑΣΣΟΜΕΝΗ ΑΡΧΙΤΕΚΤΟΝΙΚΗ

By Tychalas Dimitrios

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE

In Computer and Communication Engineering

UNIVERSITY OF THESSALY 2015

UNIVERSITY OF THESSALY

Declaration of Authorship

I, Dimitrios Tychalas, hereby certify that this thesis titled, Hardware Implementation of the entropy encoding thechnique CABAC on reconfigurable architectures‘ and the work presented in it has been composed by us and is based on our own work, unless stated otherwise.

The research was carried out wholly or mainly while in candidature for the graduate degree of Diploma of Science in Computer and Communication Engineering at the University of Thessaly, Department of Electrical and Computer Engineering, Greece.

Wherever I have consulted or quoted from the work of others, it is always attributed and the source is given. The main sources of help are referenced in the References section of this thesis.

In memory of Ninos...

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my family, for their continuous and unending support during my academic endeavors. Without their selfless sacrifices and moral encouragement, my graduation would not have been possible.

Next, I want to express my gratitude to my supervising professor Dr. Nikolaos Bellas for his guidance during the last year-and-a-half of my studying in the department, as well as his thought provoking inputs, suggestions and general routing of my developing of this thesis. Without him pushing me towards new goals and always setting the bar high quality-wise, I would not have been able to achieve this level of completeness in my final assignment.

I am also greatly thankful to my assistant supervising professor , Dr. Christos Sotiriou, whose knowledge and experience on the CABAC algorithm, coupled with the ability to relay said knowledge, set up the foundations for this endeavor. Special thanks also to fellow computer engineering student Nikos Sketopoulos who sacrificed some of his limited time to help me understand the basics of arithmetic coding.

Finally to all the friends and colleagues I have made during my time here at the ECE department in Volos, thank you for some magnificent years and many fond memories.

TABLE OF CONTENTS

Abstract

Video compression is an essential technique, especially in these days of continuous demand for higher resolution and faster framerate video. Several techniques for video compression exist nowdays, but the H.264 standard is among the most recognizable and frequently used. The last stage of said standard, is the implementation of a binary arithmetic coding algorithm, which utilizes arithmetic coding theory principals to achieve efficient encoding of the produced information. The standard of course, exist in software. A hardware implementation though, offers much higher efficiency at the cost of additional transistor count, which can be minute depending on the complexity of the algorithm. In this project, the main point of focus is implementing the algorithm CABAC as a hardware module using Verilog HDL in conjunction with the Xilinx Vivado Suite, simulated to validate its correctness and synthesized to be programmed in a ZedBoard FPGA. A next step includes the use of high level synthesis, to automatically produce a hardware module for an existing C source file and implement enhancements and modification. As a final step, the created module will be appropriately interfaced and connected as a hardware accelerator to a Zynq processor, to create a unique SoC with binary encoding capabilities.

PART A : Theoretical Background

1. Introduction to Compression

In computer science and information technology, information coding is being deployed as a means of utilizing less bits than the original representation. Compression can be lossy or lossless. Lossless compression shortens the bit count without information loss. On the other hand, lossy compression lessens the bit count by finding and removing needless information. The function of shortening the size of a data file is called data compression.

Compression is highly useful, due to the fact that it helps reducing needs in resources like data storage or transmission bandwidth._Compressed data though, need to be decompressed in order to be used. That extra process requires even more resources and costs. Data compression systems design mandates compromises between several factors, such as the degree of compression, the level of data distortion (in case of l lossy compression) and the necessary resources for compression or decompression.

Compression algorithms usually utilize a form of static reduction to losslessly shorten data, to enable a reversing process. That kind of compression is viable because most real data show static redundancies. For instance, instead of depicting an image as a series of “red pixel”, “red pixel” etc. , data can be coded as “256 red pixels”.

There are many programs with the function of reducing a file size by reducing redundancy. Lempel-Ziv compression method, is one of the most popular algorithms for lossless compression. DEFLATE is another version of_LZ, improved in matters of decompression speed and compression analogy, but compression is of itself slow. DEFLATE is being deployed in PKZIP, Gzip, and PNG. LZW (Lempel-Ziv-Welch) is being used on GIF images. Worth mentioning is also the LZR (Lempel-Ziv-Welch) algorithm, which is the core for zip compression.

1.2 Compression and video

Video compression utilizes modern coding techniques to reduce redundancy in video data. The majority of video compression algorithms use lossy compression, but there is always a tradeoff between image quality and resources used. A heavily compressed video can appear quite distorted.

Some video compression schemas, work on square teams of neighboring pixels , usually called macroblocks. These teams are being compared between frames and the coder sends only any differences that occurred. In areas with much movement in video, coding requires larger numbers of changing pixels, like during an explosion scene or panoramic views. To reduce such changes, human perception is also being considered. Subtle differences in coloration or lighting may go unnoticed by the naked eye, so techniques like JPEG tend to normalize colors to a degree to further diminish the coding procedure.

1.3 CABAC and H.264

Context Adaptive Binary Arithmetic Coding algorithm (CABAC) has been developed in the context of ITU-T and ISO / IEC joint efforts for the development and specification of the H.264 / AVC standard. CABAC has been utilized as a basic part of H.264 / AVC as well as the High Efficiency Video Coding (HVEC) standard. There is one other method for entropy coding deployed by H.264, named Context Adaptive Variable Length Coding (CAVLC), which can achieve lower complexity. In HEVC though, CABAC is the sole entropy coding technique. CABAC design consists of a number of basic blocks as depicted in Image 1.1

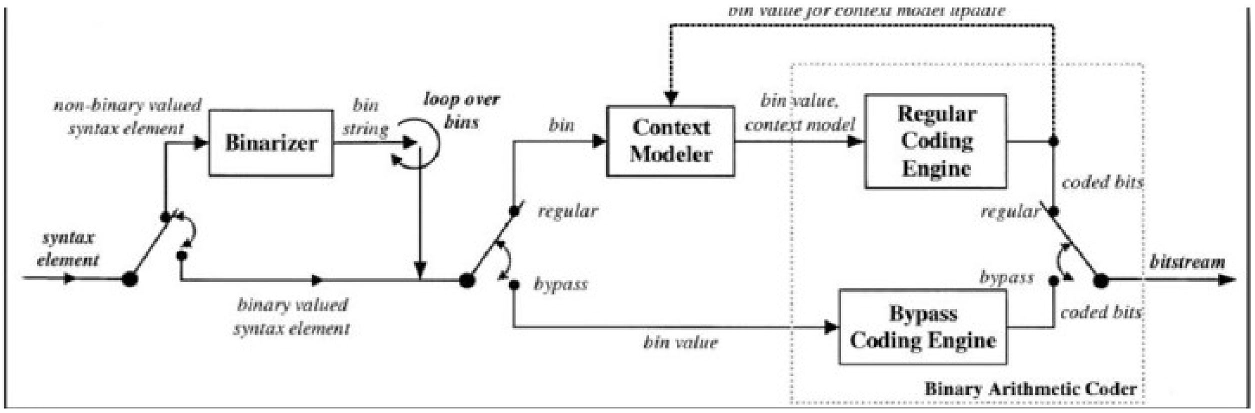


Image 1.1 : CABAC structure

1.3.1 Binarization

In general, a binarization system defines a unique representation of element values, using binary sequences, also called bins. Binarization design is based on some elemental standards, whose

structure allows simple computation and which are tailored to specific appropriate probability models.

1.3.2 Context Modeling

Following the decomposition of each element to a sequence consisting of bins, further processing depends on a Coding Mode option, that can be regular, or situation bypass. The latter is chosen for bins that are connected to the information signal or for low priority bins, that are supposed to be evenly distributed and for which the classic binary arithmetic coding process is bypassed. In the standard coding process, every bin value is being coded using standard binary coding, where the associated probability model is either defined by a static choice, without Context Modeling, or is chosen adaptively depending on the model associated with the context. As an important choice design-wise, the latter case is used on the most frequent bins, while the former on the least frequent.

1.3.3 Binary Arithmetic Coding

In the lowest level of CABAC processing, each bin enters the binary arithmetic coder, either with the classic or the bypass coding function that was described earlier. For the latter, a faster, less complex portion of the coding function is deployed, while for the former, the coding of a given bin depends on the real state of the associated context probability model.

Probability estimation on CABAC is based on approximation table that uses a finite state machine (FSM), with indexed states as shown in image 1.2.

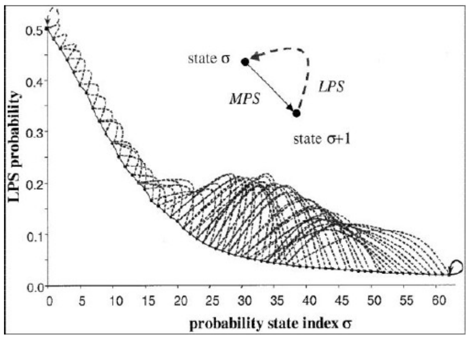


Image 1.2 : Entropy Diagram

Every probability model on CABAC can take one of 127 different states associated with relative probability p values between an interval [0.01875, 0.98125]. Let it be noted that even with the distinction between least probable symbols (LPS) and most probable ones (MPS), just the description of the LPS state can suffice for the representation of both. The method for the design of the transition states was

developed by Howard and Vitter, with the following transition rules regarding t to t+1.

$$p_{LPS}^{t+1} = \begin{Bmatrix} a \times p_{LPS}^{t} & MPS\ occur \\ a \times p_{LPS}^{t} + (1-a) & LPS\ occur \end{Bmatrix}$$

Image 1.3 : LPS/MPS Transition rule

2. Arithmetic Coding

2.1 The Algorihm

In this section, there will be a presentation and analysis of the fundamental arithmetic coding algorithm developed by *Witten, Neal and Cleary*, which became the basis for the Binary Arithmetic Coding technique deployed on CABAC.

In arithmetic coding, a message is represented by an interval of real numbers between 0 and 1. As the message becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify that interval grows. Successive symbols of the message reduce the size of the interval in accordance with the symbol probabilities generated by the model. The more likely symbols reduce the range by less than the unlikely symbols and hence add fewer bits to the message.

Before anything is transmitted, the range for the message is the entire interval [0, l), denoting the half-open interval $0 \leq x < 1$. As each symbol is processed, the range is narrowed to that portion of it allocated to the symbol. For example, suppose the alphabet is (a, e, i, O,u, !), and a fixed model is used with probabilities shown in Table 1.

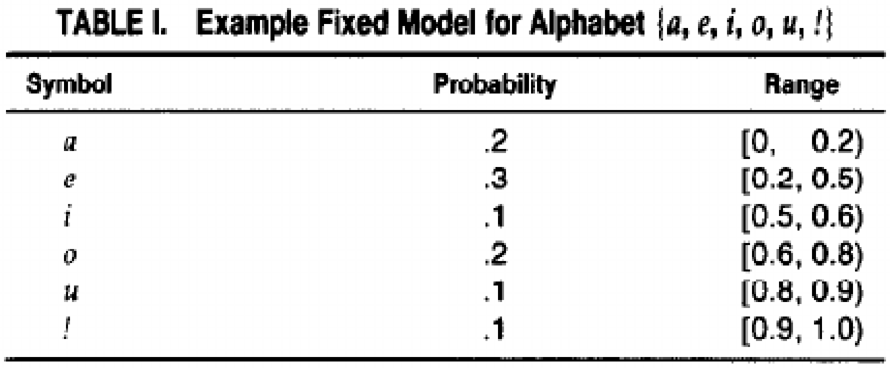


TABLE I. Example Fixed Model for Alphabet {*a*, *e*, *i*, *o*, *u*, *!*}

Symbol	Probability	Range
a	.2	[0, 0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1.0)

Table 1 : Probability distribution model

Imagine transmitting the message “eaii!”. Initially, both encoder and decoder know that the range is [0, 1). After seeing the first symbol, e, the encoder narrows it to [0.2, 04], the range the model allocates to this symbol. The second symbol, a, will narrow this new range to the first one-fifth of it, since a has been allocated [0, 0.2). This produces [O.2, 0.26), since the previous range was 0.3 units long and one-fifth of that is 0.06. The next symbol, i, is allocated [0.5, 0.6), which when applied to [0.2, 0.26) gives the smaller range [0.23, 0.236). Proceeding in this way, the encoded message builds up as follows:

Initially		[0, 1)
After seeing	e	[0.2, 0.5)
	a	[0.2, 0.26)
	i	[0.23, 0.236)
	i	[0.233, 0.2336)
	!	[0.23354, 0.2336)

Image 2.1 shows another representation of the encoding process. The vertical bars with ticks represent the symbol probabilities stipulated by the model. After the first symbol has been processed, the model is scaled into the range [0.2, 0.5), as shown in Image 2.1. The second symbol scales it again into the range [0.2, 0.26). But the picture cannot be continued in this way without a magnifying glass! Consequently, Image 2.2 shows the ranges expanded to full height at every stage and marked with a scale that gives the endpoints as numbers.

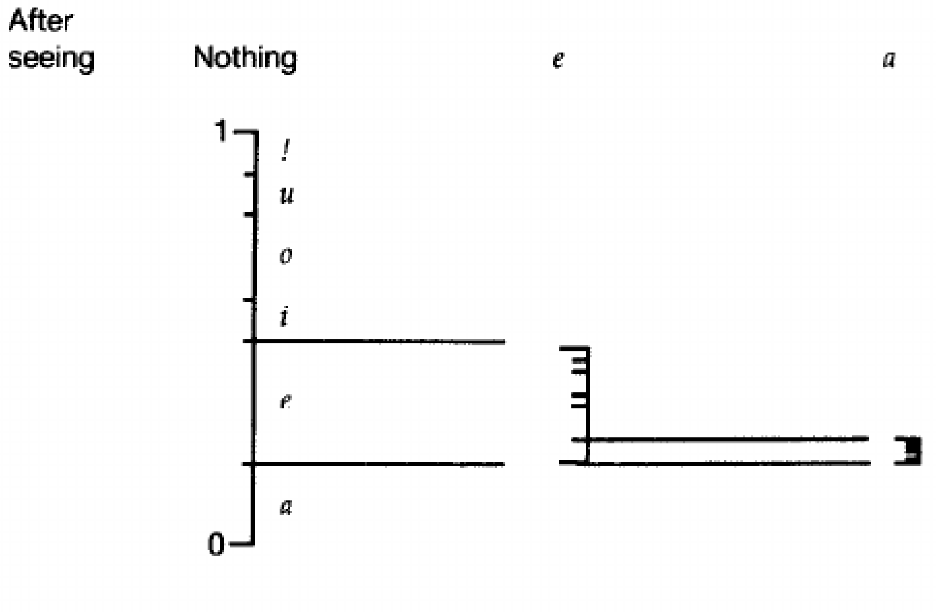


Image 2.1 : Representation of the arithmetic coding process

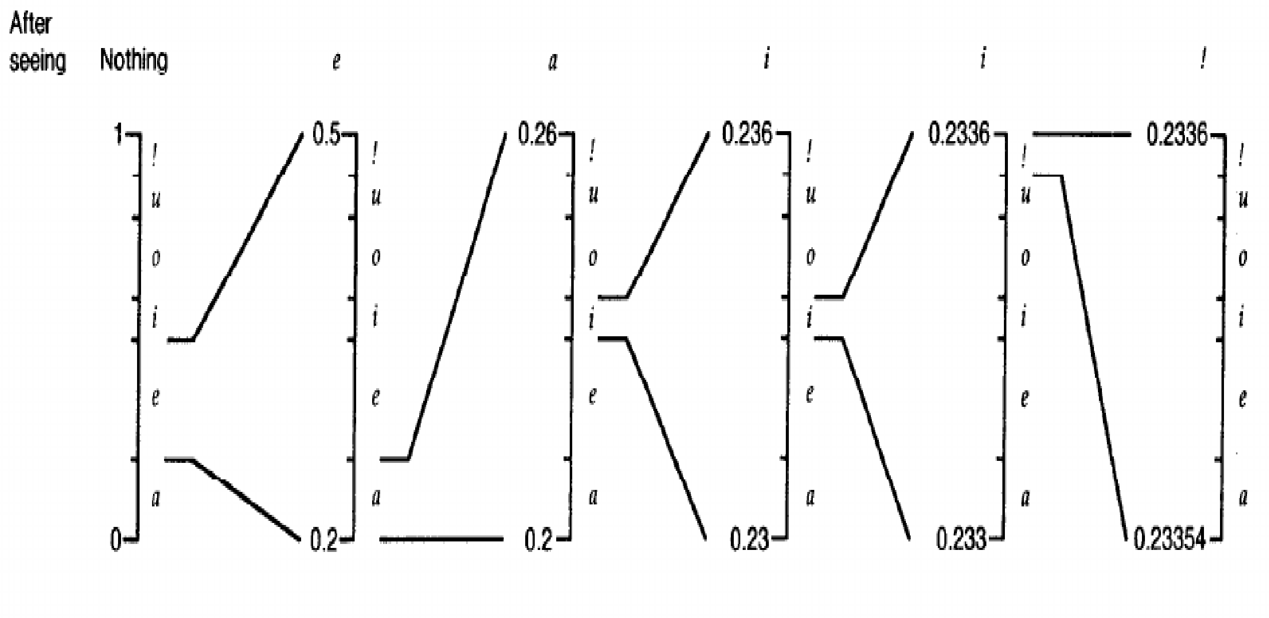


Image 2.2 : Representation of the Arithmetic Coding Process with the Interval Scaled Up at Each Stage

Suppose all the decoder knows about the message is the final *range*, [0.23354, 0.2336). It can immediately deduce that the first character was e. Now it can simulate the operation of the encoder:

Initially	[0,1)
After seeing e	[0.2, 0.5)

This makes it clear that the second character is a, since this will produce the *range*

After seeing a	[0.2, 0.26),

which entirely encloses the given *range* [0.23354, 0.2336). Proceeding like this, the decoder can identify the whole message. It is not really necessary for the decoder to know both ends of the range produced by the encoder. Instead, a single number within the range--for example, 0.23355-will suffice. (Other numbers, like 0.23354, 0.23357, or even 0.23354321, would do just as well.) However, the decoder will face the problem of detecting the end of the message, to determine when to stop decoding. After all, the single number 0.0 could represent any of a, aa, aaa, aaaa, ... To resolve the ambiguity, we ensure that each message ends with a special EOF symbol known to both encoder and decoder. Symbol ! will be used to terminate messages, and only to terminate messages. When the decoder sees this symbol, it stops decoding.

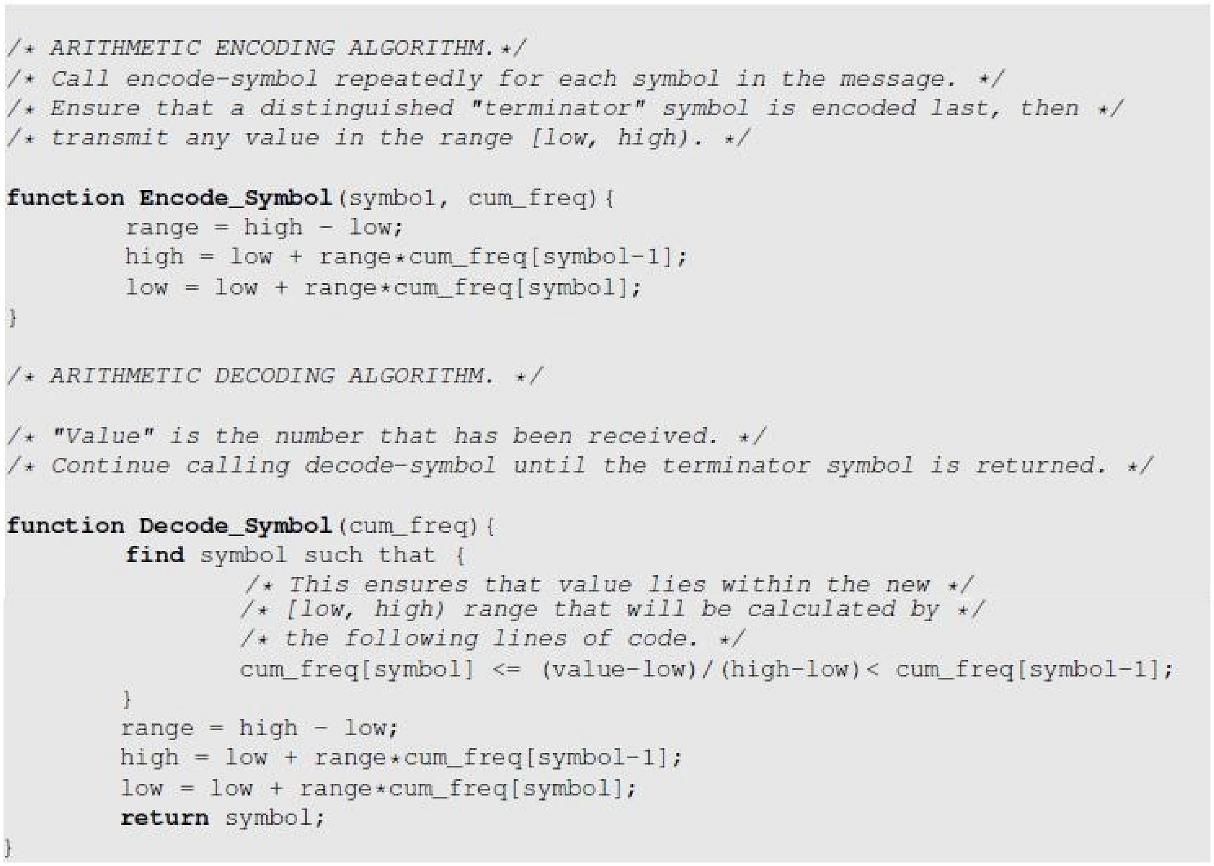
The entropy of the five-symbol message eaii! Is :

-log 0.3 - log 0.2 - log 0.1 - log 0.1 - log 0.1

= -log 0.00006 = 4.22

(using base 10, since the above encoding was performed in decimal). This explains why it takes five decimal digits to encode the message. In fact, the size of the final range is 0.2336 - 0.23354 = 0.00006, and the entropy is the negative logarithm of this figure. Of course, we normally work in binary, transmitting binary digits and measuring entropy in bits. Five decimal digits seems a lot to encode a message comprising four vowels! It is perhaps unfortunate that our example ended up by expanding rather than compressing. Needless to say, however, different models will give different entropies. The best single-character model of the message eaii! Is the set of symbol frequencies *{e(O.2), a(0.2), i(O.4), !(0.2)}*, which gives an entropy of 2.89 decimal digits. Using this model the encoding would be only three digits long. Moreover, as noted earlier, more sophisticated models give much better performance in general.

Image 3.3 :



```
/* ARITHMETIC ENCODING ALGORITHM.*/
/* Call encode-symbol repeatedly for each symbol in the message. */
/* Ensure that a distinguished "terminator" symbol is encoded last, then */
/* transmit any value in the range [low, high). */

function Encode_Symbol(symbol, cum_freq){
        range = high - low;
        high = low + range*cum_freq[symbol-1];
        low = low + range*cum_freq[symbol];
}

/* ARITHMETIC DECODING ALGORITHM. */

/* "Value" is the number that has been received. */
/* Continue calling decode-symbol until the terminator symbol is returned. */

function Decode_Symbol(cum_freq){
        find symbol such that {
                /* This ensures that value lies within the new */
                /* [low, high) range that will be calculated by */
                /* the following lines of code. */
                cum_freq[symbol] <= (value-low)/(high-low)< cum_freq[symbol-1];
        }
        range = high - low;
        high = low + range*cum_freq[symbol-1];
        low = low + range*cum_freq[symbol];
        return symbol;
}
```

Pseudocode for the Encoding and Decoding Process.

Image 3.3 shows a pseudocode fragment that summarizes the encoding and decoding procedures developed in the last section. Symbols are numbered, 1, 2, 3 . . . The frequency range for the ith symbol is from *cum-freq[i]* to *cum-freq[i - 1]*. As i decreases, *cum-freq[i]* increases, and *cum-freq[0]* = 1. (The reason for this "backwards" convention is that *cum-freq[0]* will later contain a normalizing factor, and it will be convenient to have it begin the array. The "current interval" is *[low, high)*, and for both encoding and decoding, this should be initialized to *[0, 1)*. Unfortunately, the code on Image 3.3 is overly simplistic. In practice, there are several factors that complicate both encoding and decoding:

Incremental transmission and reception. The encode algorithm as described does not transmit anything until the entire message has been encoded; neither does the decode algorithm begin decoding until it has received the complete transmission. In most applications an incremental mode of operation is necessary.

The desire to use integer arithmetic. The precision required to represent the *[low, high)* interval grows with the length of the message. Incremental operation will help overcome this, but the potential for overflow and underflow must still be examined carefully.

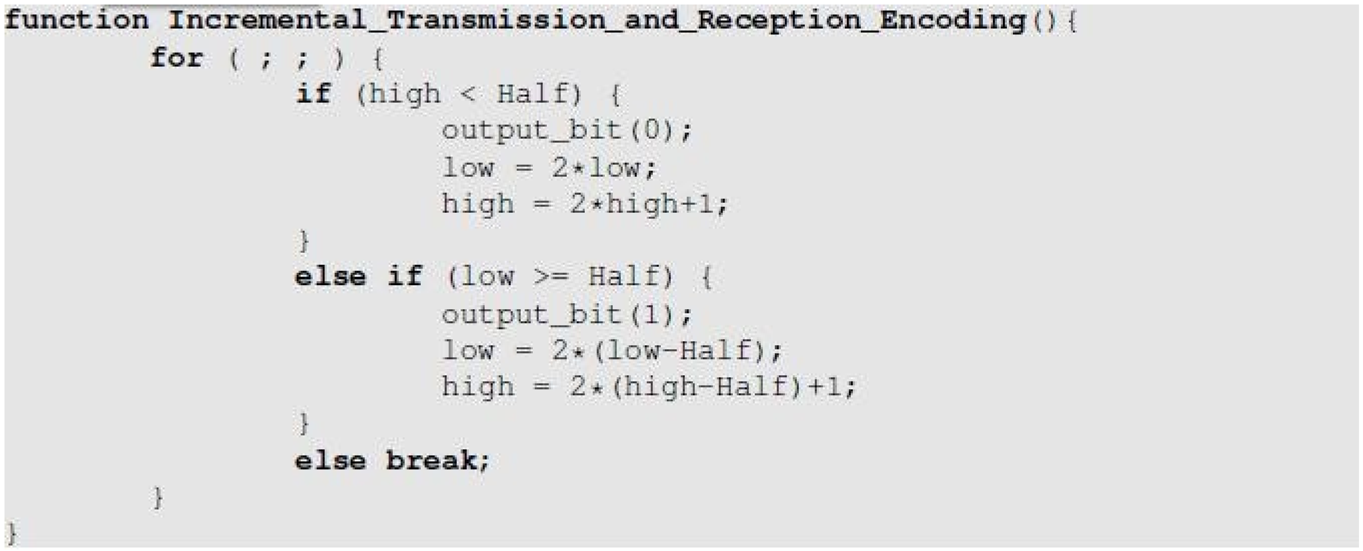
Representing the model so that it can be consulted efficiently. The representation used for the model should minimize the time required for the decode algorithm to identify the next symbol. Moreover, an adaptive model should be organized to minimize the time-consuming task of maintaining cumulative frequencies.

2.2 Representing the Model

Implementations of models are discussed in the next section; here we are concerned only with the interface to the model . In C, a byte is represented as an integer between 0 and 255 (a char). Internally, we represent a byte as an integer between 1 and 257 inclusive (an index), EOF being treated as a 257th symbol. It is advantageous to sort the model into frequency order, so as to minimize the number of executions of the decoding loop . To permit such reordering, the char/index translation is implemented as a pair of tables, *index-to-char[ ]* and *char-to-index[ ]*. In one of our models, these tables simply form the index by adding 1 to the char, but another implements a more complex translation that assigns small indexes to frequently used symbols. The probabilities in the model are represented as integer frequency counts, and cumulative counts are stored in the array *cum_freq[ ]*. As previously, this array is "backwards," and the total frequency count, which is used to normalize all frequencies, appears in *cum_frec [0]*. Cumulative counts must not exceed a predetermined maximum, *Max- frequency*, and the model implementation must prevent overflow by scaling appropriately. It must also ensure that neighboring values in the *cum_freq [ ]* array differ by at least 1. Otherwise the affected symbol cannot be transmitted.

2.3 Incremental Transmission and Reception

A special data type, *code-value*, is defined for these quantities, together with some useful constants: *Top-value*, representing the largest possible *code-value*, and *First-qtr, Half, and Third-qtr,* representing parts of the range . Whereas in previously the current interval is represented by *[low, high)*, now it is *[low, high]*; that is, the range now includes the value of high. Actually, it is more accurate (though more confusing) to say that the interval represented is [low, high + 0.11111...). This is because when the bounds are scaled up to increase the precision, zeros are shifted into the low order bits of low, but ones are shifted into high. Although it is possible to write the program to use a different convention, this one has some advantages in simplifying the code. As the code range narrows, the top bits of low and high become the same. Any bits that are the same can be transmitted immediately, since they cannot be affected by future narrowing. For encoding, since we know that *low* ≤ *high* this requires code like Image 3.4



```
function Incremental_Transmission_and_Reception_Encoding(){
        for ( ; ; ) {
                if (high < Half) {
                        output_bit(0);
                        low = 2*low;
                        high = 2*high+1;
                }
                else if (low >= Half) {
                        output_bit(1);
                        low = 2*(low-Half);
                        high = 2*(high-Half)+1;
                }
                else break;
        }
}
```

Image 3.4 : Incremental Transition Encoding

which ensures that, upon completion, *low* < *Half* ≤ *high*. Care is taken to shift ones in at the bottom when high is scaled, as noted above. Incremental reception is done using a number called value in which processed bits flow out the top (high-significance) end and newly received ones flow in the bottom. Initially, start-decoding fills value with received bits.

```
function Incremental_Transmission_and_Reception_Decoding(){
        for ( ; ; ) {
                if (high < Half) (
                        value = 2*value+input_bit();
                        low = 2*low;
                        high = 2*high+1;
                }
                else if (low > Half) {
                        value = 2*(value-Half)+input_bit();
                        low = 2*(low-Half);
                        high = 2*(high-Half)+1;
                }
                else break;
        }
}
```

2.4 Underflow

As Image 3.1 shows, arithmetic coding works by scaling the cumulative probabilities given by the model into the interval *[low, high]* for each character transmitted. Suppose low and high are very close together-so close that this scaling operation maps some different symbols of the model onto the same integer in the *[low, high]* interval. This would be disastrous, because if such a symbol actually occurred it would not be possible to continue encoding. Consequently, the encoder must guarantee that the interval *[low, high]* is always large enough to prevent this. The simplest way to do this is to ensure that this interval is at least as large as *Max_frequency*, the maximum allowed cumulative frequency count . How could this condition be violated? The bitshifting operation explained above ensures that low and high can only become close together when they straddle *Half*. Suppose in fact they become as close as

$$First\_qtr \leq low < Half \leq high < Third\_qtr.$$

Then the next two bits sent will have opposite polarity, either 01 or 10. For example, if the next bit turns out to be zero (i.e., high descends below *Half* and [0, *Half*] is expanded to the full interval), the bit after that will be one, since the range has to be above the midpoint of the expanded interval. Conversely, if the next bit happens to be one, the one after that will be zero. Therefore the interval can safely be expanded right now, its opposite must be transmitted afterwards as well. Thus *[First-qtr, Third-qtr]* is expanded into the whole interval, remembering in *bits-to- follow* that the bit that is output next must be followed by an opposite bit. But what if, after this operation, it is still true that

$$First\_qtr \le low < Half \le high < Third\_qtr.$$

Image 3.5 illustrates this
situation, where the current *[low, high]* range (shown as a thick line) has been expanded a total of three times. Suppose the next bit turns out to be zero, as indicated by the arrow in Imagee 3.5a being below the halfway point. Then the next three bits will be ones, since the arrow is not only in the top half of the bottom half of the original range, but in the top quarter, and moreover the top eighth, of that half-this is why the expansion can occur three times. Similarly, as Image 3.5b shows, if the next bit turns out to be a one, it will be followed by three zeros.

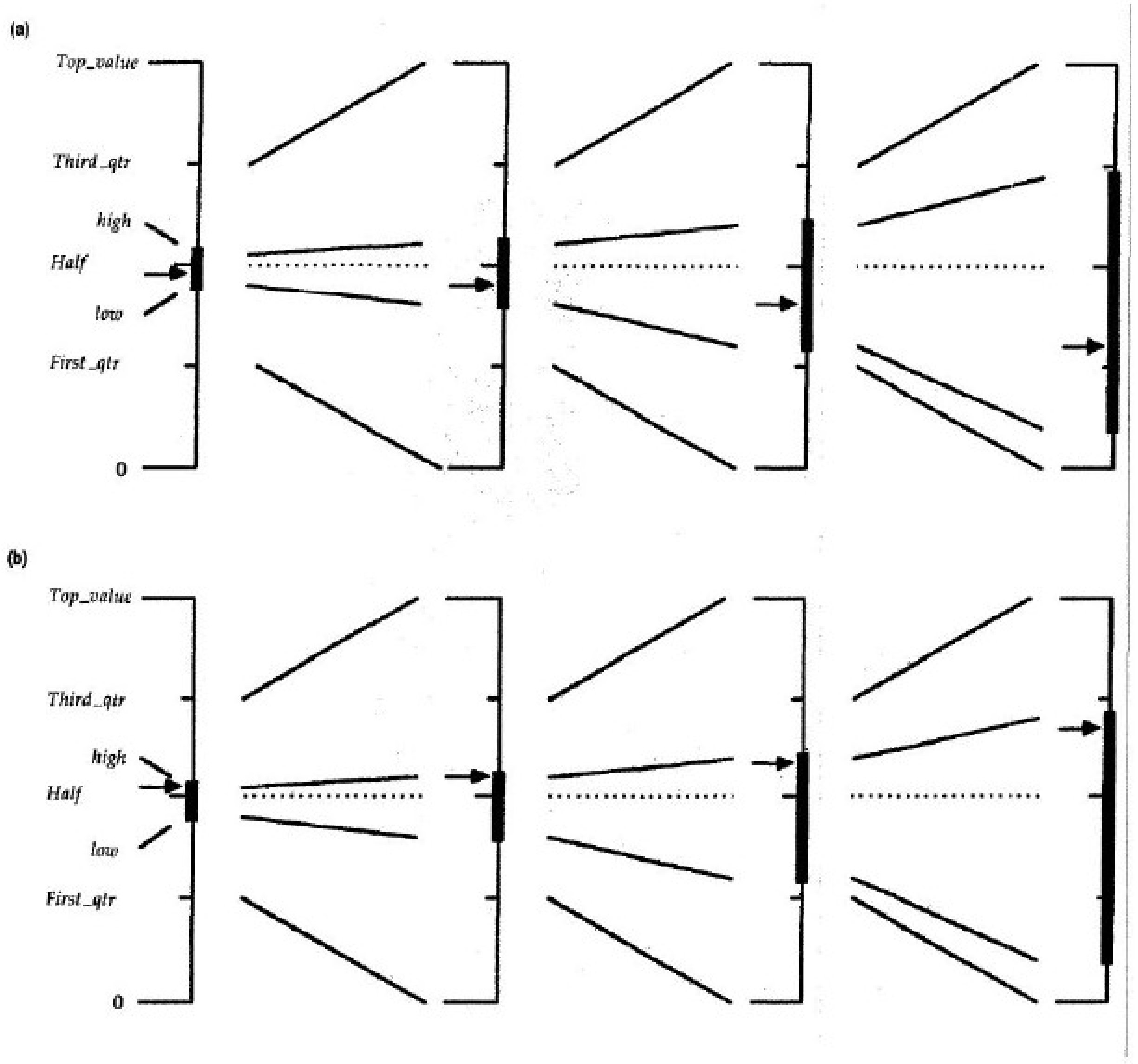


Image 3.5 : Scaling the interval to prevent underflow

Consequently, we need only count the number of expansions and follow the next bit by that number of opposites . Using this technique the encoder can guarantee that, after the shifting operations, either

$$low < First\_qtr < Half \leq high$$

or

$$low < Half < Third\_qtr \leq high$$

Therefore, as long as the
integer range spanned by the cumulative frequencies fits into a quarter of that provided by code-values, the underflow problem cannot occur. This corresponds to the condition

$$Max\_frequency \leq \frac{Top\_value + 1}{4} + 1$$

which is satisfied , since *Max_frequency* = 2^{14} - 1 and *Top_value* = 2^{16} - 1 More than 14 bits cannot be used to represent cumulative frequency counts without increasing the number of bits allocated to *code_values*. We have discussed underflow in the encoder only. Since the decoder's job, once each symbol has been decoded, is to track the operation of the encoder, underflow will be avoided if it performs the same expansion operation under the same conditions.

2.5 Overflow

Overflow cannot occur provided the product *range x Max_frequency* fits within the integer word length available, since cumulative frequencies cannot exceed *Max_frequency.* Range might be as large as *Top_value* + 1. so the largest possible product is $2^{16}(2^{14} - 1)$. which is less than 2^{30}. Long declarations are used for *code_value* and range to ensure that arithmetic is done to 32-bit precision.

2.6 Termination

To finish the transmission, it is necessary to send a unique terminating symbol (EOF-symbol,) and then follow it by enough bits to ensure that the encoded string falls within the final range. Since *done_encoding* can be sure that low and high are constrained by either the equations above, it need only transmit 01 in the first case or 10 in the second to remove the remaining ambiguity. It is convenient to do this using the *bit_plus_follow()* procedure discussed earlier. The *input_bit ()* procedure will actually read a few more bits than were sent by *output-bit()*, as it needs to keep the low end of the buffer full. It does not matter what value these bits have, since EOF is uniquely determined by the last two bits actually transmitted.

2.7 Models for Arithmetic Coding

The algorithm be used with a model that provides a pair of translation tables *index_to_char[ ]* and *char_to_index[ ]*, and a cumulative frequency array *cum_freq [ ]*, The requirements on the latter are that

- $cum\_freq[i-1] \geq cum\_freq[i]$
- $cum\_freq[0] \leq Max\_frequency.$

Provided these conditions are satisfied, the values in the array need bear no relationship to the actual cumulative symbol frequencies in messages. Encoding and decoding will still work correctly, although encodings will occupy less space if the frequencies are accurate.

2.7.1 Fixed Models

The simplest kind of model is one in which symbol frequencies are fixed. However, bytes that did not occur in that sample have been given frequency counts of one in case they do occur in messages to be encoded (so this model will still work for binary files in which all 256 bytes occur). Frequencies have been normalized to total 8000. The initialization procedure *start_model ()* simply computes a cumulative version of these frequencies , having first initialized the translation tables. Execution speed would be improved if these tables were used to reorder symbols and frequencies so that the most frequent came first in the *cum_freq [ ]* array.

2.7.2 Adaptive Models

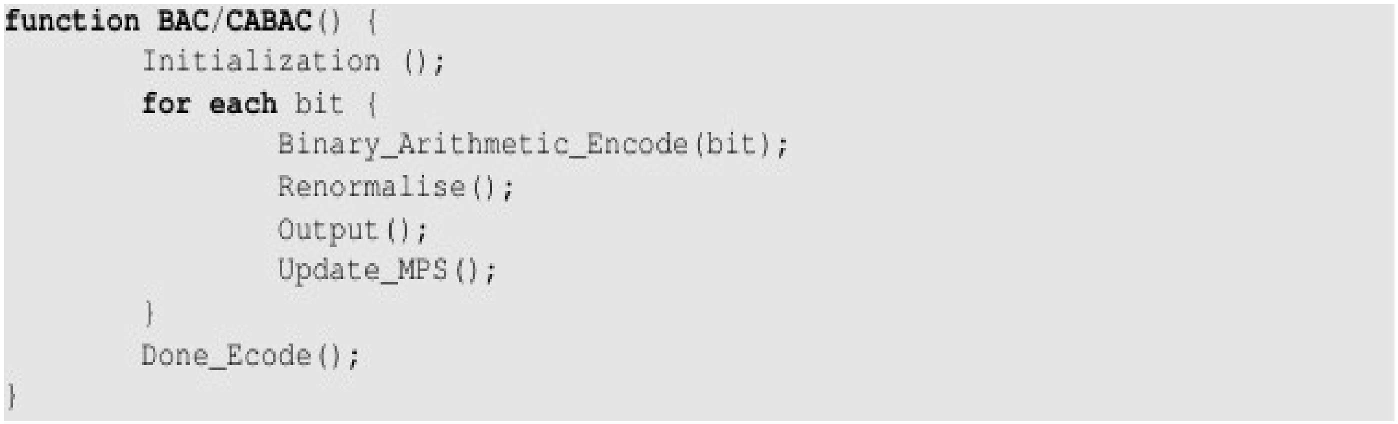
An adaptive model represents the changing symbol frequencies seen so fur in a message. Initially all counts might be the same (reflecting no initial information), but they are updated, as each symbol is seen, to approximate the observed frequencies. Provided both encoder and decoder use the same initial values (e.g., equal counts) and the same updating algorithm, their models will remain in step. The encoder receives the next symbol, encodes it, and updates its model. The decoder identifies it according to its current model and then updates its model. Initialization is the same as for the fixed model, except that all frequencies are set to one. The procedure *update_model (symbol)* is called by both *encode_symbol ()* and *decode_symbol ()* after each symbol is processed.

Updating the model is quite expensive because of the need to maintain cumulative totals. Frequency counts, which must be maintained anyway, are used to optimize access by keeping the array in frequency order-an effective kind of self-organizing linear search. *Update_model ()* first checks to see if the new model will exceed the cumulative-frequency limit, and if so scales all frequencies down by a factor of two (taking care to ensure that no count scales to zero) and recomputes cumulative values. Then, if necessary, *update_model ()* reorders the symbols to place the current one in its correct rank in the frequency ordering, altering the translation tables to reflect the change. Finally, it increments the appropriate frequency count and adjusts cumulative frequencies accordingly.

3. BAC/CABAC Algorithm

The fundamental arithmetic coding algorithm that was showcased during the last chapter doesn't cover every possible execution scenario. The algorithm doesn't function correctly while sending the intermediate bits during execution while it also doesn't take into account the last remaining bits of variable L. That causes a faulty result to be sent to the decoder, with the following decoding process being erroneous as well.

For that reasons, a variation of the basic algorithm will be showcased in this chapter, one that corrects the aforementioned irregularities and, more importantly, the one that this project is centered around. The algorithm consists of six basic functions. Initialization, binary encoding, renormalization, result output, model update and completion.



```
function BAC/CABAC() {
        Initialization ();
        for each bit {
                Binary_Arithmetic_Encode(bit);
                Renormalise();
                Output();
                Update_MPS();
        }
        Done_Ecode();
}
```

Image 3.1 The core function of the BAC/CABAC algorithm

The arithmetic coding algorithm begins with the initialization of the necessary variables. Following that, for each symbol of the message to be transmitted, utilizes repeatedly the series of functions that were mentioned earlier, as the Image 3.1 suggests. The algorithm first incorporates the current symbol to the coding process by adjusting the range from the previous iteration. It renormalizes the range if need be, and sends the suitable bit to the decoder, again if the need for it arises. Finally it updates the probability model for the next transmission. After the finalization of the aforementioned process, the encoder informs the decoder about the finish of the message tranmission.

3.1 Initialization

At first it would be prudent to present the variables used by the algorithm and their corresponding initialization.

Variable Name	Variable Description
c0	A counter for the appearance of zeroes that gets initialized to 1
c1	A counter for the appearance of ones that gets initialized to 1
t	The total number of processed symbols, that is the sum of C0 and C1, initialized to 2
bo	A counter about the bits that get accumulated during the renormalization process. This variable corresponds to the bits_to_follow that was mentioned and analyzed in Chapter 2, initialized to 0. More information on b0 in the renormalization section
LPS	The Least Probable Symbol. The value of this variable changes when the probability model of the algorithm gets updated, by getting values 0 or 1. It represents the symbol with the smaller appearance frequency.
MPS	The Most Probable Symbol. The function of this variable is identical to that of the LPS but complementary to it. t represents the symbol with the greater appearance frequency.
L	The lowest value of the interval [L, L+R]. It is initialized to 256, which is the ¼ of the initial range [0, 1024]. In the final step of the process, the L will be sent to the decoder.
R	The interval range. This value is used by both the encoding(to compute the updated L and R values) and renormalization (it is the value that controls the prime loop) processes.

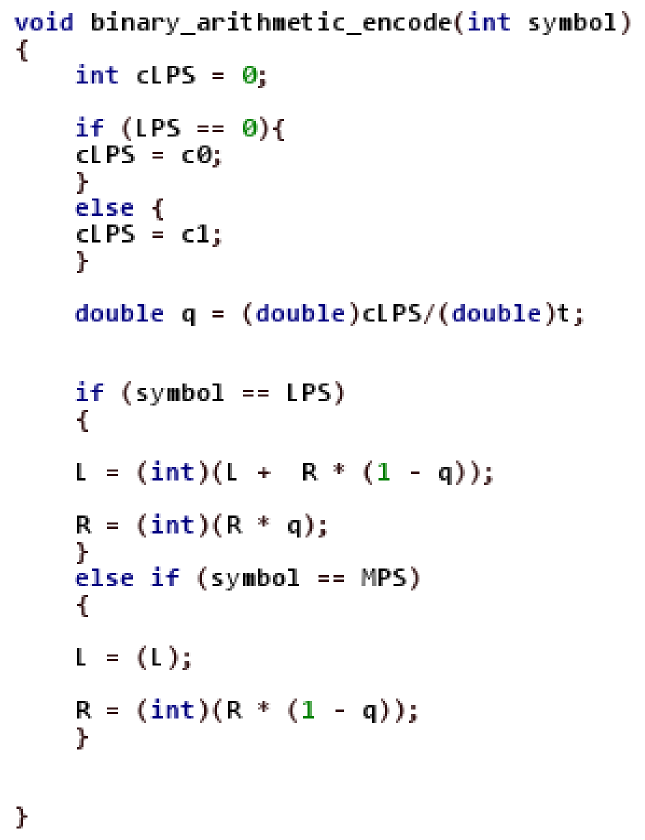
3.2 Binary Coding

The core function of this algorithm is that Binary Coding. At this point, the range R and low interval point L are computed for the current processed symbol. These new values are computed via the following formulas :

$$L = \begin{cases} L + R \times (1 - \frac{c_i}{t}) & , if symbol == LPS \\ L & , if symbol == MPS \end{cases}$$

$$R = \begin{cases} R \times (\frac{c_i}{t}) & , if symbol == LPS \\ R \times (1 - \frac{c_i}{t}) & , if symbol == MPS \end{cases}$$

The Ci / t value is the appearance frequency of the current symbol. The c source code of this process is as follows in Image 3.2



```
void binary_arithmetic_encode(int symbol)
{
    int cLPS = 0;

    if (LPS == 0){
    cLPS = c0;
    }
    else {
    cLPS = c1;
    }

    double q = (double)cLPS/(double)t;


    if (symbol == LPS)
    {

    L = (int)(L +  R * (1 - q));

    R = (int)(R * q);
    }
    else if (symbol == MPS)
    {

    L = (L);

    R = (int)(R * (1 - q));
    }


}
```

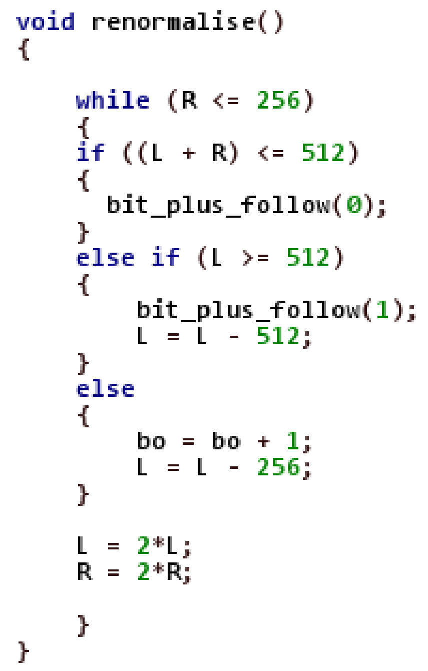
Image 3.2   Binary Arithmetic Encoding module

3.3. Renormalization

The renormalization process is another function of great importance to arithmetic encoding. When the interval range value becomes less than 256, that is the ¼ of the initial 1024 range, the most significant bits of L and R become the same. In that case, those bits can be transmitted instantly since the interval narrowing that follows won't affect them. After that transmission, the *L* and *R* values must be appropriately adapted for the next iteration of the process.

However, the transmission of the most significant bits only when it is guaranteed without errors. That safety measure, makes certain that the value to be transmitted belongs exclusively to the upper or lower bound of the interval.

If the transmission is unsafe, no value is sent. That case though, must be considered in the next transmission. For that reason, the variable *bo* is used, the one that stores the amount of bits to be sent during a following transmission. This counter gets increased during each unsafe transition and is restored to zero during a following safe one. For more detail on the matter of outstanding bits, refer to section 2.4, since the *bo* variable is identical to the *bits_to_follow*. Image 3.3 depicts the renormalization source code.

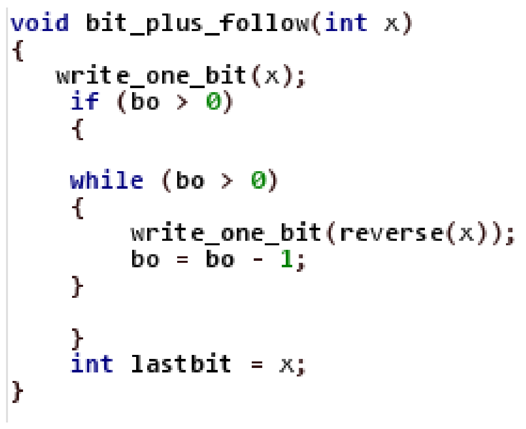


```
void renormalise()
{

    while (R <= 256)
    {
    if ((L + R) <= 512)
    {
      bit_plus_follow(0);
    }
    else if (L >= 512)
    {
        bit_plus_follow(1);
        L = L - 512;
    }
    else
    {
        bo = bo + 1;
        L = L - 256;
    }

    L = 2*L;
    R = 2*R;

    }
}
```

Image 3.3 Renormalization module

3.4 Result Output

Following the renormalization is the transmission of the appropriate bit. Similar to the renormalization process, when the range of the interval becomes sufficiently short, some bits must be sent. So if the *bo* value is zero, only one bit will be transmitted, in the way it was defined in the previous module. In any other case, that is if *bo* has a nonzero value, some additional bits must be sent. The bo variable in essence, provides the amount of bits that must be sent in a following transmission, that will make the decoding process possible. Image 3.4 depicts the output source function.

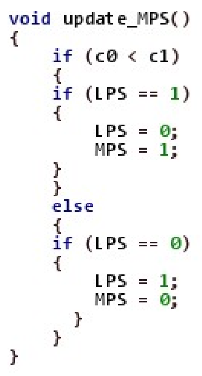


```
void bit_plus_follow(int x)
{
   write_one_bit(x);
    if (bo > 0)
    {

    while (bo > 0)
    {
        write_one_bit(reverse(x));
        bo = bo - 1;
    }

    }
    int lastbit = x;
}
```

Image 3.4 Output bits module

3.5 Model Update

After the transmission of a bit, the probability model must be updated for the next iteration. A simple way to accomplish that is via the c0 and c1 variables, the comparison of which provides the necessary information. The model update source code is as such.

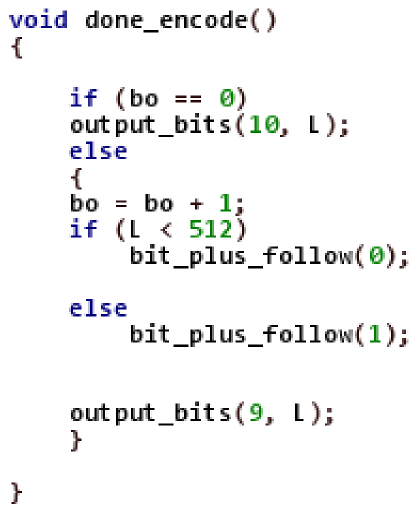


```
void update_MPS()
{
    if (c0 < c1)
    {
    if (LPS == 1)
    {
        LPS = 0;
        MPS = 1;
    }
    }
    else
    {
    if (LPS == 0)
    {
        LPS = 1;
        MPS = 0;
      }
    }
}
```

Image 3.5 Model Update module

3.6 Coding Finalization

Following the examination of each bit belonging to the input message, the value L must be sent. So a series of actions will take place to ensure the correct transmission of the final bits, as well as inform the encoding process completion. In this point, the value of variable L will be sent concluding the whole operation. In case of still existing outstanding bits, they will be sent before the L. Image 3.6 depicts the source code of this function.



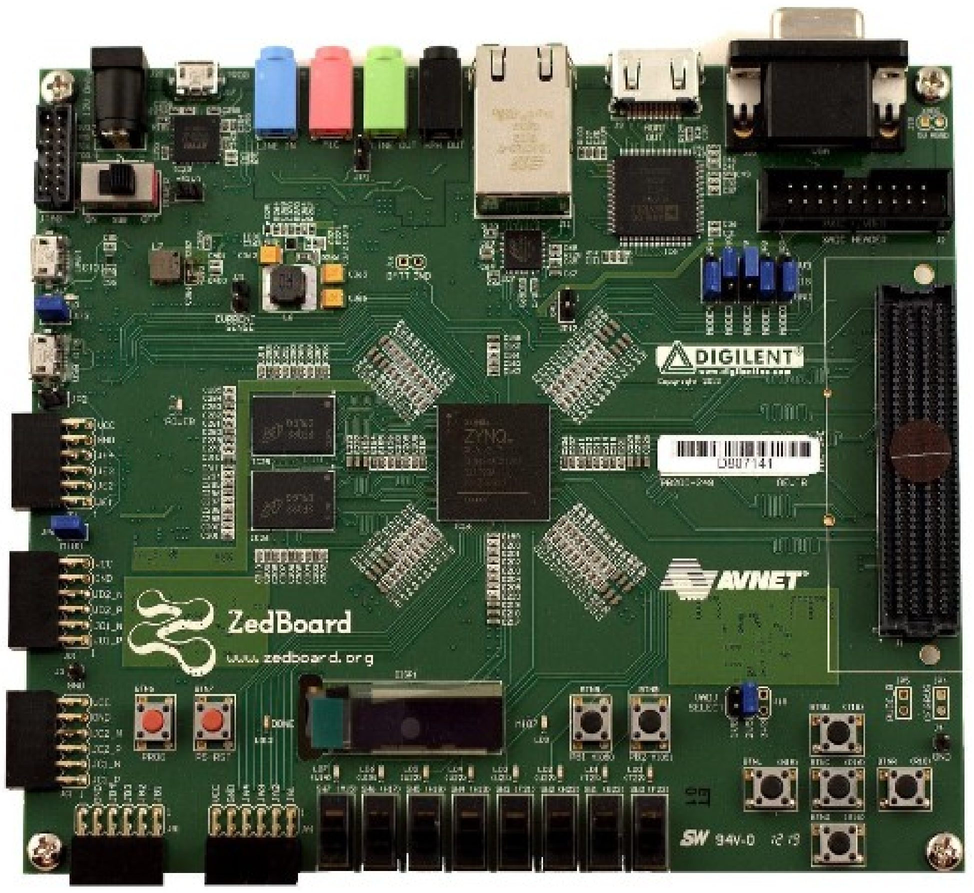
```
void done_encode()
{

    if (bo == 0)
    output_bits(10, L);
    else
    {
    bo = bo + 1;
    if (L < 512)
        bit_plus_follow(0);

    else
        bit_plus_follow(1);


    output_bits(9, L);
    }

}
```

Image 3.6 Coding Finalization module

PART B : The hardware implementation

The main objective of this thesis is to implement the algorithm BAC/CABAC presented in the last chapter, on a hardware level, and deploy it to a reconfigurable architecture, in this case a ZedBoard FPGA board designed by Xilinx. The first implementation is an original circuit designed by the author in Verilog HDL, based on the algorithm, but adapted to be able to be synthesized. The second one is a circuit product of High Level Synthesis, where a source C code, the one I quoted in the previous chapter, is being parsed and synthesized into a Verilog schematic, able to be deployed on an FPGA and also open to improvements using the appropriate tools. It should be prudent to showcase the board on which this project was develeoped.

4. The ZedBoard



The ZedBoard is an evaluation and development board based on the Xilinx Zynq 7000 Extensible Processing Platform. Combining a dual Cortex A9 Processing System (PS) with 85,000 Series 7 Programmable Logic (PL) cells, the Zynq 7000 EPP can be targeted for broad use in many applications. The ZedBoard's robust mix of onboard peripherals and expansion capabilities make it an ideal platform for both novice and experienced designers. It's memory capabilities include 512 MB of DD3 RAM, 256 Mb QSPI Flash memory and it incorporates interfaces like USB 2.0, USB and J-TAG programming, 10/100/1G Ethernet port, as well as VGA and HDMI video ports and an 128x32 OLED display. It also include seven buttons plus two reset buttons and 9 switches for user inputs, two oscillators one clocked at 33,333 MHz and the other at 100 MHz. The ZedBoard is supported by both the Vivado Design Suite as well as the Vivado High-Level Synthesis tool, both of which were of paramount importance in this endeavor. Image 4.1 showcases a design schematic of the I/O capabilities of the board.

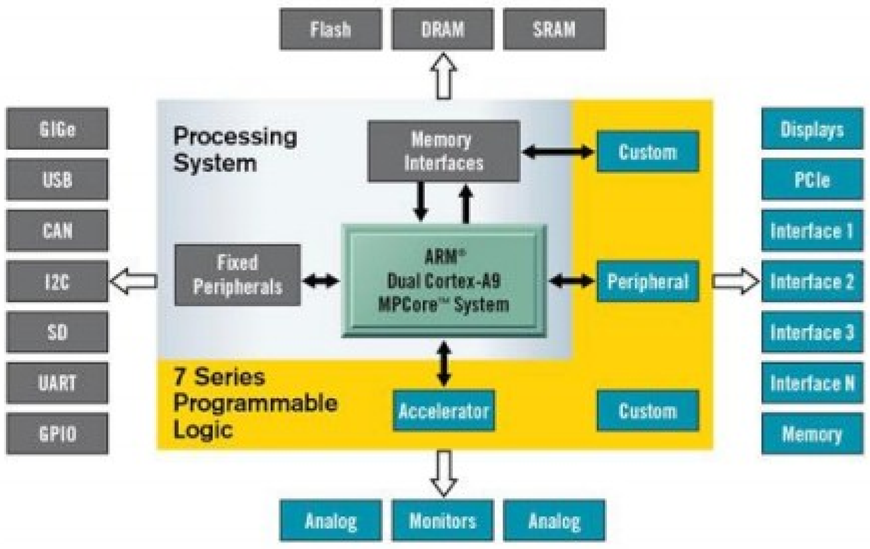


Image 4.1 System architecture's block diagram for Zynq-7000 AP SoC

As mentioned before, the centerpiece of the FPGA is the two Cortex A9 core ARM processor, which deploys the AXI protocol to connect the various peripheral to the processor. One can easily load a Linux distribution OS and using existing drivers connect various peripherals, or make take more realistic results for a custom made IP, thus making this board an extremely versatile tool.

5. The Verilog HDL implementation

The main issue with modeling the CABAC algorithm into a hardware design were the existence and the need of handling, either dividing or multiplying, real numbers. At the core of this algorithm we can find a quotient that always is calculated less than one, the probability of appearance corresponding to the current symbol. Synthesizable Verilog cannot handle either real numbers per se or producing them via non-integer division. This quotient is quite essential to the algorithm since it is used in calculating the Low bound of the interval in each iteration as well as the range.

A different course of action should have been taken regarding the computation of this quotient. The solution came from the source code of the H.264 standard, specifically in the binary encoder of the CABAC algorithm the biari_encode.c file. Instead of calculating products and quotients during each iteration, a Lookup Table system was implemented. The idea is modeling each iteration as a state in an FSM, where the previous state along with the identification of the symbol under examination as a most or least possible symbol (MPS or LPS). The Lookup Tables for next_state_LPS , next_state_MPS and rLPS are showcased in Appendix A. Image 5.1 illustrates a state diagram for the first 10 states of the FSM.

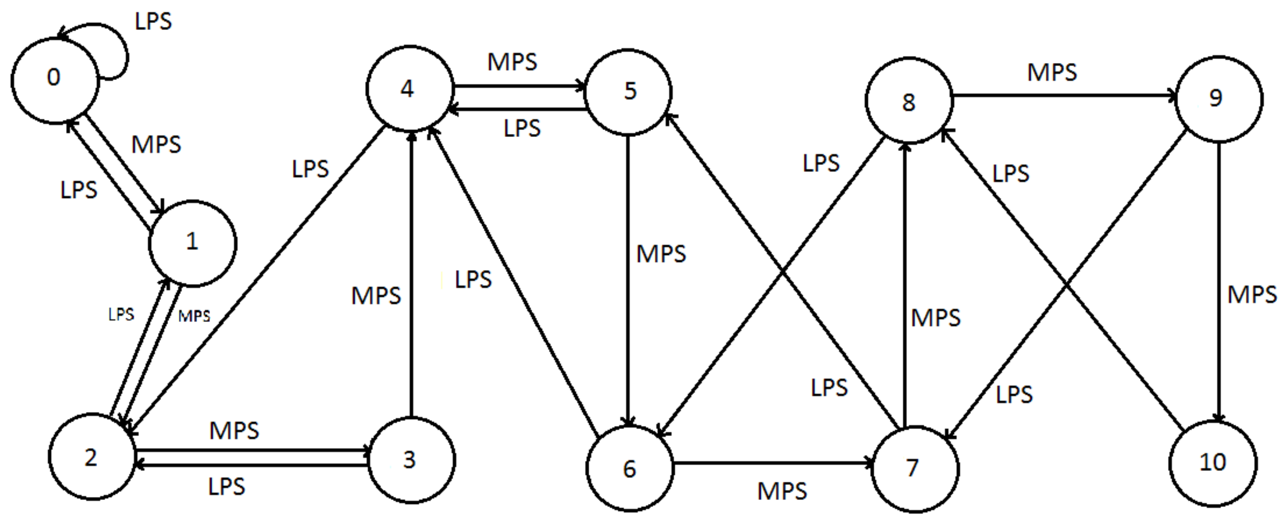


Image 5.1 The State Diagram for the first 10 states of the LPS/MPS FSM

The functions of these numbered states is to point to another Lookup Tabel. This table, named rLPS_table, contains preset values for the product *q*range*, where *q* is the quotient of the division between the number of appearances of the current symbol and the total number of symbols examined thus far, and *range* is the range of the interval for the previous iteration (the name rLPS is given since the product q*range equals the new range for the least probable symbol). This value along with the value for low bound and range of the previous iteration, are the only components needed to compute the low bound and range of the current iteration. In essence with a simple memory load this method saves a large amount of clock cycles that otherwise would be wasted on floating point multiplication and division. The number of the state point to the row of the rLPS table. The column is pointed with a number produced by the logic shift of the previous range by 6 bits which is filtered through the number three via logic conjunction (range>>6& 2'b11).

A second issue was hardware modeling the nested while loop in the renormalize module. Synthesis tools cannot statically analyze a variable loop, resulting in inability to synthesize. For the first while, an always block with an if clause sufficed. Always is similar to an infinite loop and placing an if clause contains it similar to a controlled while loop. For the second, the while inside the called bits_to_follow function, a static case statement was used, with a state for each case of possible outstanding bits count. The main loop terminates when the range becomes greater than 256. In addition each loop doubles said range regardless of branch direction taken inside it. Also the minimum value for the range variable is 2, according to the rLPS Lookup Table in the case of

being at the 63rd state and the current symbol is LPS. So the maximum number of iterations the main while loop will execute is 8, since 2<<8 = 512 > 256. As such a case was added for each outstanding bit count from 1 to 8 with the according result to the final output. In the renormalize module section this conundrum will be showcased more clearly.

5.1 The top level

The top level contains the I/O logic, along with the initialization, the update values and produce final result processes. It also contains instantiations for the Binary Coding and Renormalize submodules. The main inputs on the circuit are the system clock, the system reset, the bin to be coded as well as the length of said bin and a start signal. The outputs contain the coded result and the length of it, to facilitate the decoder.

Internal signals are the range of the current iteration *range*, the lower bound *low*, the current state *state*, the LPS/MPS status *MPS* and *LPS*, the register for the input bin, the current bo counter *bo_next* and the 0 and 1 symbol appearance counters *c1* and *c0.*

The whole circuit works as an FSM, the states of which are assigned with the help of a hardware counter. The states are in actuality the enable signals for the submodules and functions of the circuit. When the start signal is high, the counter initializes to 0 and starts counting. At counter = 1 the state becomes 2'b00. At counter = 2 it becomes 2'b01. At counter = 3 it becomes 2'b11 and at (counter = 11 or done_renorm) the state becomes 2'b10. Finally when counter = 12 it resets to 0.

During the first iteration, the input bin is assigned to a register called *string_to_encode.* The rightmost bit of said register represents the current symbol to encode. Also the MPS and LPS are updated, MPS becoming 0 and LPS becoming 1. The register *length* is loaded with the *input_length.* The rest of the top level logic will be analyzed in chapter 5.4.

5.2 Binary Coding Module

This module is enabled by enable value of 00. This module gets the system clock, system reset and current range, low bound, LPS, MPS and state values. In it there exist the instantiations for the LPS_next_state, MPS_next_state and 4 rLPS lookup tables, one for each column of the original table, as static ROMs.

All of this modules take as input, aside from the clock and reset, the current state, which works as a pointer to the next state, or the next value for the rLPS. The next state is stored in the *BC_state* register, according to the probability status of the current symbol (LPS or MPS). The 4 rLPS table outputs are stored in a four register table, and the logic conjunction ((*range*>>6) & 2'b11) points to the right table entry. Then, according with the aforementioned probability status the new low bound and range are computed in registers *BC_low* and *BC_range* according to the original BAC algorithm showcased earlier.

Finally the three BC register signals are routed to the Renormalize Module.

5.3 Renormalize Module

This is the module containing the most logic of the circuit, calculating the range and low bound for the next iteration, as well as produce part of the result each cycle. The inputs are as before, system clock and reset, the *BC_range* and *BC_low* values loaded at the previous module, the outstanding bit count from the previous iteration and the 2 bit enable signal. Two registers, *R_range* and *R_low* are used to store the values of range and low bound after each normalization.

When the enable signal is assigned the value 01, the inputted BC_range and BC_low values are being loaded in the *R_range* and *R_low* registers along with the bits_outstanding count *bits_outstanding_prev* from the previous iteration into the register *bits_outstanding.* Also, two more registers are initialized to zero, one that keeps the mid-result, *mid_result*, the bits that are being assigned to the output in this iteration, as well as a register for the number of these bits, *output_bits_counter.*

When the enable signal becomes 11, the renormalization process begins. The main clause to trigger this process is the value of the interval range, loaded to register R_range. If the value is higher than 256, or when it becomes through renormalization, the module produces a high signal named *done_renorm* that, outputted to the top level module, causes the main FSM to jump to state 2'b10 (also assigning that value to the enable signal) and ignore the counter value.

If the value of *R_range* is lower or equal to 256, renormalization must take place. There are three distinct cases. The first is triggered if the high point of the interval (*R_Range* + *R_low)* is less than 256. Then, an 1'b0 bit is added to the mid_result register after it was left shifted by one bit and to the *output_bits_counter* is added 1 plus the value of *bits_outstanding.* In the case of *bits_outstanding* is greater than zero, a number of bits equal to that register, of reverse polarity than the bit that was added previously, are added to the *mid_result* after it is left shifted by that number. The assignment is made through a case statement of 8 static cases, corresponding to *bits_outstanding* value 1 to 8. After that, the *bits_outstanding* counter becomes zero, and the R_range and R_low values are left shifted by 1. If the value of R_low is greater or equal to 512, a similar process is followed. The mid_result is left shifted by one bit and a 1'b1 bit is added to it. The *output_bits_counter* is increased like before and, in the case of *bits_outstanding* is greater than zero, a number of bits of reverse polarity to the bit that was just added, are added to the *mid_result* register after it is left shifted by that number, using the same case statement as before. Following that, the R_range and R_low (R_low is first decreased by 512) values are left shifted by 1.

Finally, the third and last case is the ambiguous, narrow interval, when the high point of the interval (R_low + R_range) is larger than 256 and the low point of the interval is less than 512 (the middle portion of the initial interval) . In this case the *bits_outstanding* counter is increased by 1, followed by the left shift by 1 bit of both *R_rang*e and *R_low.* As it is showcased until now, the R_range is left shifted by 1 in all three cases, and since the main clause for these cases to be executed is R to be less or equal to 256, the number of executions is limited to the number of left shifts the minimum possible value of R_range can go through. That minimum value can be seen in the rLPS Lookup Table on Appendix A, which is 2, in the case the previous state is state 63 and the current symbol is *LPS* (and *BC_rang*e will be assigned the value rLPS and not the *range* – rLPS). So, the number of possible iterations executed in the Renormalize module is maximum 8. That is the reason that there are 8 cases of outstanding_bits value, and the reason that the main hardware counter increases maximum 8 times before assigning the enable signal to the next function.

The *R_range*, *R_low*, *mid_result (*as *final_result)*, bits_outstanding, as *R_bo* (in case the R_range became greater than 256 before adding them to the *mid_result*) and *output_bits_counter* are routed back to the top module.

5.4 Model Update

This is the final stage of the circuit, where all the essential values produced during the current iteration, are loaded to the main module's corresponding registers, and the probability model variables are updated. More specifically :

range takes the value of *R_range*

low takes the value of *R_low*

bo_next takes the value of *R_bo*

state takes the value of *BC_state*

result is left shifted by amount of *output_bits_counter* and the *mid_result* is added to it

0 and 1 bit appearance counters (*c0* and *c1*) are increased based on the current symbol

LPS and *MPS* statuses are updated based on the aforementioned counters

string_to_encode is right shifted by one to prepare the next symbol to encode

length is decreased by 1 and

output_bits_number is increased by *output_bits_counter*

5.5 Produce output values

When the length register becomes zero, it means that all the bits of the input bin have been encoded. This action triggers the finalization of the process, where there is performed a final check for outstanding bits, similar to the case statements in the renormalization process. Following that, the *result* register is shifted by 11 bits and the value of *low* updated after the last iteration is added to it. The *output_bits_number* is increased also by 11 and finally the signal *done_encode* is raised to 1, signaling the conclusion of the encoding process.

The source code of the modules in Verilog HDL are showcased in Appendix A.

5.6 Simulation

A necessary step to validate any design is to simulate and observe the results. In this case, having the software code as a reference point, the results of both are compared to test the validity of the circuit. Image 5.3 illustrates the simulation process by means of a waveform, with input bin = 010101010101 (decimal value of 1365 for convenience), input_length = 12 and clock cycle of 20 ns.

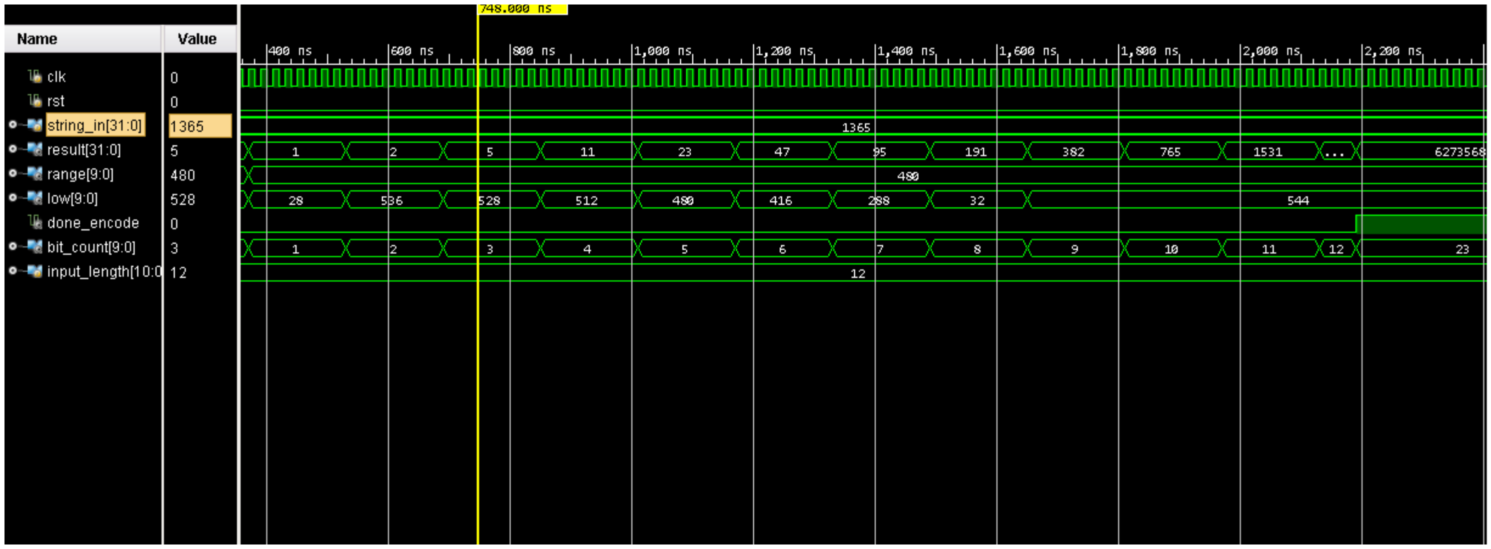
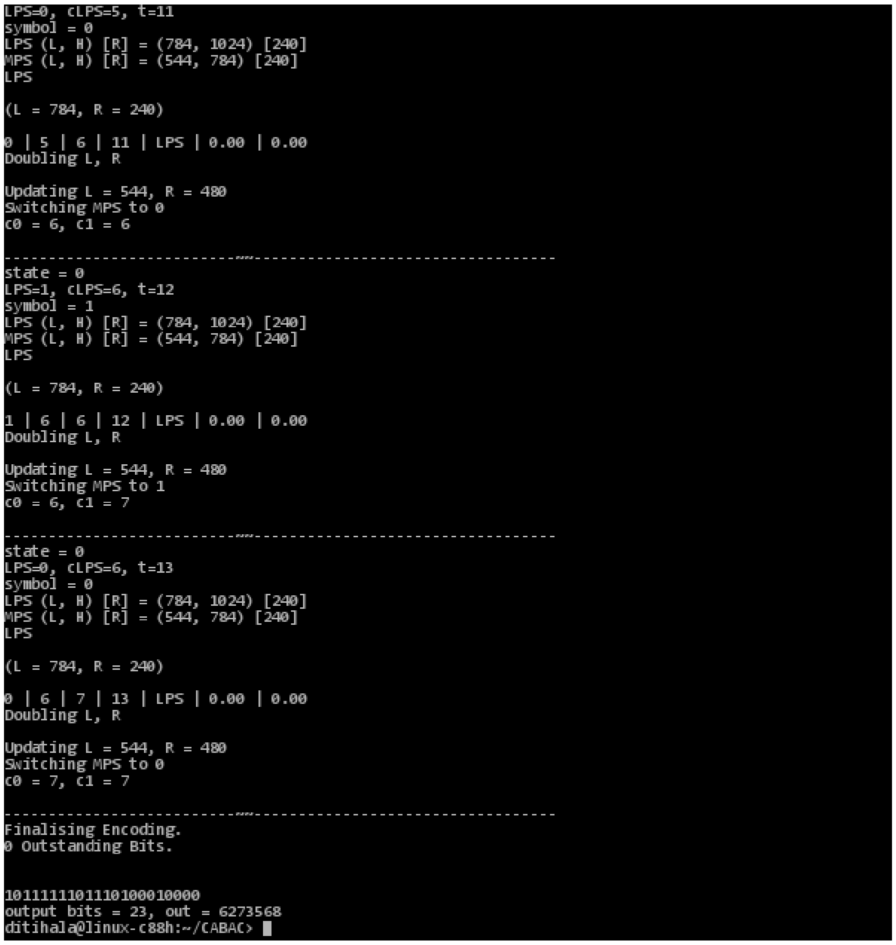


Image 5.3 Circuit Simulation using Vivado Simulation

In Image 5.4 the analogous software execution is shown :

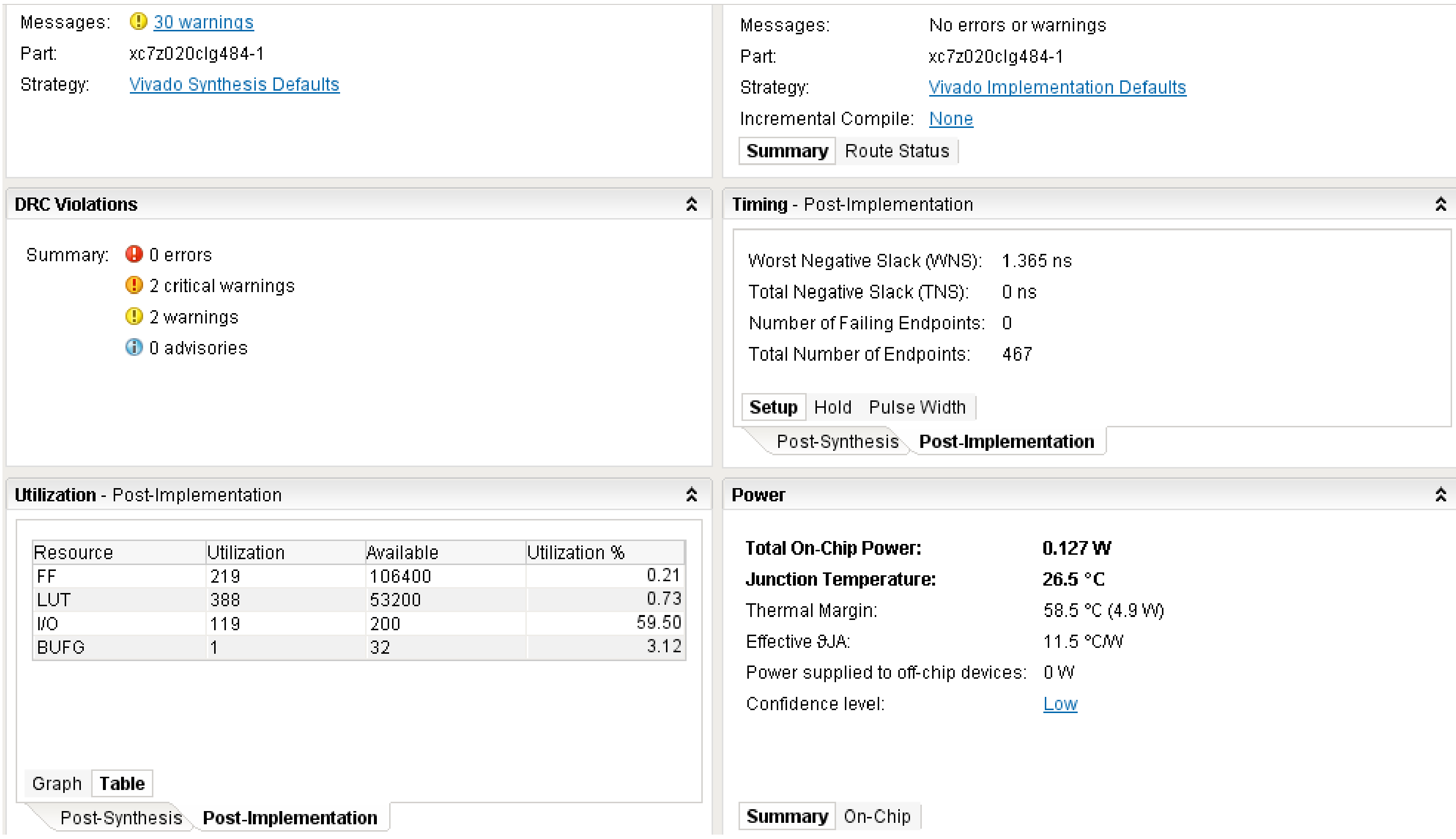


```
LPS=0, cLPS=5, t=11
symbol = 0
LPS (L, H) [R] = (784, 1024) [240]
MPS (L, H) [R] = (544, 784) [240]
LPS

(L = 784, R = 240)

0 | 5 | 6 | 11 | LPS | 0.00 | 0.00
Doubling L, R

Updating L = 544, R = 480
Switching MPS to 0
c0 = 6, c1 = 6

--------------------------~~---------------------------------
state = 0
LPS=1, cLPS=6, t=12
symbol = 1
LPS (L, H) [R] = (784, 1024) [240]
MPS (L, H) [R] = (544, 784) [240]
LPS

(L = 784, R = 240)

1 | 6 | 6 | 12 | LPS | 0.00 | 0.00
Doubling L, R

Updating L = 544, R = 480
Switching MPS to 1
c0 = 6, c1 = 7

--------------------------~~---------------------------------
state = 0
LPS=0, cLPS=6, t=13
symbol = 0
LPS (L, H) [R] = (784, 1024) [240]
MPS (L, H) [R] = (544, 784) [240]
LPS

(L = 784, R = 240)

0 | 6 | 7 | 13 | LPS | 0.00 | 0.00
Doubling L, R

Updating L = 544, R = 480
Switching MPS to 0
c0 = 7, c1 = 7

--------------------------~~---------------------------------
Finalising Encoding.
0 Outstanding Bits.


10111111101110100010000
output bits = 23, out = 6273568
ditihala@linux-c88h:~/CABAC> █
```

Image 5.4 Execution of the BAC algorithm with LUTs

As it can be seen, the results of software execution and circuit simulation coincide, meaning the validation of the circuit functionality (range and low registers are kept for debugging reasons in the simulation, to better showcase the validity of each cycle run by the circuit. Also the output value is kept in decimal form for convenience)

The next and final step is to synthesize the circuit to be able to program it onto an FPGA, using the Xilinx ZedBoard as a reference platform. Again, using the Vivado automated process, the circuit was validated, analyzed, synthesized and implemented. Image 5.5 depicts the resources used (in LUTs and flip-flops) and the timing analysis of the implementation. Finally in Image 5.6 is show the implemented overview of the circuit.



Messages:	30 warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Synthesis Defaults

Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Implementation Defaults
Incremental Compile:	None

DRC Violations

Summary: 0 errors
2 critical warnings
2 warnings
0 advisories

Timing - Post-Implementation

Worst Negative Slack (WNS):	1.365 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	467

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
FF	219	106400	0.21
LUT	388	53200	0.73
I/O	119	200	59.50
BUFG	1	32	3.12

Power

Total On-Chip Power:	**0.127 W**
Junction Temperature:	**26.5 °C**
Thermal Margin:	58.5 °C (4.9 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Image 5.5 Post Implementation Measurements

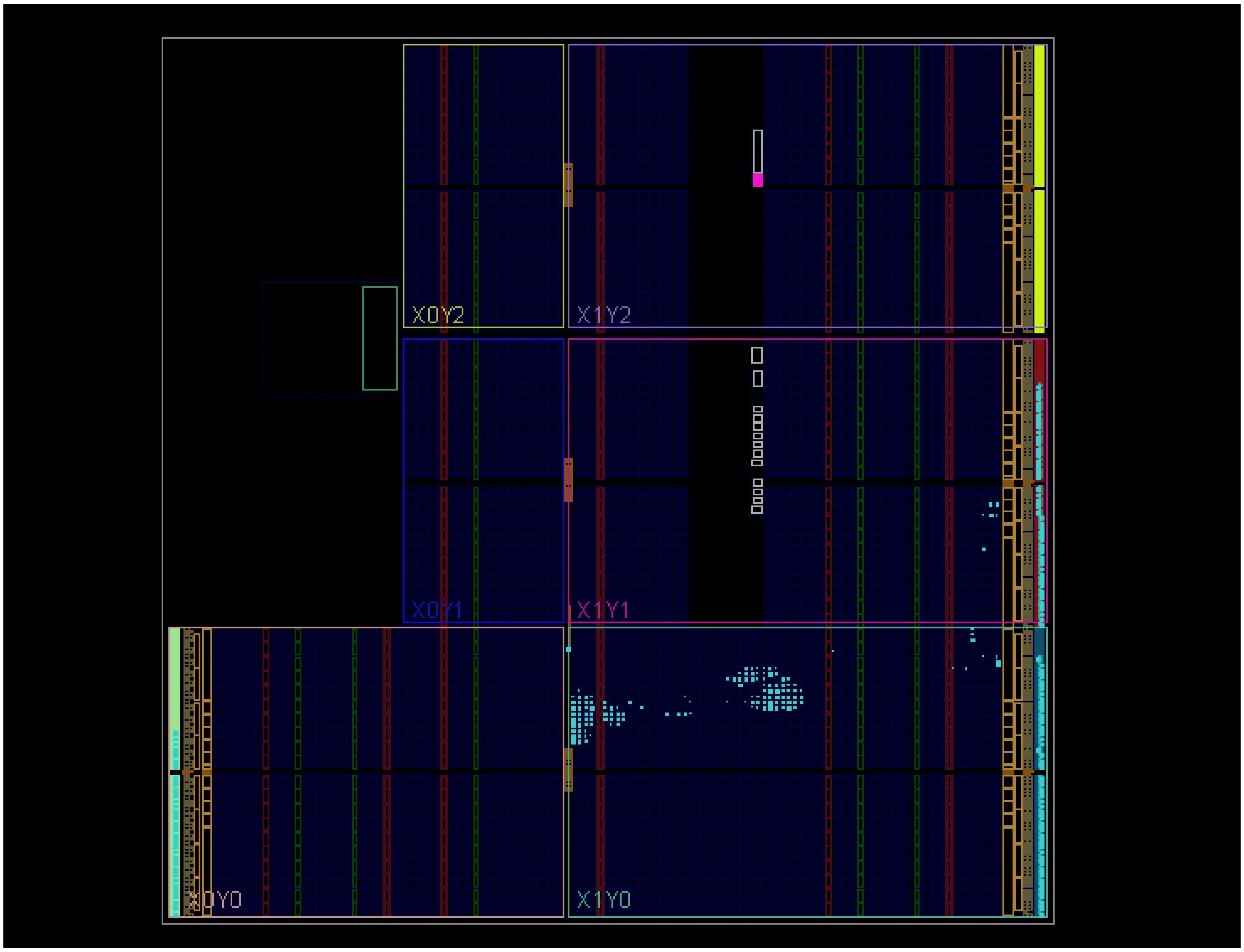


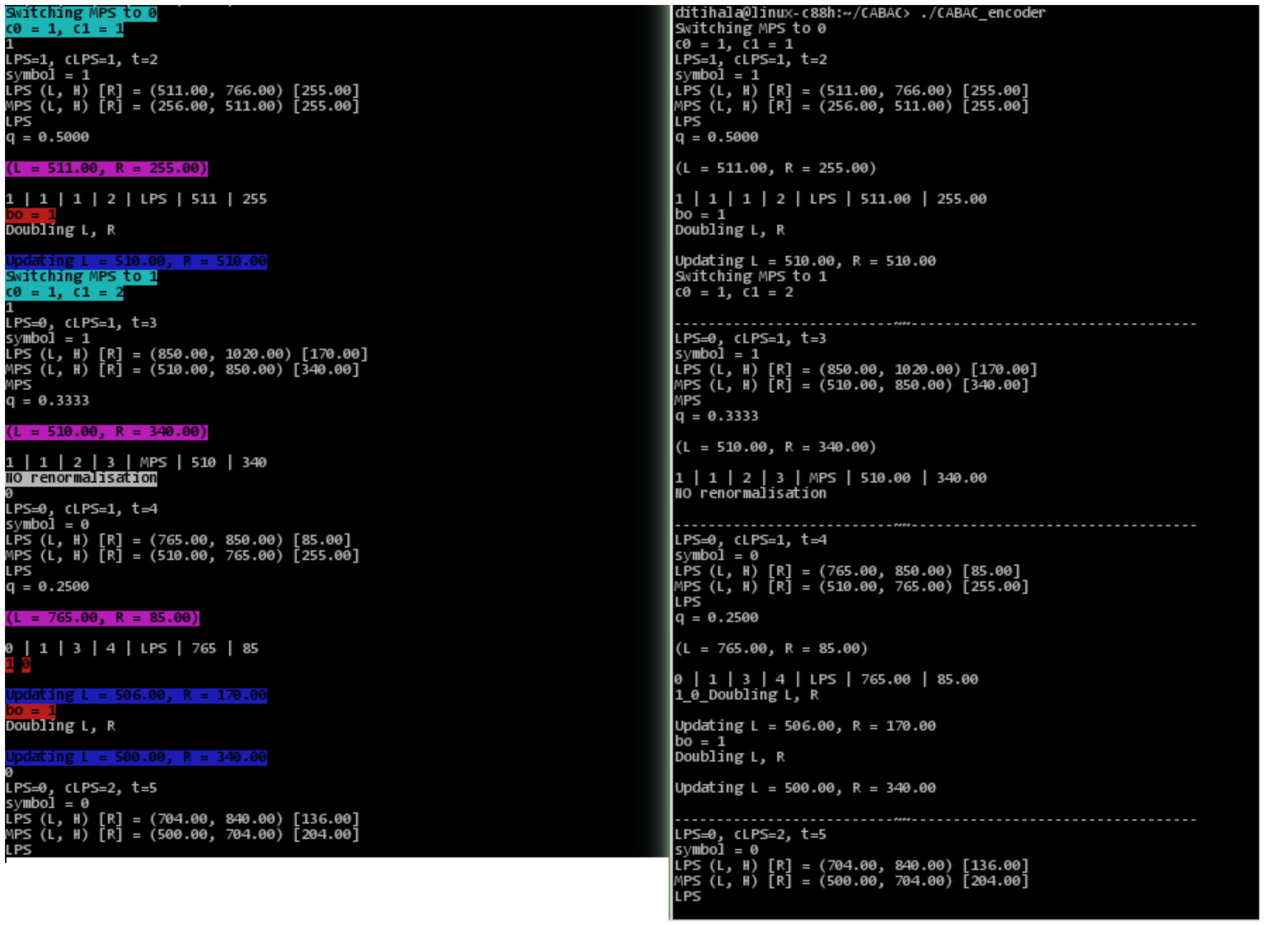
Image 5.6 Implemented overview of the circuit

6. HLS Implementation

The next step of this project, is to synthesize a circuit similar to the previous one, this time in an automated fashion by utilizing the Vivado HLS tool. Complementary to this, the produced circuit will be exported as a stand-alone IP and connected to a Zynq processor as an accelerator, forming a unique SoC based on the CABAC encoding.

6.1 The source Code

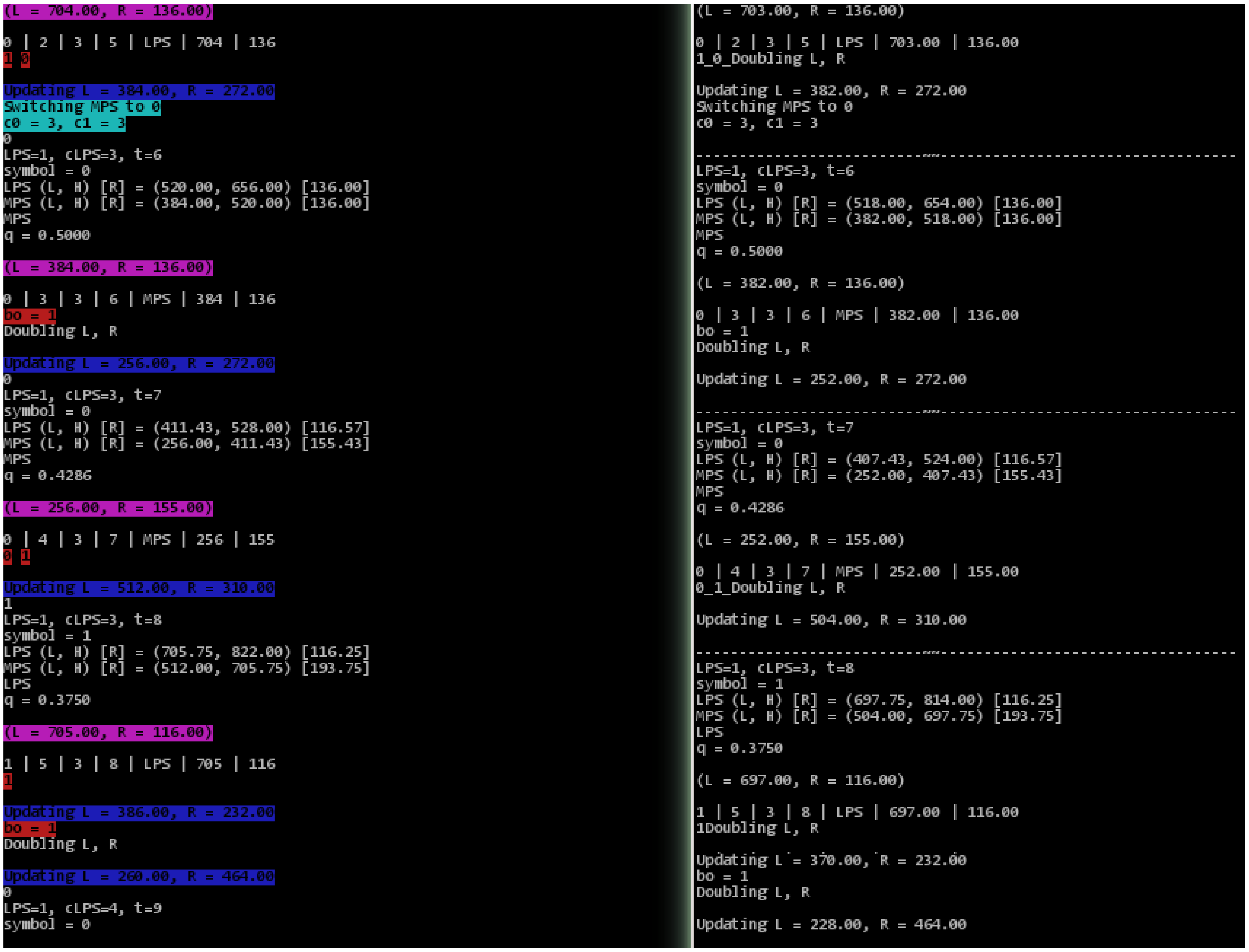
The first step is to create a piece of software in C language, that will capture the algorithm and will be fed to the HLS tool to be synthesized. The original code developed by professor Mr. Sotiriou, is in awk scripting language. The main advantage of awk is its offering unlimited precision, as noted by the professor, executing flawlessly the algorithm without any loss. However this is not the case in C, where even double precision variables still have limited precision. A side by side execution of the two programs can illustrate the slow but gradual degeneration of the computed values in the C execution. Images 6.1, 6.2 and 6.3 illustrate this conundrum.



```
Switching MPS to 0
c0 = 1, c1 = 1
1
LPS=1, cLPS=1, t=2
symbol = 1
LPS (L, H) [R] = (511.00, 766.00) [255.00]
MPS (L, H) [R] = (256.00, 511.00) [255.00]
LPS
q = 0.5000

(L = 511.00, R = 255.00)

1 | 1 | 1 | 2 | LPS | 511 | 255
bo = 1
Doubling L, R

Updating L = 510.00, R = 510.00
Switching MPS to 1
c0 = 1, c1 = 2
1
LPS=0, cLPS=1, t=3
symbol = 1
LPS (L, H) [R] = (850.00, 1020.00) [170.00]
MPS (L, H) [R] = (510.00, 850.00) [340.00]
MPS
q = 0.3333

(L = 510.00, R = 340.00)

1 | 1 | 2 | 3 | MPS | 510 | 340
NO renormalisation
0
LPS=0, cLPS=1, t=4
symbol = 0
LPS (L, H) [R] = (765.00, 850.00) [85.00]
MPS (L, H) [R] = (510.00, 765.00) [255.00]
LPS
q = 0.2500

(L = 765.00, R = 85.00)

0 | 1 | 3 | 4 | LPS | 765 | 85
1 0

Updating L = 506.00, R = 170.00
bo = 1
Doubling L, R

Updating L = 500.00, R = 340.00
0
LPS=0, cLPS=2, t=5
symbol = 0
LPS (L, H) [R] = (704.00, 840.00) [136.00]
MPS (L, H) [R] = (500.00, 704.00) [204.00]
LPS
```

```
ditihala@linux-c88h:~/CABAC> ./CABAC_encoder
Switching MPS to 0
c0 = 1, c1 = 1
LPS=1, cLPS=1, t=2
symbol = 1
LPS (L, H) [R] = (511.00, 766.00) [255.00]
MPS (L, H) [R] = (256.00, 511.00) [255.00]
LPS
q = 0.5000

(L = 511.00, R = 255.00)

1 | 1 | 1 | 2 | LPS | 511.00 | 255.00
bo = 1
Doubling L, R

Updating L = 510.00, R = 510.00
Switching MPS to 1
c0 = 1, c1 = 2

---------------------------~~-----------------------------------
LPS=0, cLPS=1, t=3
symbol = 1
LPS (L, H) [R] = (850.00, 1020.00) [170.00]
MPS (L, H) [R] = (510.00, 850.00) [340.00]
MPS
q = 0.3333

(L = 510.00, R = 340.00)

1 | 1 | 2 | 3 | MPS | 510.00 | 340.00
NO renormalisation

---------------------------~~-----------------------------------
LPS=0, cLPS=1, t=4
symbol = 0
LPS (L, H) [R] = (765.00, 850.00) [85.00]
MPS (L, H) [R] = (510.00, 765.00) [255.00]
LPS
q = 0.2500

(L = 765.00, R = 85.00)

0 | 1 | 3 | 4 | LPS | 765.00 | 85.00
1_0_Doubling L, R

Updating L = 506.00, R = 170.00
bo = 1
Doubling L, R

Updating L = 500.00, R = 340.00

---------------------------~~-----------------------------------
LPS=0, cLPS=2, t=5
symbol = 0
LPS (L, H) [R] = (704.00, 840.00) [136.00]
MPS (L, H) [R] = (500.00, 704.00) [204.00]
LPS
```

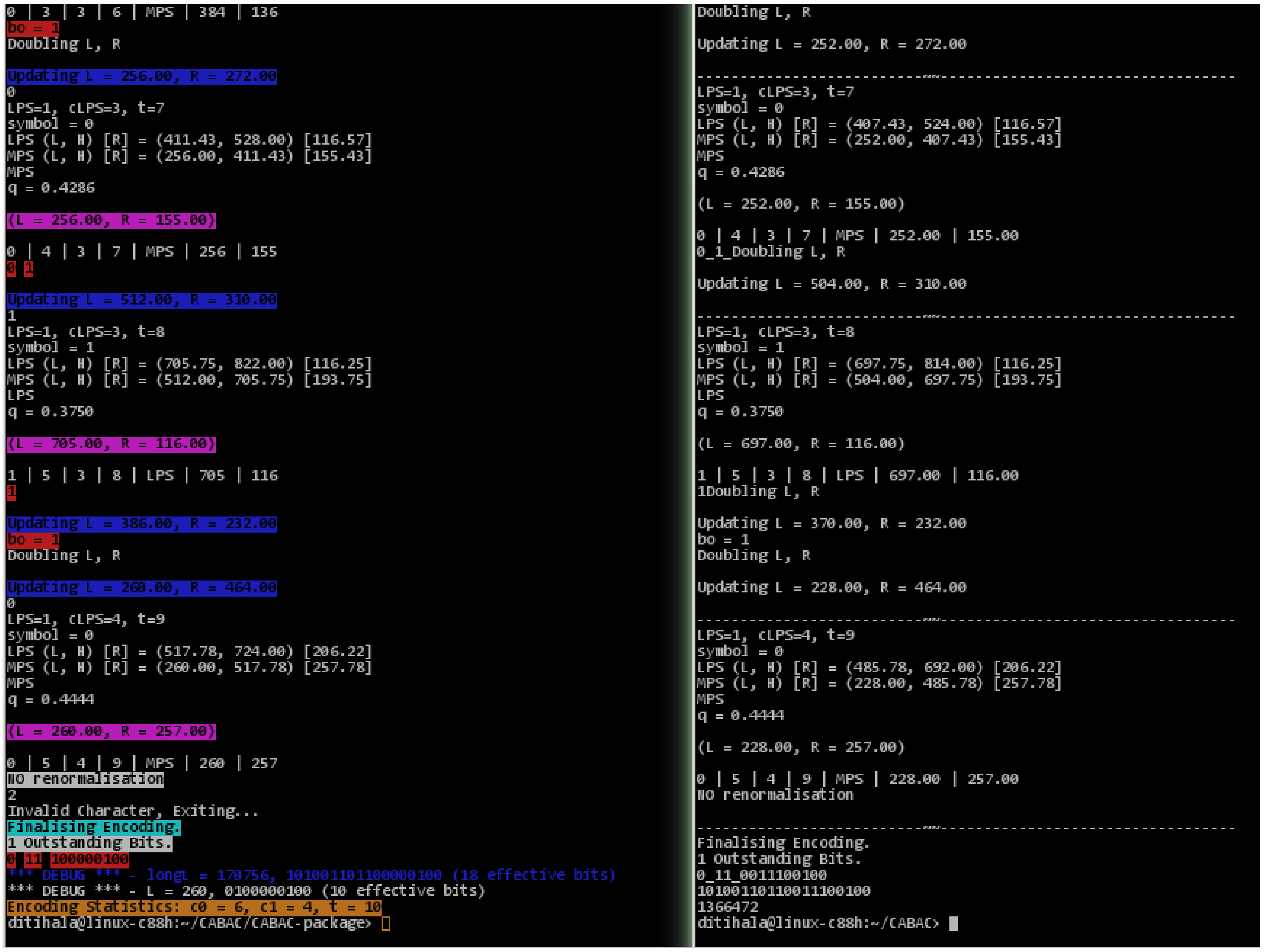
Image 6.1 Snapshot of a side by side execution in C and AWK



```
(L = 704.00, R = 136.00)

0 | 2 | 3 | 5 | LPS | 704 | 136
1 0

Updating L = 384.00, R = 272.00
Switching MPS to 0
c0 = 3, c1 = 3
0
LPS=1, cLPS=3, t=6
symbol = 0
LPS (L, H) [R] = (520.00, 656.00) [136.00]
MPS (L, H) [R] = (384.00, 520.00) [136.00]
MPS
q = 0.5000

(L = 384.00, R = 136.00)

0 | 3 | 3 | 6 | MPS | 384 | 136
bo = 1
Doubling L, R

Updating L = 256.00, R = 272.00
0
LPS=1, cLPS=3, t=7
symbol = 0
LPS (L, H) [R] = (411.43, 528.00) [116.57]
MPS (L, H) [R] = (256.00, 411.43) [155.43]
MPS
q = 0.4286

(L = 256.00, R = 155.00)

0 | 4 | 3 | 7 | MPS | 256 | 155
0 1

Updating L = 512.00, R = 310.00
1
LPS=1, cLPS=3, t=8
symbol = 1
LPS (L, H) [R] = (705.75, 822.00) [116.25]
MPS (L, H) [R] = (512.00, 705.75) [193.75]
LPS
q = 0.3750

(L = 705.00, R = 116.00)

1 | 5 | 3 | 8 | LPS | 705 | 116
1

Updating L = 386.00, R = 232.00
bo = 1
Doubling L, R

Updating L = 260.00, R = 464.00
0
LPS=1, cLPS=4, t=9
symbol = 0
```

```
(L = 703.00, R = 136.00)

0 | 2 | 3 | 5 | LPS | 703.00 | 136.00
1_0_Doubling L, R

Updating L = 382.00, R = 272.00
Switching MPS to 0
c0 = 3, c1 = 3

-----------------------------~~----------------------------------
LPS=1, cLPS=3, t=6
symbol = 0
LPS (L, H) [R] = (518.00, 654.00) [136.00]
MPS (L, H) [R] = (382.00, 518.00) [136.00]
MPS
q = 0.5000

(L = 382.00, R = 136.00)

0 | 3 | 3 | 6 | MPS | 382.00 | 136.00
bo = 1
Doubling L, R

Updating L = 252.00, R = 272.00

-----------------------------~~----------------------------------
LPS=1, cLPS=3, t=7
symbol = 0
LPS (L, H) [R] = (407.43, 524.00) [116.57]
MPS (L, H) [R] = (252.00, 407.43) [155.43]
MPS
q = 0.4286

(L = 252.00, R = 155.00)

0 | 4 | 3 | 7 | MPS | 252.00 | 155.00
0_1_Doubling L, R

Updating L = 504.00, R = 310.00

-----------------------------~~----------------------------------
LPS=1, cLPS=3, t=8
symbol = 1
LPS (L, H) [R] = (697.75, 814.00) [116.25]
MPS (L, H) [R] = (504.00, 697.75) [193.75]
LPS
q = 0.3750

(L = 697.00, R = 116.00)

1 | 5 | 3 | 8 | LPS | 697.00 | 116.00
1Doubling L, R

Updating L = 370.00, R = 232.00
bo = 1
Doubling L, R

Updating L = 228.00, R = 464.00
```

Image 6.2 Snapshot of a side by side execution in C and AWK



```
0 | 3 | 3 | 6 | MPS | 384 | 136
bo = 1
Doubling L, R

Updating L = 256.00, R = 272.00
0
LPS=1, cLPS=3, t=7
symbol = 0
LPS (L, H) [R] = (411.43, 528.00) [116.57]
MPS (L, H) [R] = (256.00, 411.43) [155.43]
MPS
q = 0.4286

(L = 256.00, R = 155.00)

0 | 4 | 3 | 7 | MPS | 256 | 155
0 1

Updating L = 512.00, R = 310.00
1
LPS=1, cLPS=3, t=8
symbol = 1
LPS (L, H) [R] = (705.75, 822.00) [116.25]
MPS (L, H) [R] = (512.00, 705.75) [193.75]
LPS
q = 0.3750

(L = 705.00, R = 116.00)

1 | 5 | 3 | 8 | LPS | 705 | 116
1

Updating L = 386.00, R = 232.00
bo = 1
Doubling L, R

Updating L = 260.00, R = 464.00
0
LPS=1, cLPS=4, t=9
symbol = 0
LPS (L, H) [R] = (517.78, 724.00) [206.22]
MPS (L, H) [R] = (260.00, 517.78) [257.78]
MPS
q = 0.4444

(L = 260.00, R = 257.00)

0 | 5 | 4 | 9 | MPS | 260 | 257
NO renormalisation
2
Invalid Character, Exiting...
Finalising Encoding.
1 Outstanding Bits.
0 11 100000100
*** DEBUG *** - longL = 170756, 101001101100000100 (18 effective bits)
*** DEBUG *** - L = 260, 0100000100 (10 effective bits)
Encoding Statistics: c0 = 6, c1 = 4, t = 10
ditihala@linux-c88h:~/CABAC/CABAC-package>
```

```
Doubling L, R

Updating L = 252.00, R = 272.00

-------------------------------------------------------------------------
LPS=1, cLPS=3, t=7
symbol = 0
LPS (L, H) [R] = (407.43, 524.00) [116.57]
MPS (L, H) [R] = (252.00, 407.43) [155.43]
MPS
q = 0.4286

(L = 252.00, R = 155.00)

0 | 4 | 3 | 7 | MPS | 252.00 | 155.00
0_1_Doubling L, R

Updating L = 504.00, R = 310.00

-------------------------------------------------------------------------
LPS=1, cLPS=3, t=8
symbol = 1
LPS (L, H) [R] = (697.75, 814.00) [116.25]
MPS (L, H) [R] = (504.00, 697.75) [193.75]
LPS
q = 0.3750

(L = 697.00, R = 116.00)

1 | 5 | 3 | 8 | LPS | 697.00 | 116.00
1Doubling L, R

Updating L = 370.00, R = 232.00
bo = 1
Doubling L, R

Updating L = 228.00, R = 464.00

-------------------------------------------------------------------------
LPS=1, cLPS=4, t=9
symbol = 0
LPS (L, H) [R] = (485.78, 692.00) [206.22]
MPS (L, H) [R] = (228.00, 485.78) [257.78]
MPS
q = 0.4444

(L = 228.00, R = 257.00)

0 | 5 | 4 | 9 | MPS | 228.00 | 257.00
NO renormalisation

-------------------------------------------------------------------------
Finalising Encoding.
1 Outstanding Bits.
0_11_0011100100
1010011011001110010 0
1366472
ditihala@linux-c88h:~/CABAC>
```

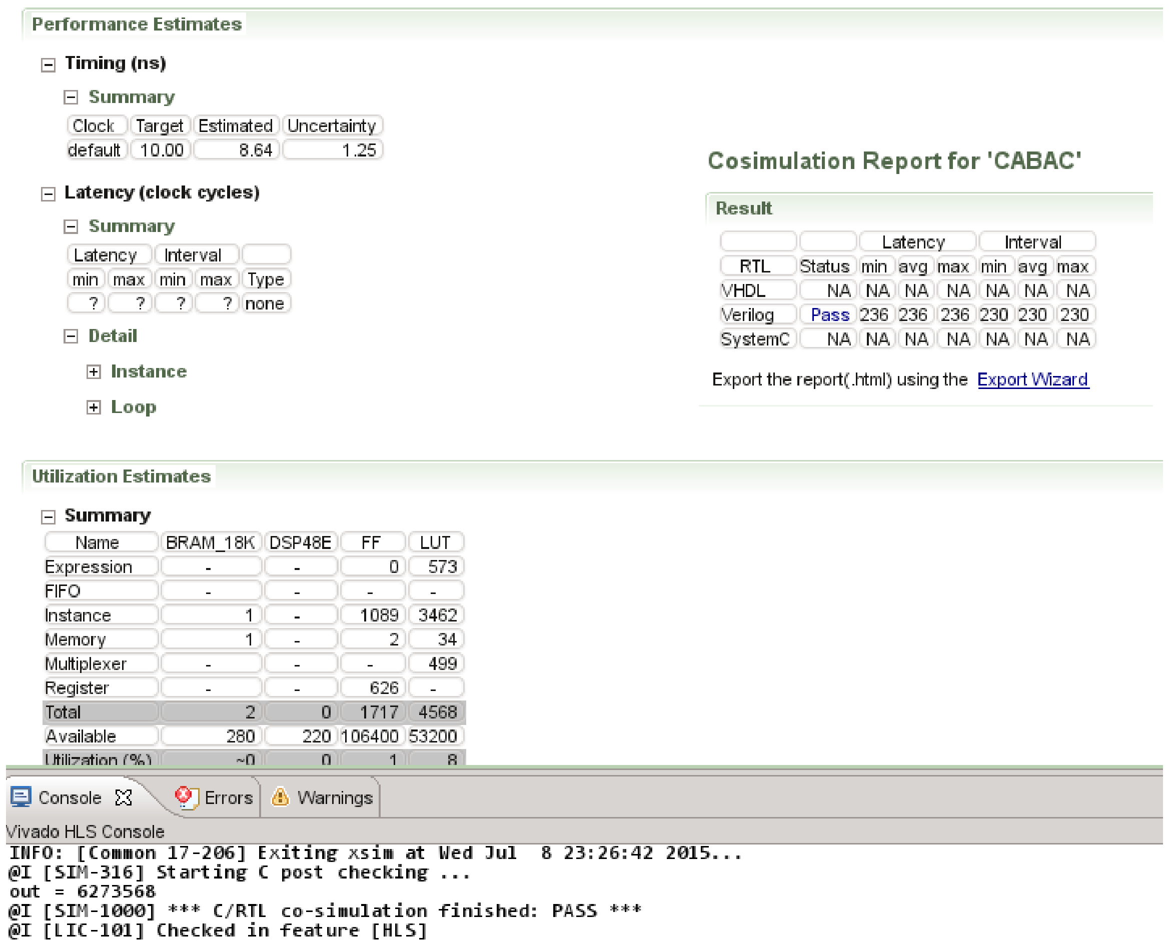
Image 6.3 Snapshot of a side by side execution in C and AWK

As one can see, the L computed in the last iteration is different in these two executions, a fact that results in an also different output string, though the bits sent prior to the L are identical. In other words, there is no way to truly emulate in C the pure BAC/CABAC algorithm. In addition to that having double precision variables and floating point multiplication and division will add unnecessary complexity and latency to the circuit that will be produced form that code. So once again, the LUT table method, like in the previous chapter, will be implemented in the C code. The output results will of course be different, since all intermediate values are calculated differently, but the encoding is not a standalone process. Since the decoder will also be using LUTs there will be no issue in decoding process. The only change to the original BAC/CABAC algorithm is the addition of a variable *state*, that will stipulate the current state/pointer to the rLPS Lookup table.

Let it be noted also, that all updating variables such as R, L, MPS, LPS etc. are stated as static globals, since pointer synthesis in HLS is very tricky to achieve and adds unnecessary complexity to the design. The source C code can be found in Appendix B.

6.2 High Level Synthesis and enhancements

Using the quite user friendly GUI provided by the Vivado HLS tool, synthesis is achieved instantaneously by just pushing the corresponding button. The synthesized design is stored and can be browsed freely, although the produced Verilog code is not friendly to the reader. The reports for resources used, timing and the cosimulation with the source C code along with the running time in clock cycles can be seen in image 6.4



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.64	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

Loop

Cosimulation Report for 'CABAC'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	236	236	236	230	230	230
SystemC	NA	NA	NA	NA	NA	NA	NA

Export the report(.html) using the Export Wizard

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	573
FIFO	-	-	-	-
Instance	1	-	1089	3462
Memory	1	-	2	34
Multiplexer	-	-	-	499
Register	-	-	626	-
Total	2	0	1717	4568
Available	280	220	106400	53200
Utilization (%)	~0	0	1	8

Console Errors Warnings

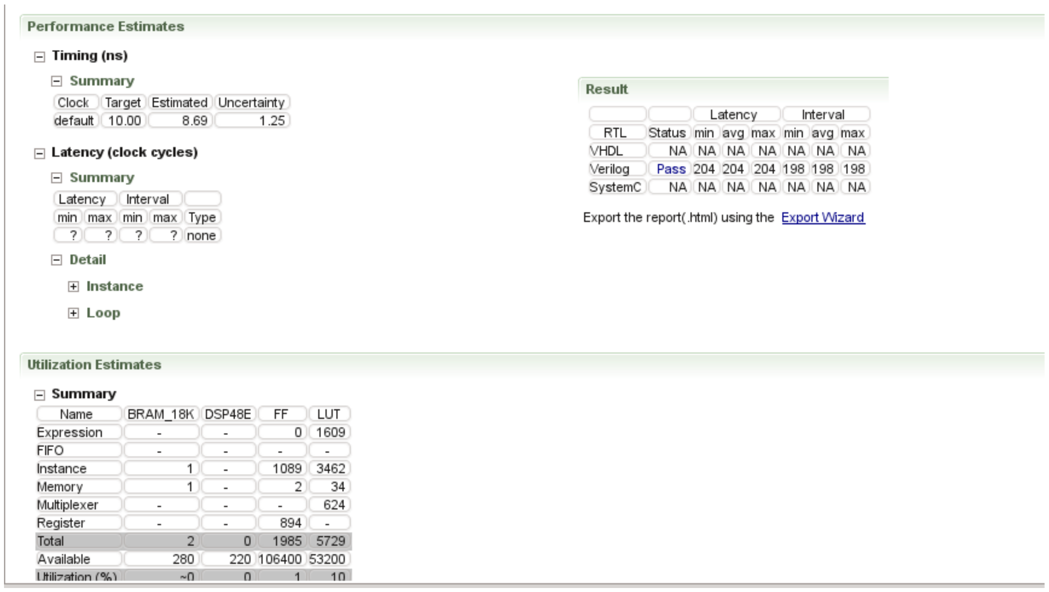
Vivado HLS Console

```
INFO: [Common 17-206] Exiting xsim at Wed Jul  8 23:26:42 2015...
@I [SIM-316] Starting C post checking ...
out = 6273568
@I [SIM-1000] *** C/RTL co-simulation finished: PASS ***
@I [LIC-101] Checked in feature [HLS]
```

Image 6.4 Reports for the first try at High Level Synthesis

As we can see, the output result coincides with the simulation of the Verilog circuit in the previous chapter, although the execution time is more than double. Please refer to the simulation waveform of the previous chapter showcasing that the circuit finished the encoding process in roughly 2000 ns on a 20 ns clock, which is approximately 100 clock cycles.

The HLS tool however, provides methods to improve upon the circuit, by means of loop unrolling and pipelining. The first step is to implement loop unrolling. Choosing the heaviest loops on the code, and the most certain to execute like the integer-to-binary loop or the main encoding process, a directive was added to them specifying a loop unroll by a factor of 2. This means that two loops in assembly level are unrolled, forming a software pipeline with them at the center, and initialization and finalizing pieces of code with preset displacement values for the store and load operations. The results for this enhancement can be shown in Image 6.5


Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.69	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

Loop

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	204	204	204	198	198	198
SystemC	NA	NA	NA	NA	NA	NA	NA

Export the report(.html) using the Export Wizard

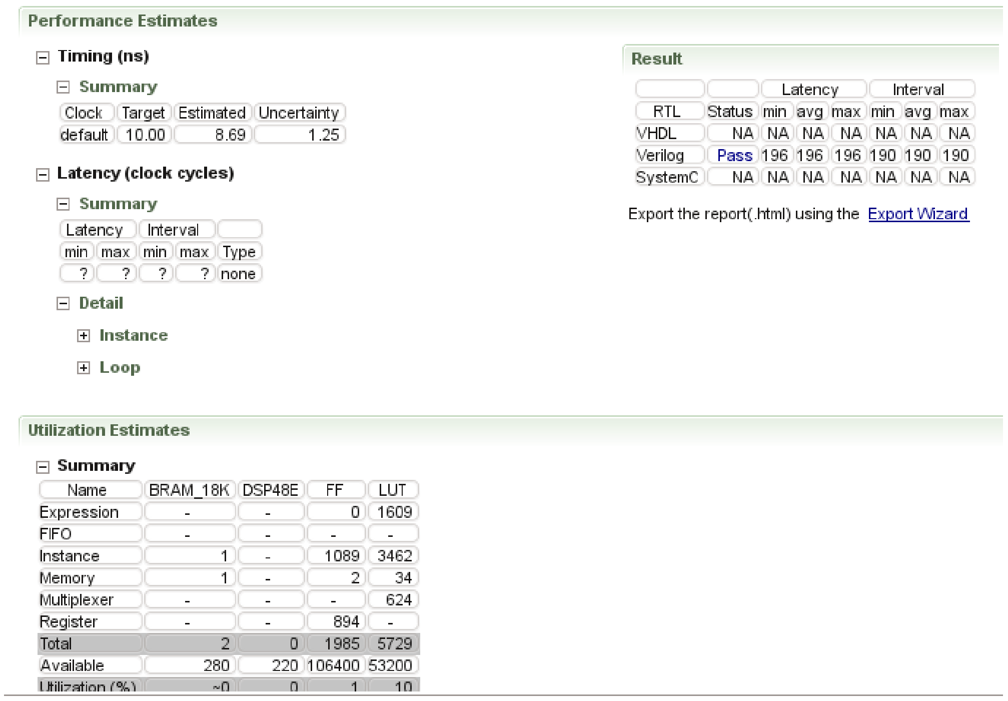
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	1609
FIFO	-	-	-	-
Instance	1	-	1089	3462
Memory	1	-	2	34
Multiplexer	-	-	-	624
Register	-	-	894	-
Total	2	0	1985	5729
Available	280	220	106400	53200
Utilization (%)	~0	0	1	10

Image 6.5 High Level Synthesis with loop unroll factor of 2

The circuit performance improved by 32 clock cycles at the expense of more than 200 flip-flops and 1200 LUTs. Again it's execution efficiency is less than half than the Verilog coded circuit. In the next step the loops will be unrolled by a factor of 4 to see if there is any significant improve upon execution duration. Inserting again directives for loop unroll with a factor of 4 on the heavy loops, the designed is again synthesized and cosimulated with the C source code producing the results shown in Image 6.6


Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.69	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

Loop

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	196	196	196	190	190	190
SystemC	NA	NA	NA	NA	NA	NA	NA

Export the report(.html) using the Export Wizard

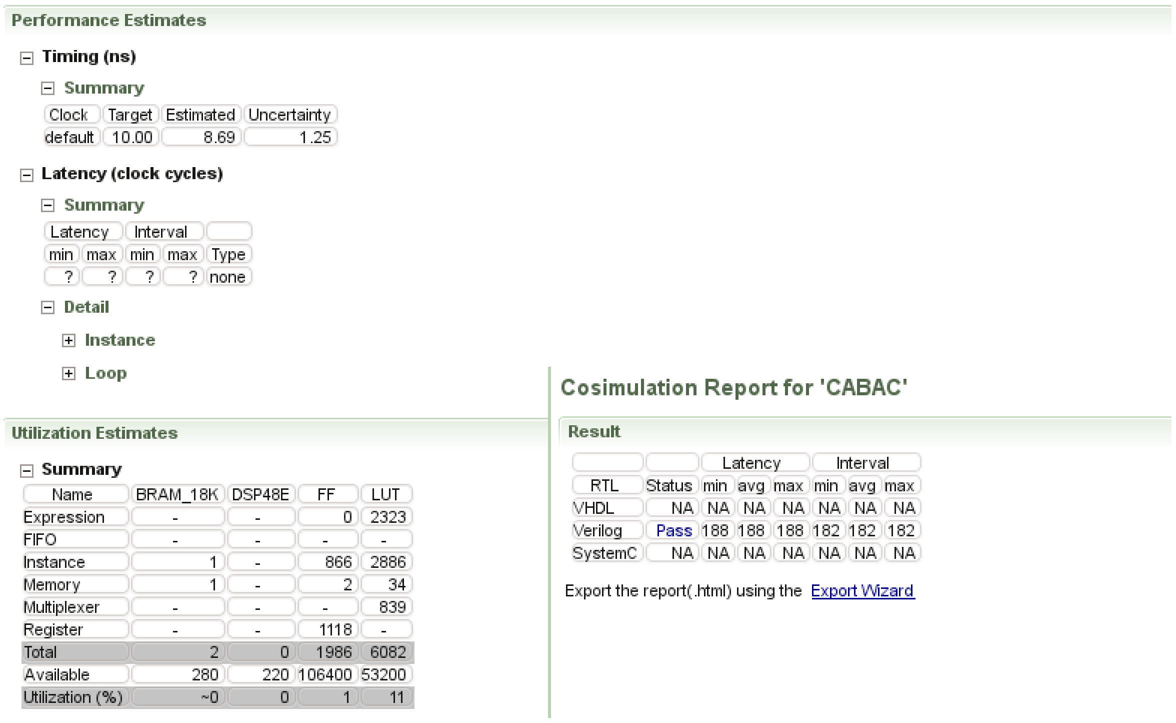
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	1609
FIFO	-	-	-	-
Instance	1	-	1089	3462
Memory	1	-	2	34
Multiplexer	-	-	-	624
Register	-	-	894	-
Total	2	0	1985	5729
Available	280	220	106400	53200
Utilization (%)	~0	0	1	10

Image 6.6 High Level Synthesis with loop unroll factor of 4

This directive actively brought the execution time down from 200 cycles without any added need for resources from the FPGA. However the improvement brought upon by this directive is minute. The need arises for another approach. This time the enhancement that will be utilized in the synthesized design is pipelining. By doing an analysis on the assembly code type execution, it is clear that the longest loop is the main one. It is the one that reads one symbol and enables the whole process of encoding, renormalizing and result outputting. This serial type of execution of this loop makes it the perfect candidate to be pipelined. So adding the pipeline directive to the existing unrolled loop, synthesizing and cosimulating it with the source code yields the results shown in Image 6.7.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.69	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	2323
FIFO	-	-	-	-
Instance	1	-	866	2886
Memory	1	-	2	34
Multiplexer	-	-	-	839
Register	-	-	1118	-
Total	2	0	1986	6082
Available	280	220	106400	53200
Utilization (%)	~0	0	1	11

Cosimulation Report for 'CABAC'

Result

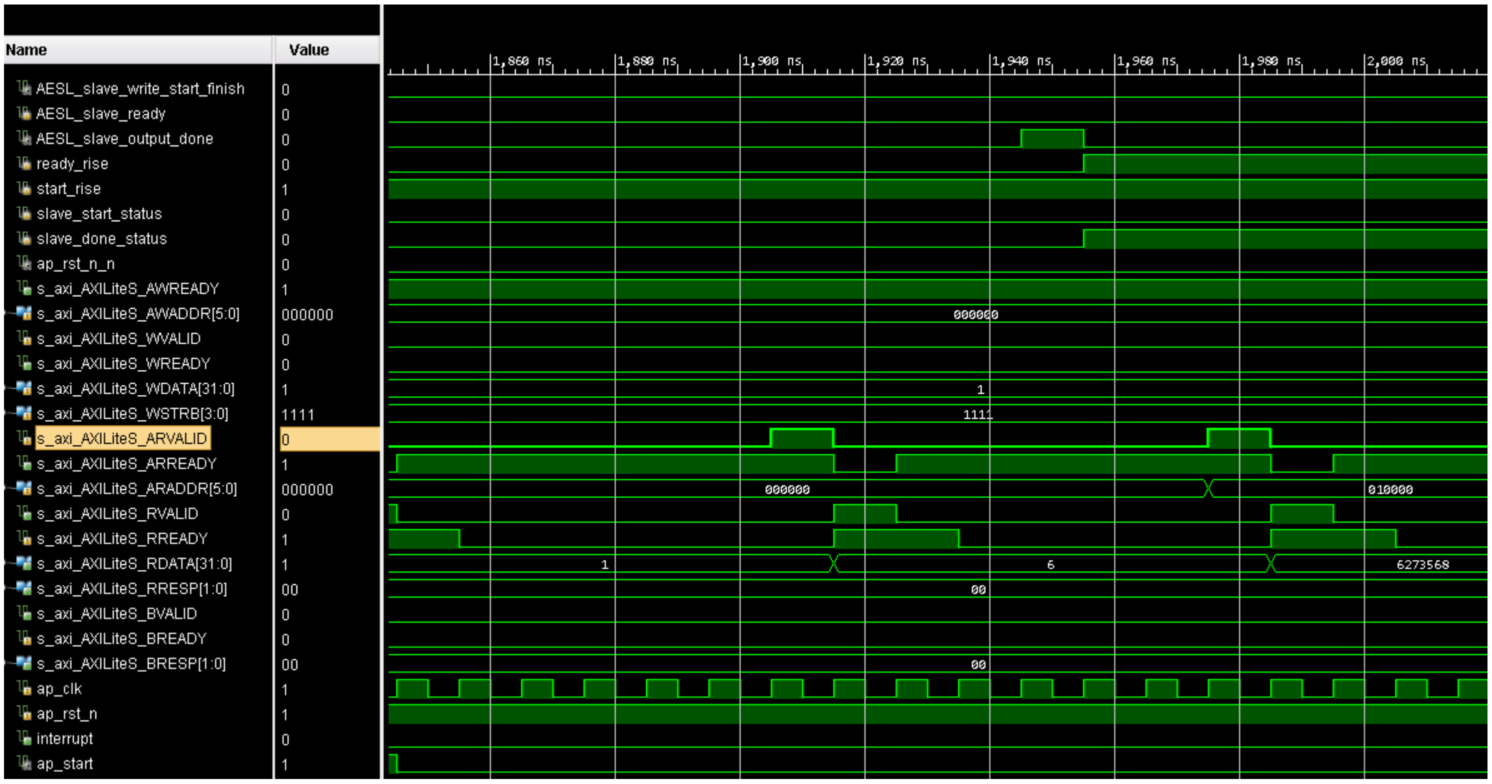
		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	188	188	188	182	182	182
SystemC	NA	NA	NA	NA	NA	NA	NA

Export the report(.html) using the Export Wizard

Image 6.7 High Level Synthesis with pipelining on the main loop

The gain once again is very small, just 8 cycles, although small is the increase in LUT and flip flop resources as well. Trying to make more loops pipelined proved ineffective, Although the gain in clock cycles was significant, there was a high increase in the estimated clock cycle period, almost doubling it. So with that improvement the was optimized as much as the HLS tool can offer. But even with these enhancements implemented, the Verilog circuit presented in the last chapter, remains almost two times faster and implemented at a fraction of the hardware resources. Instead of one fully optimized HLS-produced implementation, at least eight of the chapter 5 circuits can be deployed on the FPGA. The level of simplicity however provided by the HLS tool, makes it a fine replacement for a time consuming custom built Verilog circuit, especially one as small in area as this one.

The final stage of this project is to export the HLS synthesized circuit as a custom IP, with an AXI interface mediating I/O and connecting it to a Zynq processor. First the input and output of the circuit are defined by pragmas as AXIv4 interfaced ports. This step is accomplished by adding directives to the aforementioned ports specifying them as interfaced outputs and checking the AXIv4 option. The resulted synthesized circuit is exported as an independent IP on the project folder. Then, using the Vivado Design Suite, the simulated waveform is loaded to ensure the validity of the circuit's function. Image 5.8 illustrates that waveform



Name	Value
AESL_slave_write_start_finish	0
AESL_slave_ready	0
AESL_slave_output_done	0
ready_rise	0
start_rise	1
slave_start_status	0
slave_done_status	0
ap_rst_n_n	0
s_axi_AXILiteS_AWREADY	1
s_axi_AXILiteS_AWADDR[5:0]	000000
s_axi_AXILiteS_WVALID	0
s_axi_AXILiteS_WREADY	0
s_axi_AXILiteS_WDATA[31:0]	1
s_axi_AXILiteS_WSTRB[3:0]	1111
s_axi_AXILiteS_ARVALID	0
s_axi_AXILiteS_ARREADY	1
s_axi_AXILiteS_ARADDR[5:0]	000000
s_axi_AXILiteS_RVALID	0
s_axi_AXILiteS_RREADY	1
s_axi_AXILiteS_RDATA[31:0]	1
s_axi_AXILiteS_RRESP[1:0]	00
s_axi_AXILiteS_BVALID	0
s_axi_AXILiteS_BREADY	0
s_axi_AXILiteS_BRESP[1:0]	00
ap_clk	1
ap_rst_n	1
interrupt	0
ap_start	1

Image 6.8 Simulation waveform for the HLS synthesized IP with implemented AXI interface

It can be observed, that the produced output, stored in the s_axi_AXILiteS_RDATA[31:0] is identical to that of the C execution and the chapter 5 Verilog simulation. It is also worth noting the function of the AXI interface, where a clock cycle before the produced output an address is loaded on RADDR register followed by a high RREADY signal and the correct output.

6.3 SoC Production

The circuit is then added to the IP repository of the Vivado Design Suite, and added to a block design along with the instantiation of a Zynq processor, preset for the ZedBoard platform. An additional interrupt signal is added to the processor along with modules to facilitate the AXI protocol. Finally all of the modules are connected together, forming a unique SoC, the block diagramm of which can be seen in Image 6.9

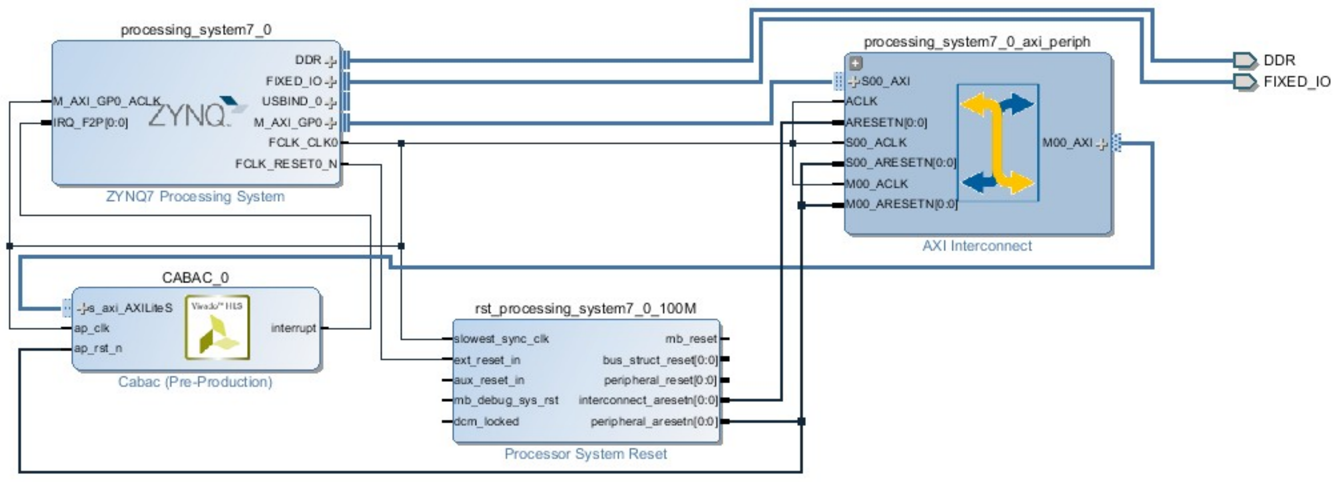
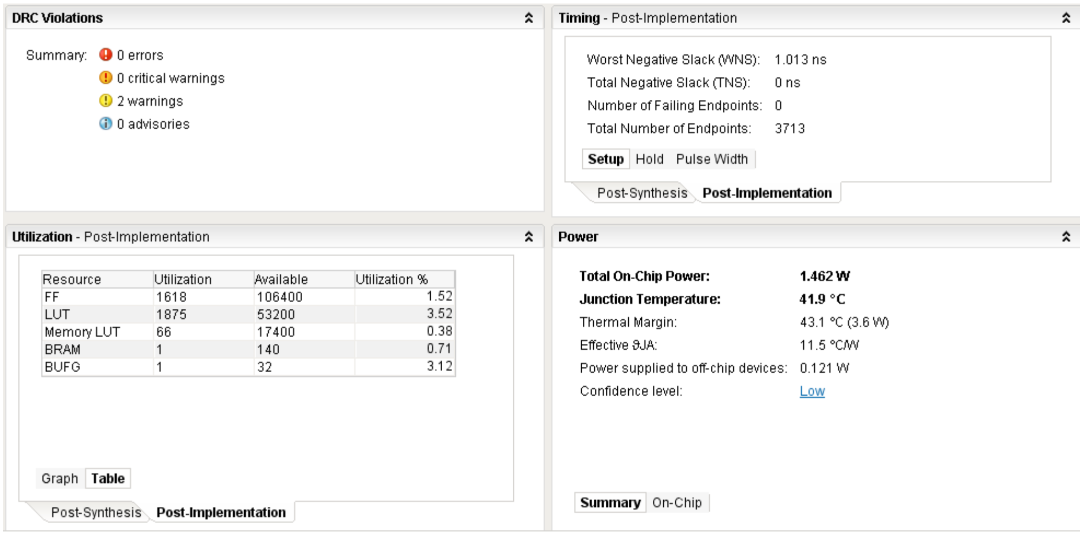


Image 6.9 The SoC consisting of a Zynq processor and the CABAC accelerator unit interfaced with AXI

Finally, an HDL wrap is produce to make this collection of modules just one, and it is verified, synthesized, implemented and producing the programmable bitstream to load on the ZedBoard. Images 6.10 and 6.11 illustrate the synthesized statistics and the implemented design respectively.


DRC Violations

Summary: 0 errors
0 critical warnings
2 warnings
0 advisories

Timing - Post-Implementation

Worst Negative Slack (WNS):	1.013 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	3713

Setup Hold Pulse Width

Post-Synthesis **Post-Implementation**

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
FF	1618	106400	1.52
LUT	1875	53200	3.52
Memory LUT	66	17400	0.38
BRAM	1	140	0.71
BUFG	1	32	3.12

Graph **Table**

Post-Synthesis **Post-Implementation**

Power

Total On-Chip Power:	**1.462 W**
Junction Temperature:	**41.9 °C**
Thermal Margin:	43.1 °C (3.6 W)
Effective θJA:	11.5 °C/W
Power supplied to off-chip devices:	0.121 W
Confidence level:	Low

Summary On-Chip

Image 6.9 Synthesized SoC implementation measurements

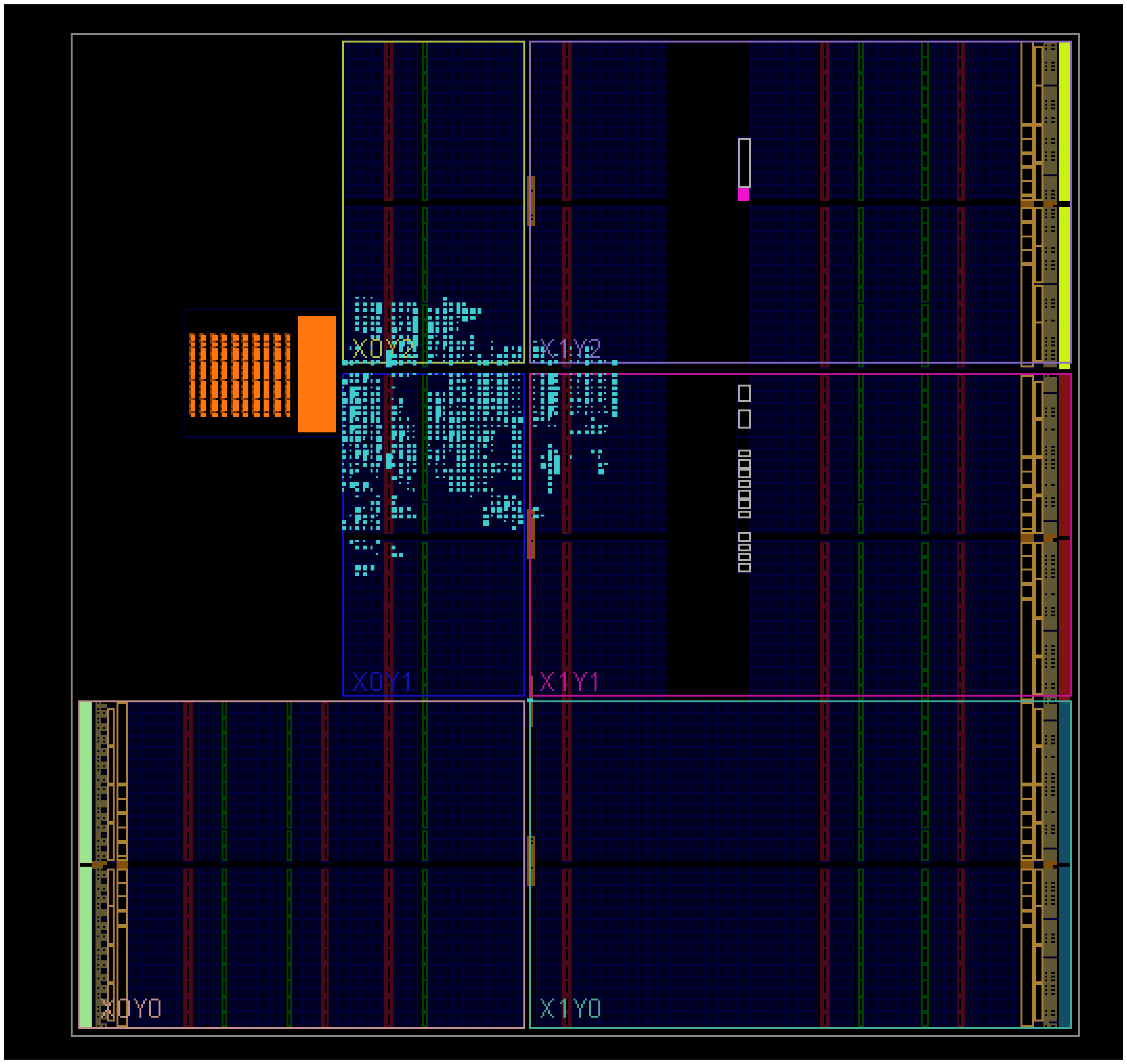


Image 6.10 Synthesized SoC implementation design

7. Closing Remarks / Future

As it was shown during the previous two chapters, the hardware implementation of the CABAC algorithm proves to be highly efficient, with approximately 100ns real time execution of encoding a 11 bit string. Truth be told, the circuit can be improved upon with certain techniques. Pipeline registers for instance, can be utilized between each module, to effectively load at least 4 inputs at the same time, without great increases in resources need (such as FFTs or LUTs).

In the SoC implementation, more than one accelerators can be connected to the same processor, to achieve a level of pseudo parallelism, loading for example ten different bins on ten distinct modules to be encoded at the same time. Also, a software code could be created and loaded on the SoC to activate and use the accelerators, which compared to the source code executed on a similar machine, could provide a more hands-on comparison between hardware and software.

HLS though not so efficient as the implementations showcased, is an excellent tool to produce a hardware module, validate it and synthesize it with the push of a button. Such ease of use more than counterbalance the inefficiencies of the produced circuit, inefficiencies compared of course to a similar custom made Verilog module that could require weeks or even months to develop.

A thought provoking concept is also the development of a complementary decoding circuit, mirroring the encoding process and using the same efficient techniques (such as LUTs instead of division/multiplication). The modules could be paired and help a computer peripheral handle or transmit faster large vectors, without any losses whatsoever by quickly and efficiently encoding and decoding said vector.

Finally, although it is highly unlike to be achieved due to the serial nature of the algorithm, an effort could be made to truly parallelize the code, that is to succeed in encoding more than one symbol in the same time. The need to update many values and use them as feed for the next iteration all but forbid said hypothetical improvement, although some things, unthinkable otherwise, may be achieved just by looking at them form a different angle.

REFERENCES

[1] Youn-Long Steve Lin, ChaoYang, Kao, Jianwen Chen, and HuangChih, Kuo. *VLSI Design for Video Coding*. Springer, 2010.

[2] Salauddin Mahmud. An improved data compression method for general data. *International Journal of Scientific & Engineering Research*, 3(3):2.

[3] Christos P. Sotiriou. Implementation of bac/cabac algorithm.

[4] Xiaohua Tian, M Le Thinh, and Yong Lian. *Entropy Coders of the H. 264/AVC Standard*. Springer, 2011.

[5] Peter Wayner. *Compression algorithms for real programmers*. Morgan Kaufmann, 1999.

[6] Ian H Witten, RadfordMNeal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[7] Nikolaos Sketopoulos, Implementation analysis of the algorithm CABAC in the H.264 standards

[8] University of Thessaly, CE435 Embedded Systems Course website, http://inf-server.inf.uth.gr/courses/CE43

[9] Vivado HLS Tutorial doc,

http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug871-vivado-high-level-synthesis-tutorial.pdf

[10] Vivado HLS User Guide

http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[11] ZedBoard User Guide http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf

[12] ZedBoard Board Schematic

http://zedboard.com/misc/files/ZedBoard_RevC.1_Schematic_preliminary.pdf

[13] Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard Detlev Marpe, Member, IEEE, Heiko Schwarz, and Thomas Wiegand

[14] Yao-Chang Yang, Chien-Chang Lin, Hsui-Cheng Chang, Ching-Lung Su, and JiunIn Guo, A high throughput VLSI architecture design for H.264 context-based adaptive binary arithmetic decoding with look ahead parsing, ICME, IEEE, 2006, pp. 357–360.

[15] Y. S. Yi and I. C. Park, High-speed H.264/AVC CABAC decoding, IEEE Trans. Circuits and Systems for Video Technology 17 (2007), no. 4, 490–494.

[16] Wei Yu and Yun He, A high performance cabac decoding architecture, Consumer Electronics, IEEE Transactions on 51 (2005), no. 4, 1352–1359.

[17] Peng Zhang, Wen Gao, Don Xie, and Di Wu, High-performance cabac engine for h.264/avc high definition real-time decoding, Consumer Electronics, 2007. ICCE 2007. Digest of Technical Papers. International Conference on (2007), 1–2.

[18] Junhao Zheng, David Wu, Don Xie, and Wen Gao, A novel pipeline design for H.264 CABAC decoding, Advances in Multimedia Information Processing - PCM 2007, 8th Pacific Rim Conference on Multimedia, Hong Kong, China, December 11- 14, 2007, Proceedings (Horace Ho-Shing Ip, Oscar C. Au, Howard Leung, Ming-Ting Sun, Wei-Ying Ma, and Shi-Min Hu, eds.), Lecture Notes in Computer Science, vol. 4810, Springer, 2007, pp. 559–568.

[19] G. Pastuszak, A high-performance architecture of the double-mode binary coder for H.264.AVC, IEEE Trans. Circuits and Systems for Video Technology 18 (2008), no. 7, 949–960.

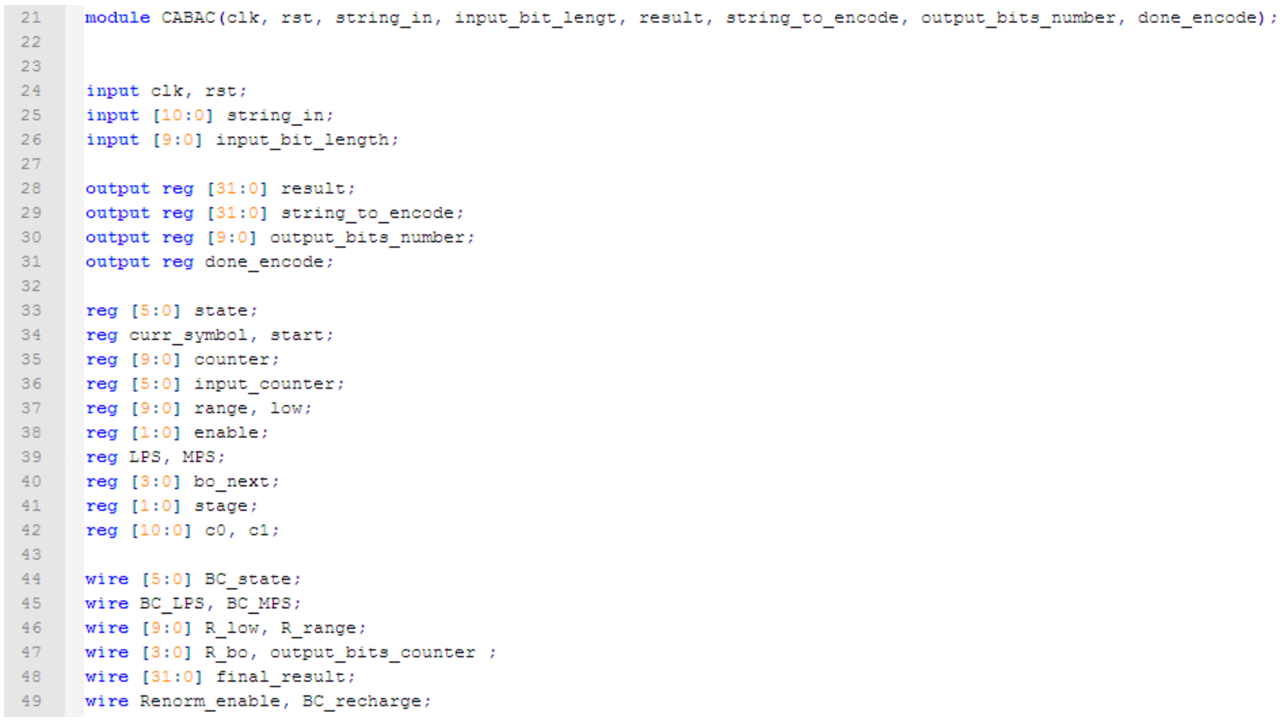
[20] Iain E. Richardson, H.264 and mpeg-4 video compression: Video coding for next generation multimedia, 1 ed., Wiley, August 2003.

[21] Sergio Saponara, Carolina Blanch, Kristof Denolf, and Jan Bormans, The jvt advanced video coding standard: Complexity and performance analysis on a tool-by-tool basis, unknown journal name (2003), –.

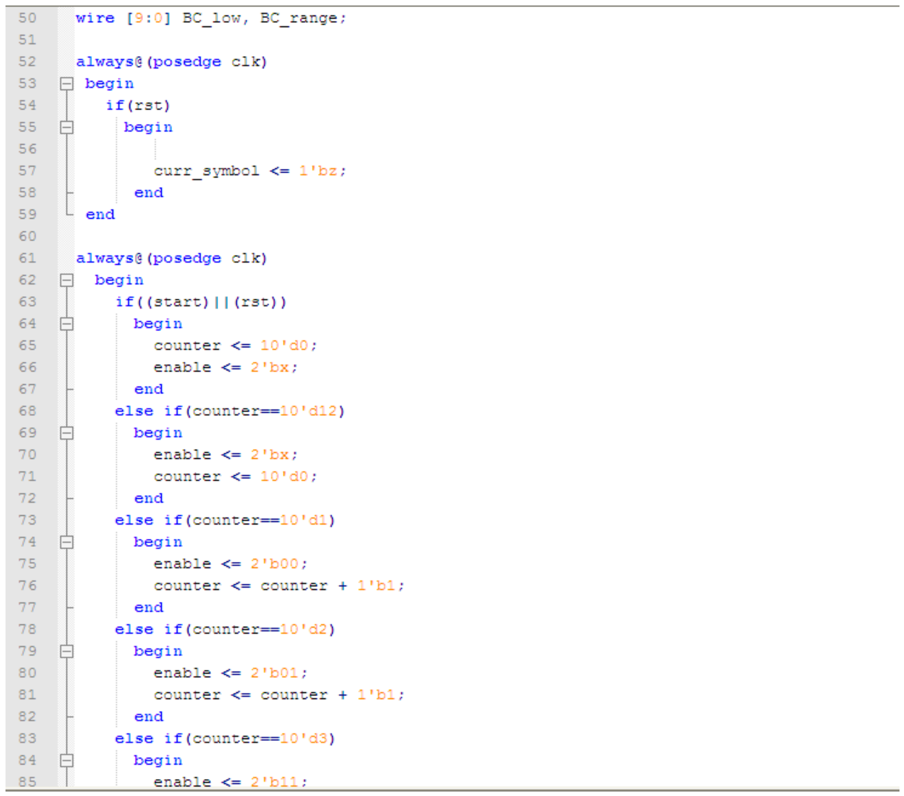
APPENDIX A

Verilog Source Code

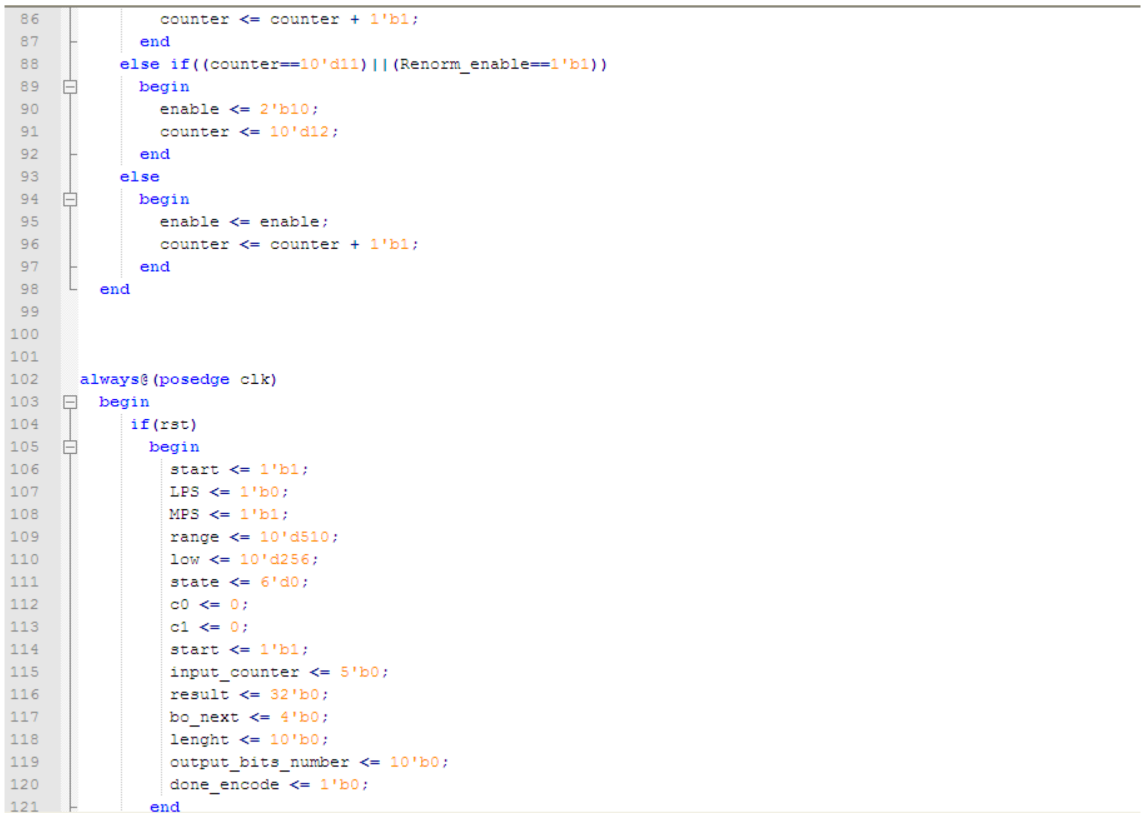
Main module CABAC



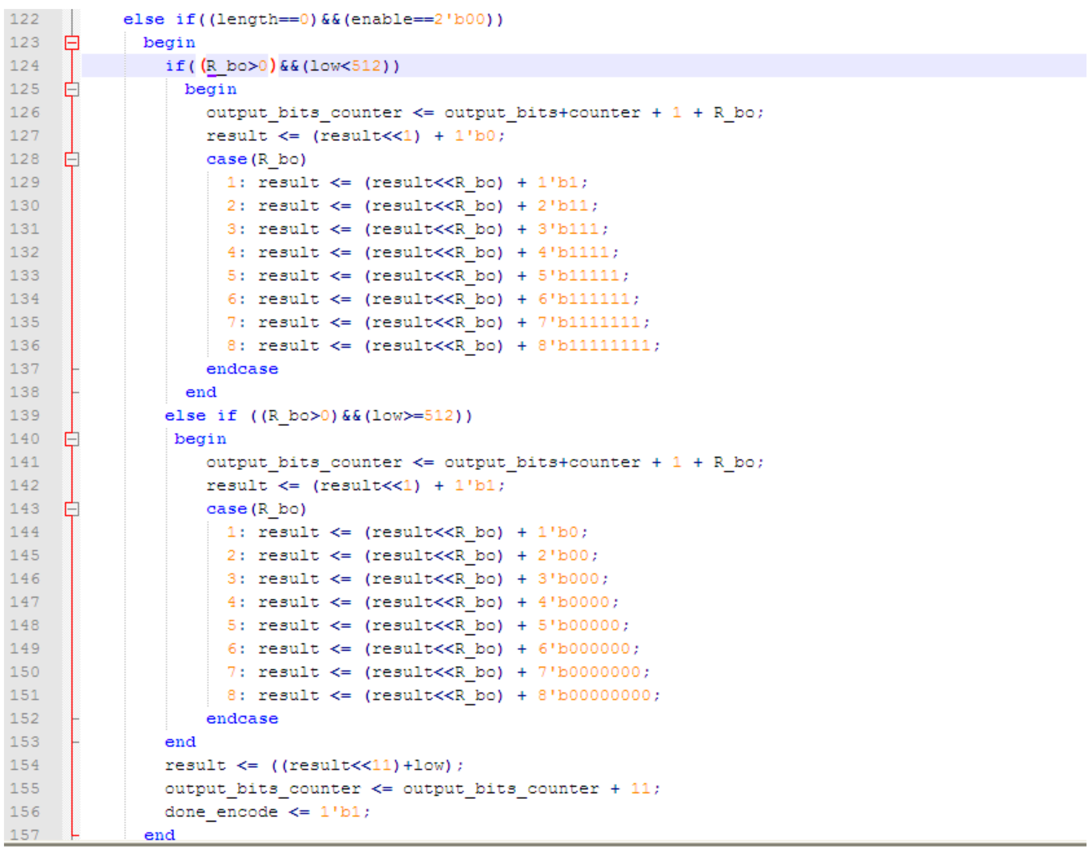
```
module CABAC(clk, rst, string_in, input_bit_lengt, result, string_to_encode, output_bits_number, done_encode);


input clk, rst;
input [10:0] string_in;
input [9:0] input_bit_length;

output reg [31:0] result;
output reg [31:0] string_to_encode;
output reg [9:0] output_bits_number;
output reg done_encode;

reg [5:0] state;
reg curr_symbol, start;
reg [9:0] counter;
reg [5:0] input_counter;
reg [9:0] range, low;
reg [1:0] enable;
reg LPS, MPS;
reg [3:0] bo_next;
reg [1:0] stage;
reg [10:0] c0, c1;

wire [5:0] BC_state;
wire BC_LPS, BC_MPS;
wire [9:0] R_low, R_range;
wire [3:0] R_bo, output_bits_counter ;
wire [31:0] final_result;
wire Renorm_enable, BC_recharge;
```



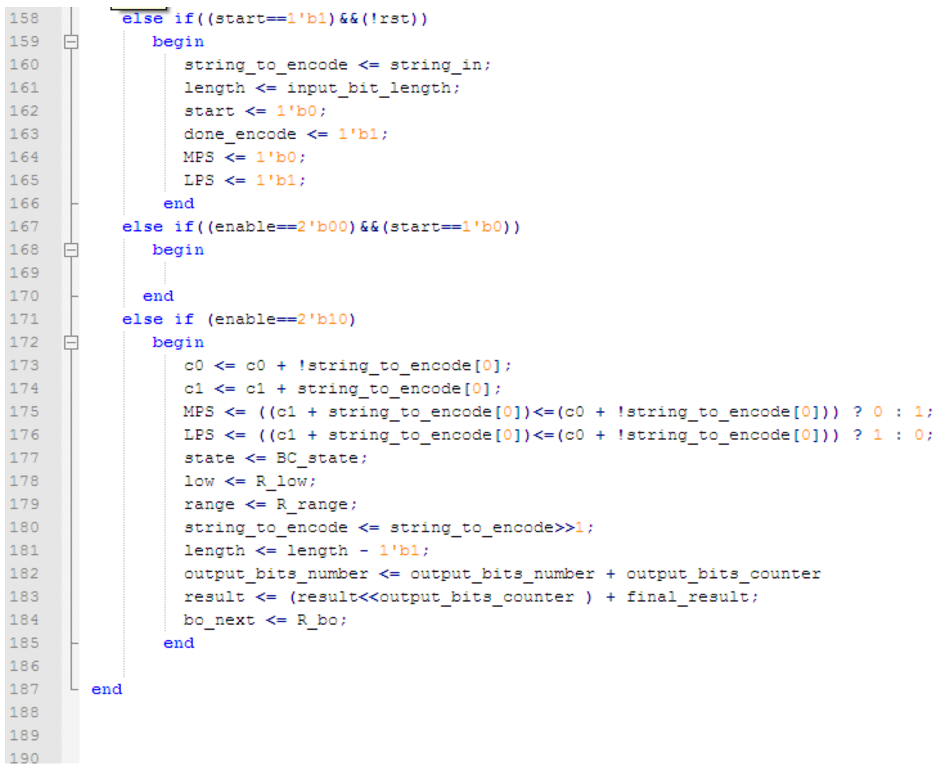
```
wire [9:0] BC_low, BC_range;

always@(posedge clk)
begin
  if(rst)
   begin

      curr_symbol <= 1'bz;
    end
end

always@(posedge clk)
 begin
   if((start)||(rst))
     begin
       counter <= 10'd0;
       enable <= 2'bx;
     end
   else if(counter==10'd12)
     begin
       enable <= 2'bx;
       counter <= 10'd0;
     end
   else if(counter==10'd1)
     begin
       enable <= 2'b00;
       counter <= counter + 1'b1;
     end
   else if(counter==10'd2)
     begin
       enable <= 2'b01;
       counter <= counter + 1'b1;
     end
   else if(counter==10'd3)
     begin
       enable <= 2'b11;
```



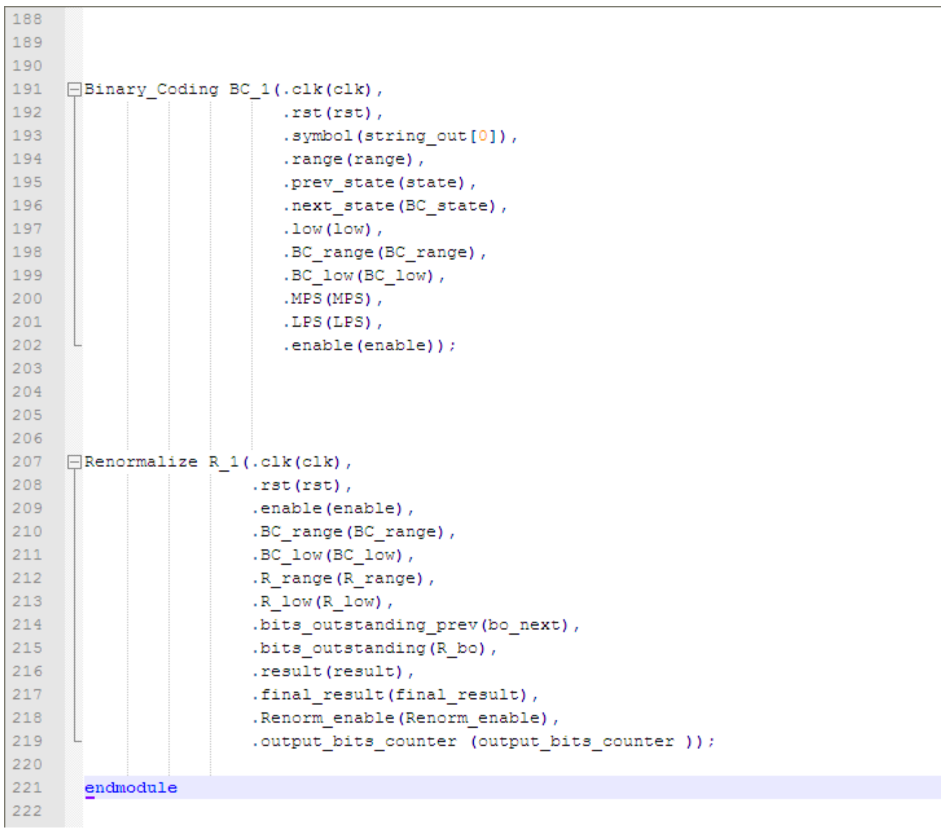
```
        counter <= counter + 1'b1;
      end
    else if((counter==10'd11)||(Renorm_enable==1'b1))
      begin
        enable <= 2'b10;
        counter <= 10'd12;
      end
    else
      begin
        enable <= enable;
        counter <= counter + 1'b1;
      end
  end



always@(posedge clk)
  begin
     if(rst)
       begin
         start <= 1'b1;
         LPS <= 1'b0;
         MPS <= 1'b1;
         range <= 10'd510;
         low <= 10'd256;
         state <= 6'd0;
         c0 <= 0;
         c1 <= 0;
         start <= 1'b1;
         input_counter <= 5'b0;
         result <= 32'b0;
         bo_next <= 4'b0;
         lenght <= 10'b0;
         output_bits_number <= 10'b0;
         done_encode <= 1'b0;
       end
```



```
     else if((length==0)&&(enable==2'b00))
       begin
         if((R_bo>0)&&(low<512))
           begin
             output_bits_counter <= output_bits+counter + 1 + R_bo;
             result <= (result<<1) + 1'b0;
             case(R_bo)
               1: result <= (result<<R_bo) + 1'b1;
               2: result <= (result<<R_bo) + 2'b11;
               3: result <= (result<<R_bo) + 3'b111;
               4: result <= (result<<R_bo) + 4'b1111;
               5: result <= (result<<R_bo) + 5'b11111;
               6: result <= (result<<R_bo) + 6'b111111;
               7: result <= (result<<R_bo) + 7'b1111111;
               8: result <= (result<<R_bo) + 8'b11111111;
             endcase
           end
         else if ((R_bo>0)&&(low>=512))
          begin
             output_bits_counter <= output_bits+counter + 1 + R_bo;
             result <= (result<<1) + 1'b1;
             case(R_bo)
               1: result <= (result<<R_bo) + 1'b0;
               2: result <= (result<<R_bo) + 2'b00;
               3: result <= (result<<R_bo) + 3'b000;
               4: result <= (result<<R_bo) + 4'b0000;
               5: result <= (result<<R_bo) + 5'b00000;
               6: result <= (result<<R_bo) + 6'b000000;
               7: result <= (result<<R_bo) + 7'b0000000;
               8: result <= (result<<R_bo) + 8'b00000000;
             endcase
         end
         result <= ((result<<11)+low);
         output_bits_counter <= output_bits_counter + 11;
         done_encode <= 1'b1;
       end
```

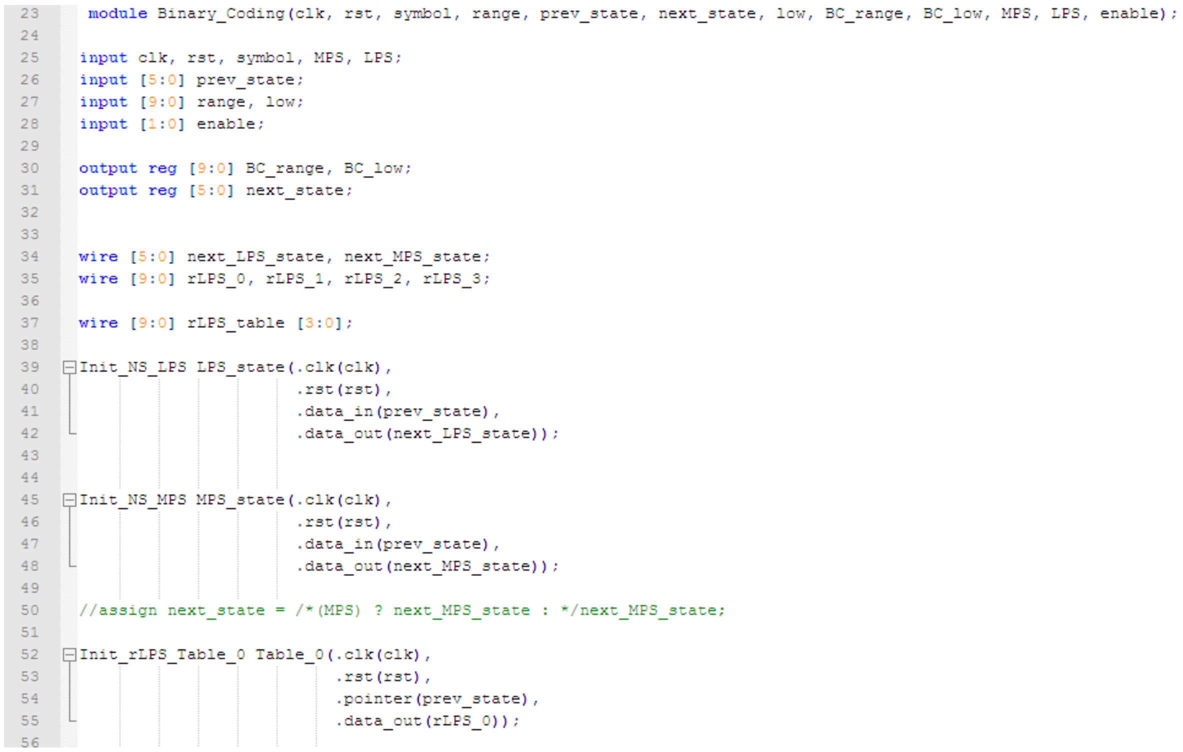


```
    else if((start==1'b1)&&(!rst))
       begin
          string_to_encode <= string_in;
          length <= input_bit_length;
          start <= 1'b0;
          done_encode <= 1'b1;
          MPS <= 1'b0;
          LPS <= 1'b1;
        end
    else if((enable==2'b00)&&(start==1'b0))
       begin

      end
    else if (enable==2'b10)
       begin
          c0 <= c0 + !string_to_encode[0];
          c1 <= c1 + string_to_encode[0];
          MPS <= ((c1 + string_to_encode[0])<=(c0 + !string_to_encode[0])) ? 0 : 1;
          LPS <= ((c1 + string_to_encode[0])<=(c0 + !string_to_encode[0])) ? 1 : 0;
          state <= BC_state;
          low <= R_low;
          range <= R_range;
          string_to_encode <= string_to_encode>>1;
          length <= length - 1'b1;
          output_bits_number <= output_bits_number + output_bits_counter
          result <= (result<<output_bits_counter ) + final_result;
          bo_next <= R_bo;
        end

 end
```

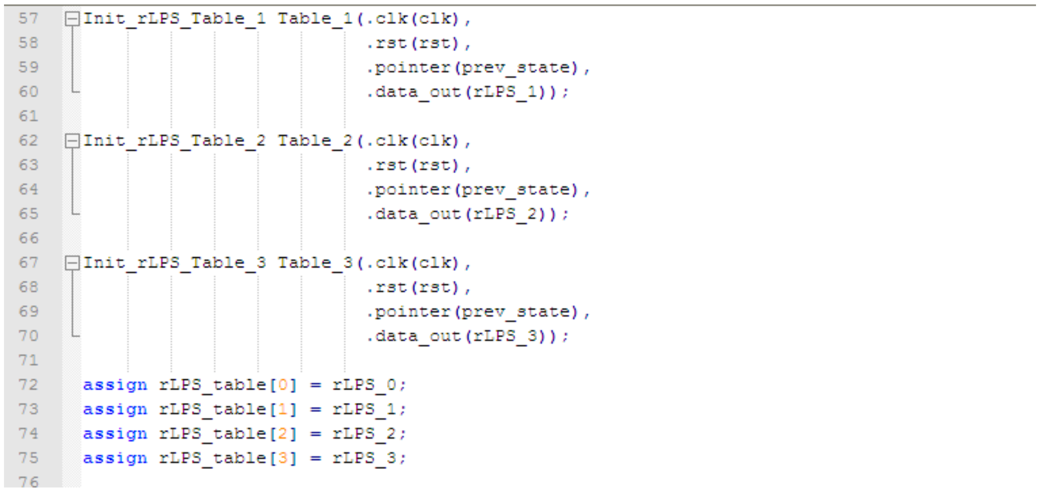


```
Binary_Coding BC_1(.clk(clk),
                   .rst(rst),
                   .symbol(string_out[0]),
                   .range(range),
                   .prev_state(state),
                   .next_state(BC_state),
                   .low(low),
                   .BC_range(BC_range),
                   .BC_low(BC_low),
                   .MPS(MPS),
                   .LPS(LPS),
                   .enable(enable));




Renormalize R_1(.clk(clk),
                .rst(rst),
                .enable(enable),
                .BC_range(BC_range),
                .BC_low(BC_low),
                .R_range(R_range),
                .R_low(R_low),
                .bits_outstanding_prev(bo_next),
                .bits_outstanding(R_bo),
                .result(result),
                .final_result(final_result),
                .Renorm_enable(Renorm_enable),
                .output_bits_counter (output_bits_counter ));

endmodule
```

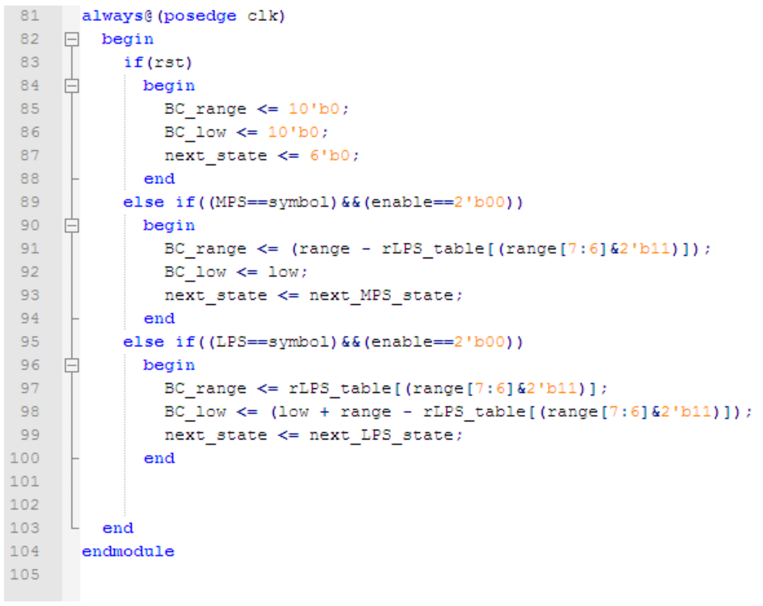
Module Binary Coding



```verilog
 module Binary_Coding(clk, rst, symbol, range, prev_state, next_state, low, BC_range, BC_low, MPS, LPS, enable);

input clk, rst, symbol, MPS, LPS;
input [5:0] prev_state;
input [9:0] range, low;
input [1:0] enable;

output reg [9:0] BC_range, BC_low;
output reg [5:0] next_state;


wire [5:0] next_LPS_state, next_MPS_state;
wire [9:0] rLPS_0, rLPS_1, rLPS_2, rLPS_3;

wire [9:0] rLPS_table [3:0];

Init_NS_LPS LPS_state(.clk(clk),
                      .rst(rst),
                      .data_in(prev_state),
                      .data_out(next_LPS_state));


Init_NS_MPS MPS_state(.clk(clk),
                      .rst(rst),
                      .data_in(prev_state),
                      .data_out(next_MPS_state));

//assign next_state = /*(MPS) ? next_MPS_state : */next_MPS_state;

Init_rLPS_Table_0 Table_0(.clk(clk),
                          .rst(rst),
                          .pointer(prev_state),
                          .data_out(rLPS_0));

```

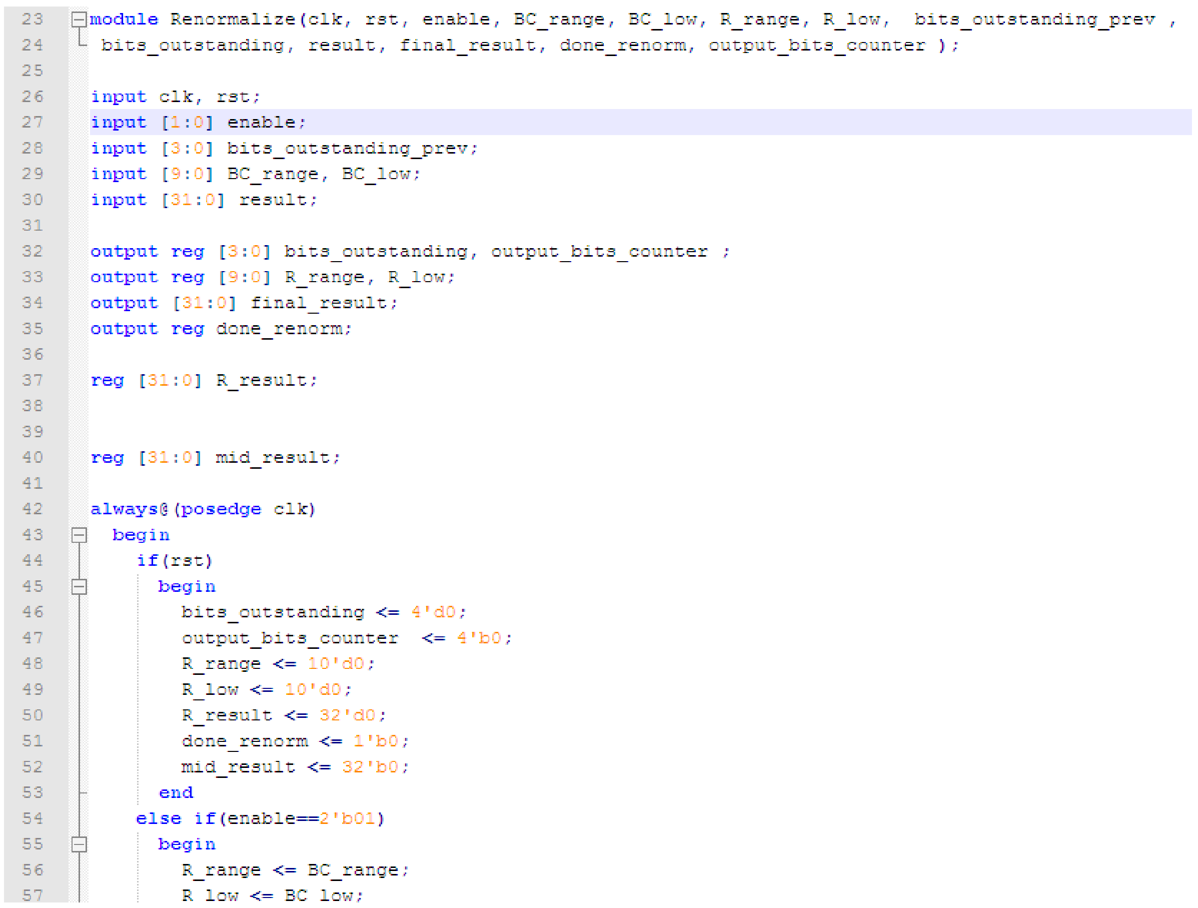


```verilog
Init_rLPS_Table_1 Table_1(.clk(clk),
                          .rst(rst),
                          .pointer(prev_state),
                          .data_out(rLPS_1));

Init_rLPS_Table_2 Table_2(.clk(clk),
                          .rst(rst),
                          .pointer(prev_state),
                          .data_out(rLPS_2));

Init_rLPS_Table_3 Table_3(.clk(clk),
                          .rst(rst),
                          .pointer(prev_state),
                          .data_out(rLPS_3));

assign rLPS_table[0] = rLPS_0;
assign rLPS_table[1] = rLPS_1;
assign rLPS_table[2] = rLPS_2;
assign rLPS_table[3] = rLPS_3;

```

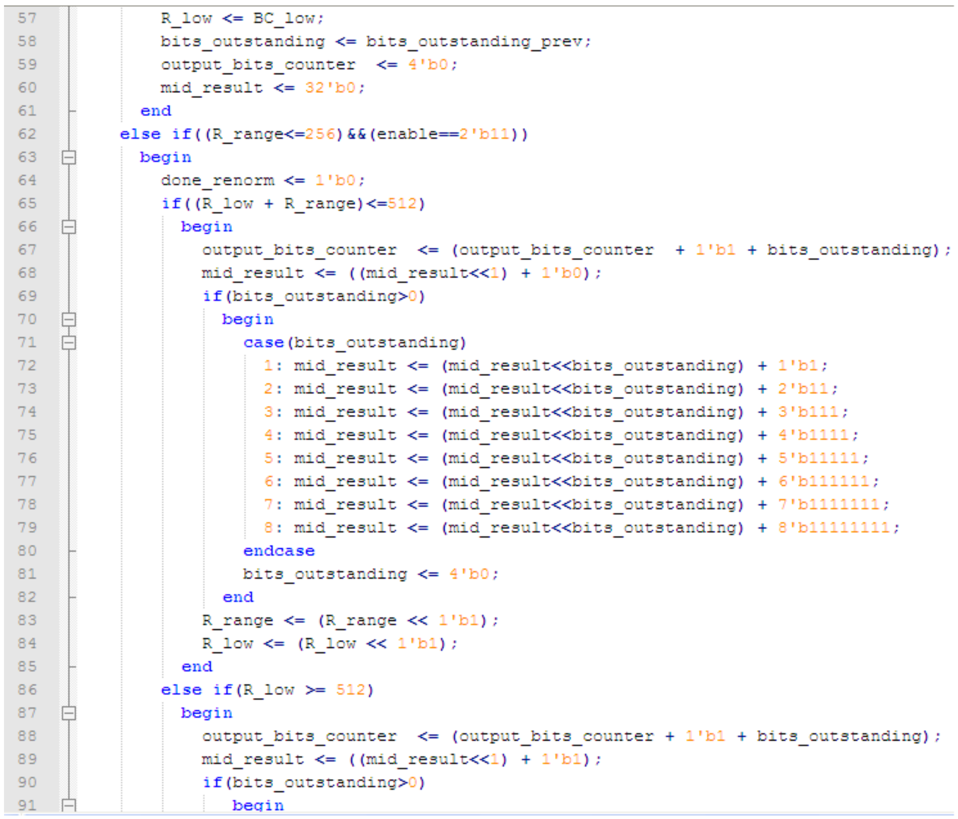


```
always@(posedge clk)
  begin
    if(rst)
      begin
        BC_range <= 10'b0;
        BC_low <= 10'b0;
        next_state <= 6'b0;
      end
    else if((MPS==symbol)&&(enable==2'b00))
      begin
        BC_range <= (range - rLPS_table[(range[7:6]&2'b11)]);
        BC_low <= low;
        next_state <= next_MPS_state;
      end
    else if((LPS==symbol)&&(enable==2'b00))
      begin
        BC_range <= rLPS_table[(range[7:6]&2'b11)];
        BC_low <= (low + range - rLPS_table[(range[7:6]&2'b11)]);
        next_state <= next_LPS_state;
      end


  end
endmodule

```

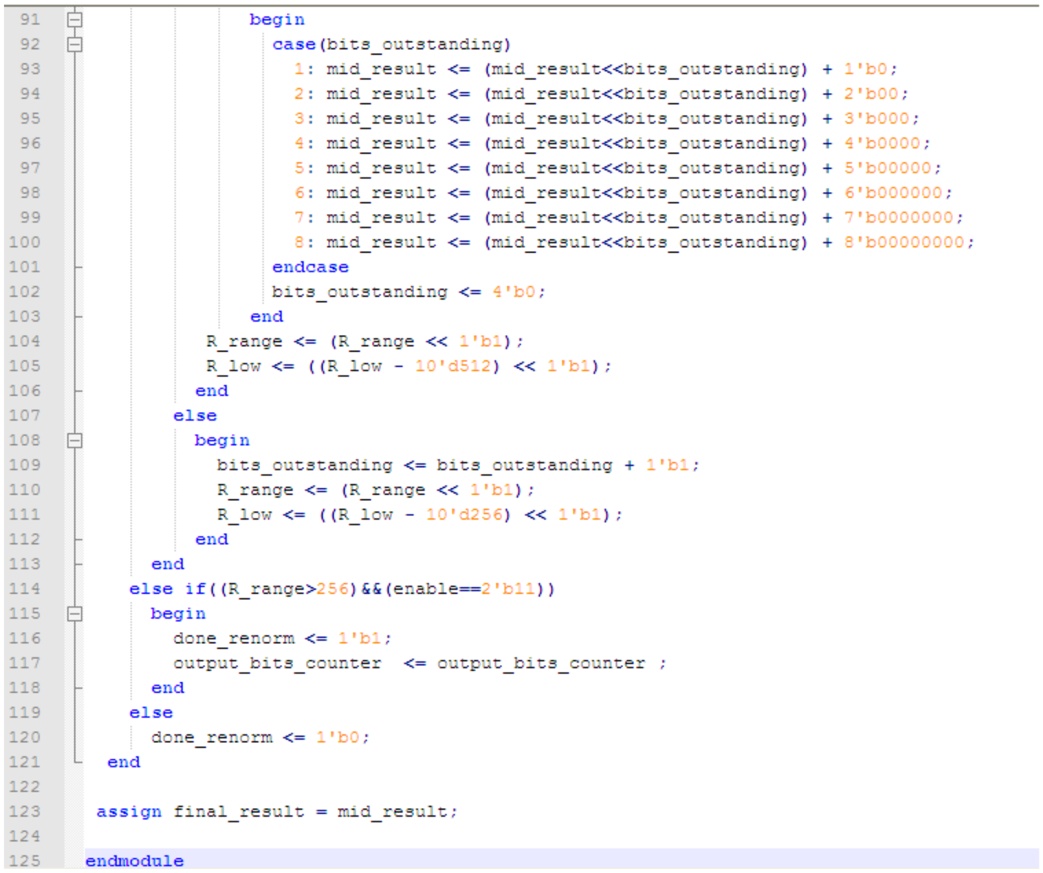
Module Renormalize



```
module Renormalize(clk, rst, enable, BC_range, BC_low, R_range, R_low,  bits_outstanding_prev ,
 bits_outstanding, result, final_result, done_renorm, output_bits_counter );

input clk, rst;
input [1:0] enable;
input [3:0] bits_outstanding_prev;
input [9:0] BC_range, BC_low;
input [31:0] result;

output reg [3:0] bits_outstanding, output_bits_counter ;
output reg [9:0] R_range, R_low;
output [31:0] final_result;
output reg done_renorm;

reg [31:0] R_result;


reg [31:0] mid_result;

always@(posedge clk)
  begin
    if(rst)
      begin
        bits_outstanding <= 4'd0;
        output_bits_counter  <= 4'b0;
        R_range <= 10'd0;
        R_low <= 10'd0;
        R_result <= 32'd0;
        done_renorm <= 1'b0;
        mid_result <= 32'b0;
      end
    else if(enable==2'b01)
      begin
        R_range <= BC_range;
        R_low <= BC_low;
```

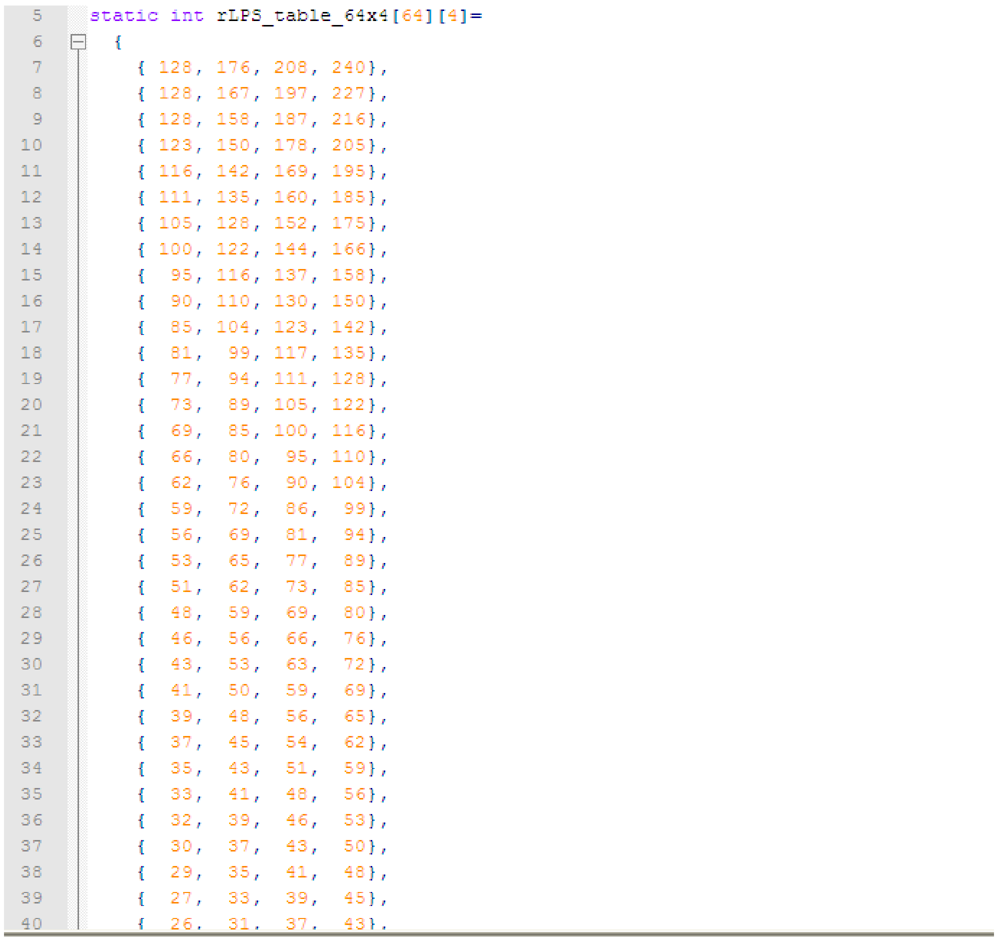


```
        R_low <= BC_low;
        bits_outstanding <= bits_outstanding_prev;
        output_bits_counter  <= 4'b0;
        mid_result <= 32'b0;
      end
    else if((R_range<=256)&&(enable==2'b11))
      begin
        done_renorm <= 1'b0;
        if((R_low + R_range)<=512)
          begin
            output_bits_counter  <= (output_bits_counter  + 1'b1 + bits_outstanding);
            mid_result <= ((mid_result<<1) + 1'b0);
            if(bits_outstanding>0)
              begin
                case(bits_outstanding)
                  1: mid_result <= (mid_result<<bits_outstanding) + 1'b1;
                  2: mid_result <= (mid_result<<bits_outstanding) + 2'b11;
                  3: mid_result <= (mid_result<<bits_outstanding) + 3'b111;
                  4: mid_result <= (mid_result<<bits_outstanding) + 4'b1111;
                  5: mid_result <= (mid_result<<bits_outstanding) + 5'b11111;
                  6: mid_result <= (mid_result<<bits_outstanding) + 6'b111111;
                  7: mid_result <= (mid_result<<bits_outstanding) + 7'b1111111;
                  8: mid_result <= (mid_result<<bits_outstanding) + 8'b11111111;
                endcase
                bits_outstanding <= 4'b0;
              end
            R_range <= (R_range << 1'b1);
            R_low <= (R_low << 1'b1);
          end
        else if(R_low >= 512)
          begin
            output_bits_counter  <= (output_bits_counter + 1'b1 + bits_outstanding);
            mid_result <= ((mid_result<<1) + 1'b1);
            if(bits_outstanding>0)
```

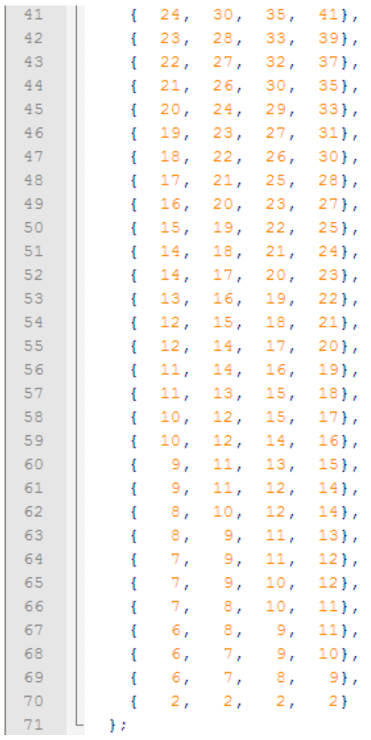


```
              begin
                case(bits_outstanding)
                  1: mid_result <= (mid_result<<bits_outstanding) + 1'b0;
                  2: mid_result <= (mid_result<<bits_outstanding) + 2'b00;
                  3: mid_result <= (mid_result<<bits_outstanding) + 3'b000;
                  4: mid_result <= (mid_result<<bits_outstanding) + 4'b0000;
                  5: mid_result <= (mid_result<<bits_outstanding) + 5'b00000;
                  6: mid_result <= (mid_result<<bits_outstanding) + 6'b000000;
                  7: mid_result <= (mid_result<<bits_outstanding) + 7'b0000000;
                  8: mid_result <= (mid_result<<bits_outstanding) + 8'b00000000;
                endcase
                bits_outstanding <= 4'b0;
              end
            R_range <= (R_range << 1'b1);
            R_low <= ((R_low - 10'd512) << 1'b1);
          end
        else
          begin
            bits_outstanding <= bits_outstanding + 1'b1;
            R_range <= (R_range << 1'b1);
            R_low <= ((R_low - 10'd256) << 1'b1);
          end
      end
    else if((R_range>256)&&(enable==2'b11))
      begin
        done_renorm <= 1'b1;
        output_bits_counter  <= output_bits_counter ;
      end
    else
      done_renorm <= 1'b0;
  end

 assign final_result = mid_result;

endmodule
```

rLPS LookUp Table

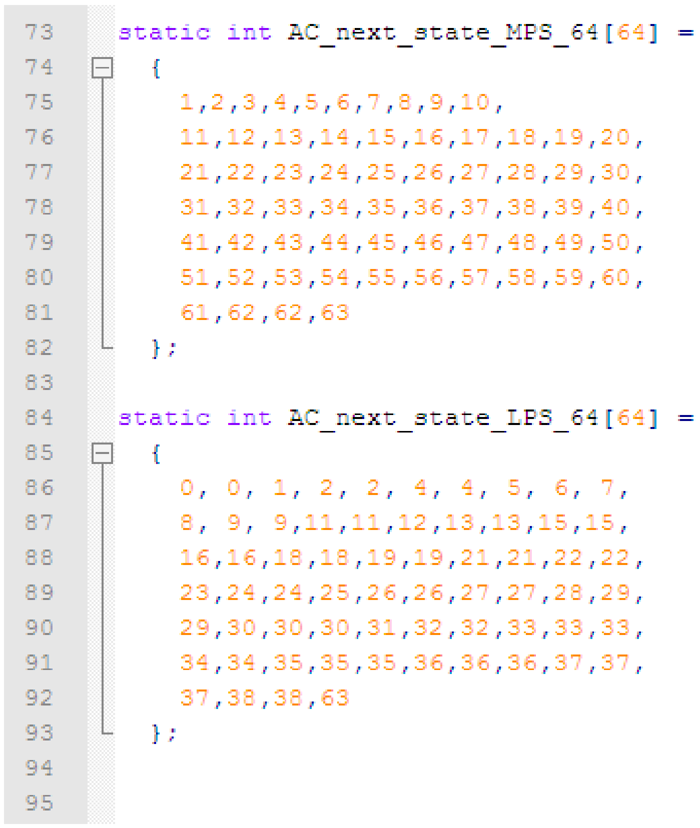


```
static int rLPS_table_64x4[64][4]=
 {
   { 128, 176, 208, 240},
   { 128, 167, 197, 227},
   { 128, 158, 187, 216},
   { 123, 150, 178, 205},
   { 116, 142, 169, 195},
   { 111, 135, 160, 185},
   { 105, 128, 152, 175},
   { 100, 122, 144, 166},
   {  95, 116, 137, 158},
   {  90, 110, 130, 150},
   {  85, 104, 123, 142},
   {  81,  99, 117, 135},
   {  77,  94, 111, 128},
   {  73,  89, 105, 122},
   {  69,  85, 100, 116},
   {  66,  80,  95, 110},
   {  62,  76,  90, 104},
   {  59,  72,  86,  99},
   {  56,  69,  81,  94},
   {  53,  65,  77,  89},
   {  51,  62,  73,  85},
   {  48,  59,  69,  80},
   {  46,  56,  66,  76},
   {  43,  53,  63,  72},
   {  41,  50,  59,  69},
   {  39,  48,  56,  65},
   {  37,  45,  54,  62},
   {  35,  43,  51,  59},
   {  33,  41,  48,  56},
   {  32,  39,  46,  53},
   {  30,  37,  43,  50},
   {  29,  35,  41,  48},
   {  27,  33,  39,  45},
   {  26,  31,  37,  43},
```



```
   {  24,  30,  35,  41},
   {  23,  28,  33,  39},
   {  22,  27,  32,  37},
   {  21,  26,  30,  35},
   {  20,  24,  29,  33},
   {  19,  23,  27,  31},
   {  18,  22,  26,  30},
   {  17,  21,  25,  28},
   {  16,  20,  23,  27},
   {  15,  19,  22,  25},
   {  14,  18,  21,  24},
   {  14,  17,  20,  23},
   {  13,  16,  19,  22},
   {  12,  15,  18,  21},
   {  12,  14,  17,  20},
   {  11,  14,  16,  19},
   {  11,  13,  15,  18},
   {  10,  12,  15,  17},
   {  10,  12,  14,  16},
   {   9,  11,  13,  15},
   {   9,  11,  12,  14},
   {   8,  10,  12,  14},
   {   8,   9,  11,  13},
   {   7,   9,  11,  12},
   {   7,   9,  10,  12},
   {   7,   8,  10,  11},
   {   6,   8,   9,  11},
   {   6,   7,   9,  10},
   {   6,   7,   8,   9},
   {   2,   2,   2,   2}
 };
```

next_state_LPS, next_state_MPS

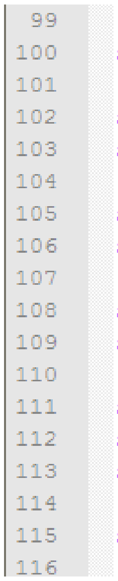


```
static int AC_next_state_MPS_64[64] =
  {
    1,2,3,4,5,6,7,8,9,10,
    11,12,13,14,15,16,17,18,19,20,
    21,22,23,24,25,26,27,28,29,30,
    31,32,33,34,35,36,37,38,39,40,
    41,42,43,44,45,46,47,48,49,50,
    51,52,53,54,55,56,57,58,59,60,
    61,62,62,63
  };

static int AC_next_state_LPS_64[64] =
  {
    0, 0, 1, 2, 2, 4, 4, 5, 6, 7,
    8, 9, 9,11,11,12,13,13,15,15,
    16,16,18,18,19,19,21,21,22,22,
    23,24,24,25,26,26,27,27,28,29,
    29,30,30,30,31,32,32,33,33,33,
    34,34,35,35,35,36,36,36,37,37,
    37,38,38,63
  };
```

APPENDIX B

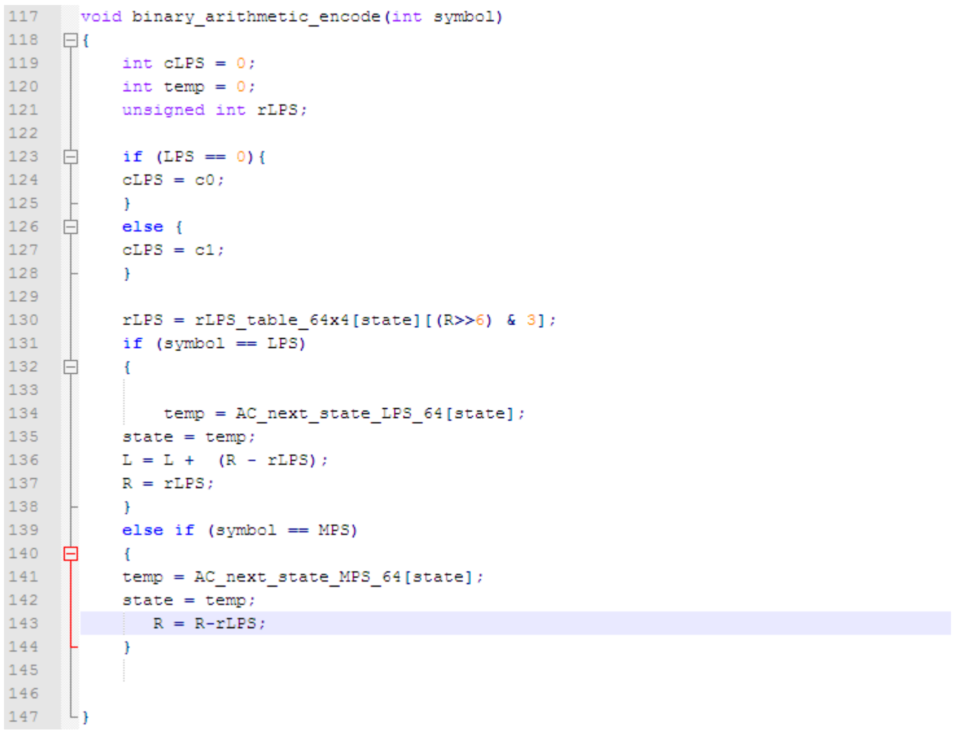
C Source Code

Global variables



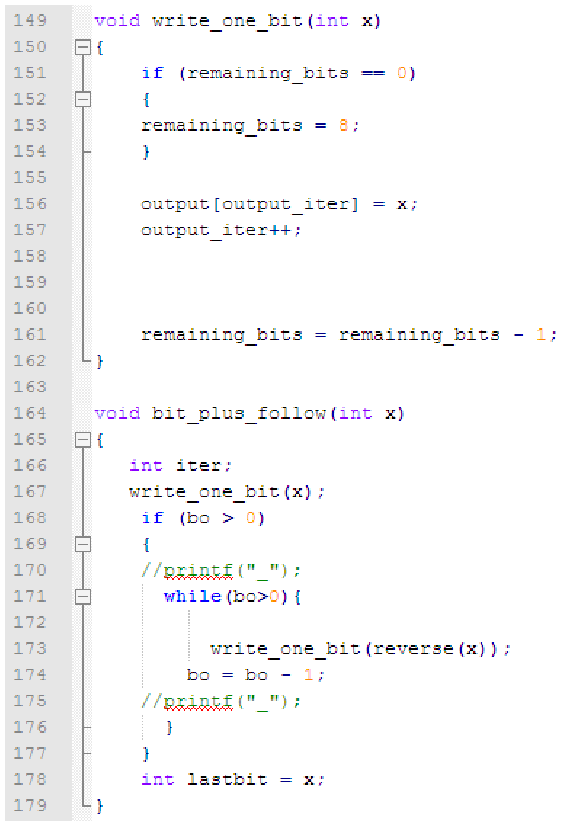
```c

static int bo;

static int LPS;
static int MPS;

static int L;
static int R;

static int lastbit;
static int remaining_bits;

static int output[200];
static int output_iter;
static int output_number;

static int state;

```

Binary encode Function



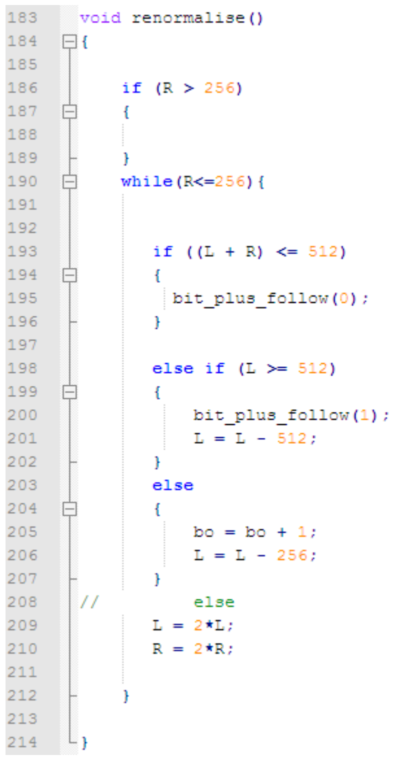
```c
void binary_arithmetic_encode(int symbol)
{
    int cLPS = 0;
    int temp = 0;
    unsigned int rLPS;

    if (LPS == 0){
    cLPS = c0;
    }
    else {
    cLPS = c1;
    }

    rLPS = rLPS_table_64x4[state][(R>>6) & 3];
    if (symbol == LPS)
    {

        temp = AC_next_state_LPS_64[state];
    state = temp;
    L = L +  (R - rLPS);
    R = rLPS;
    }
    else if (symbol == MPS)
    {
    temp = AC_next_state_MPS_64[state];
    state = temp;
       R = R-rLPS;
    }


}
```

write_one_bit Function / bit_plus_follow Function



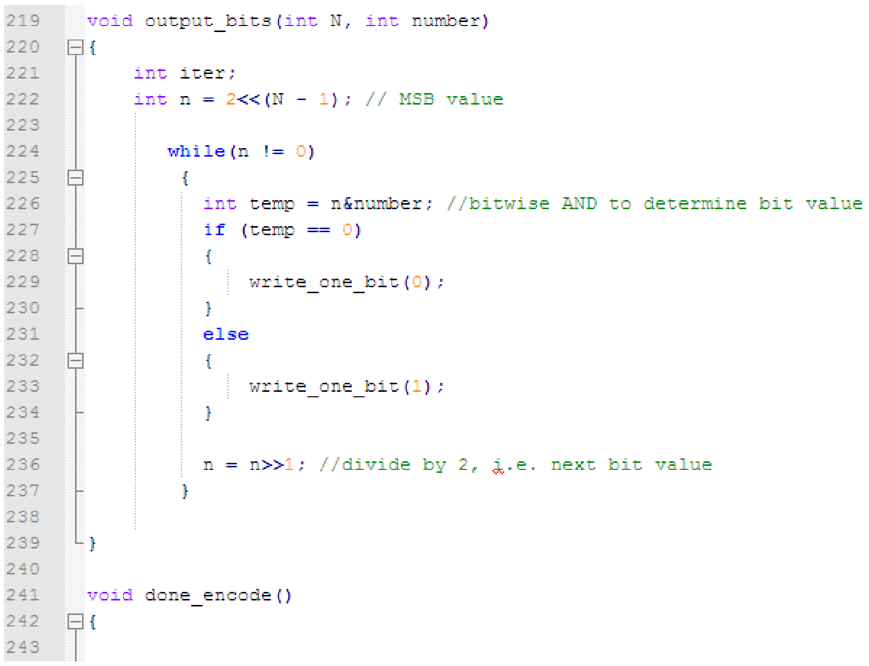
```
void write_one_bit(int x)
{
    if (remaining_bits == 0)
    {
    remaining_bits = 8;
    }

    output[output_iter] = x;
    output_iter++;



    remaining_bits = remaining_bits - 1;
}

void bit_plus_follow(int x)
{
   int iter;
   write_one_bit(x);
    if (bo > 0)
    {
    //printf("_");
      while(bo>0){

          write_one_bit(reverse(x));
        bo = bo - 1;
    //printf("_");
      }
    }
    int lastbit = x;
}
```

renormalize Fuction



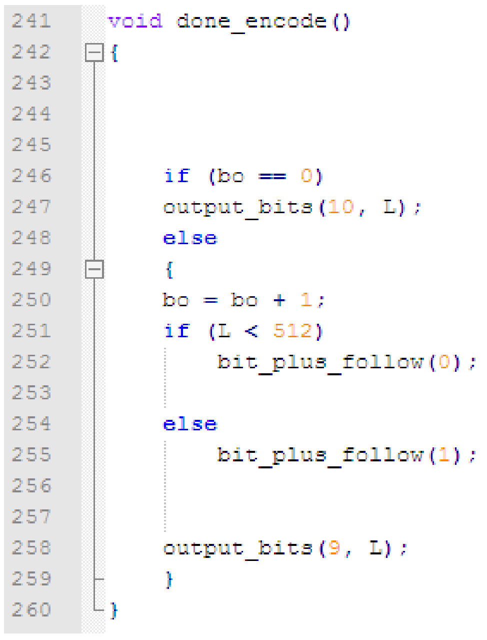
```
void renormalise()
{

    if (R > 256)
    {

    }
    while(R<=256){


        if ((L + R) <= 512)
        {
         bit_plus_follow(0);
        }

        else if (L >= 512)
        {
           bit_plus_follow(1);
           L = L - 512;
        }
        else
        {
           bo = bo + 1;
           L = L - 256;
        }
//              else
        L = 2*L;
        R = 2*R;

    }

}
```

output_bits Function



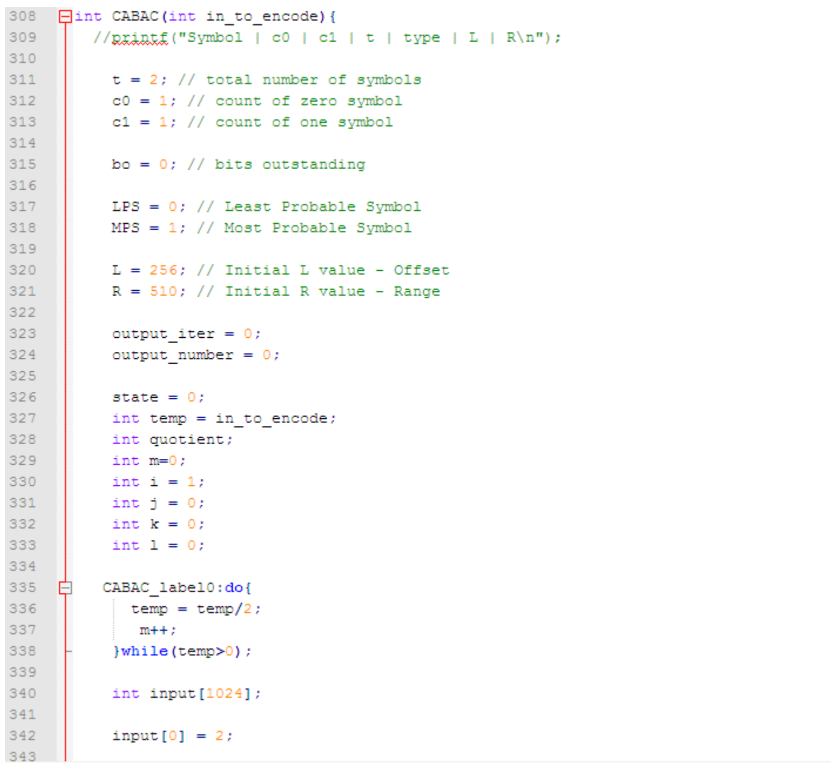
```
void output_bits(int N, int number)
{
    int iter;
    int n = 2<<(N - 1); // MSB value

       while(n != 0)
        {
          int temp = n&number; //bitwise AND to determine bit value
          if (temp == 0)
          {
              write_one_bit(0);
          }
          else
          {
              write_one_bit(1);
          }

          n = n>>1; //divide by 2, i.e. next bit value
        }

}

void done_encode()
{

```

done_encode Function

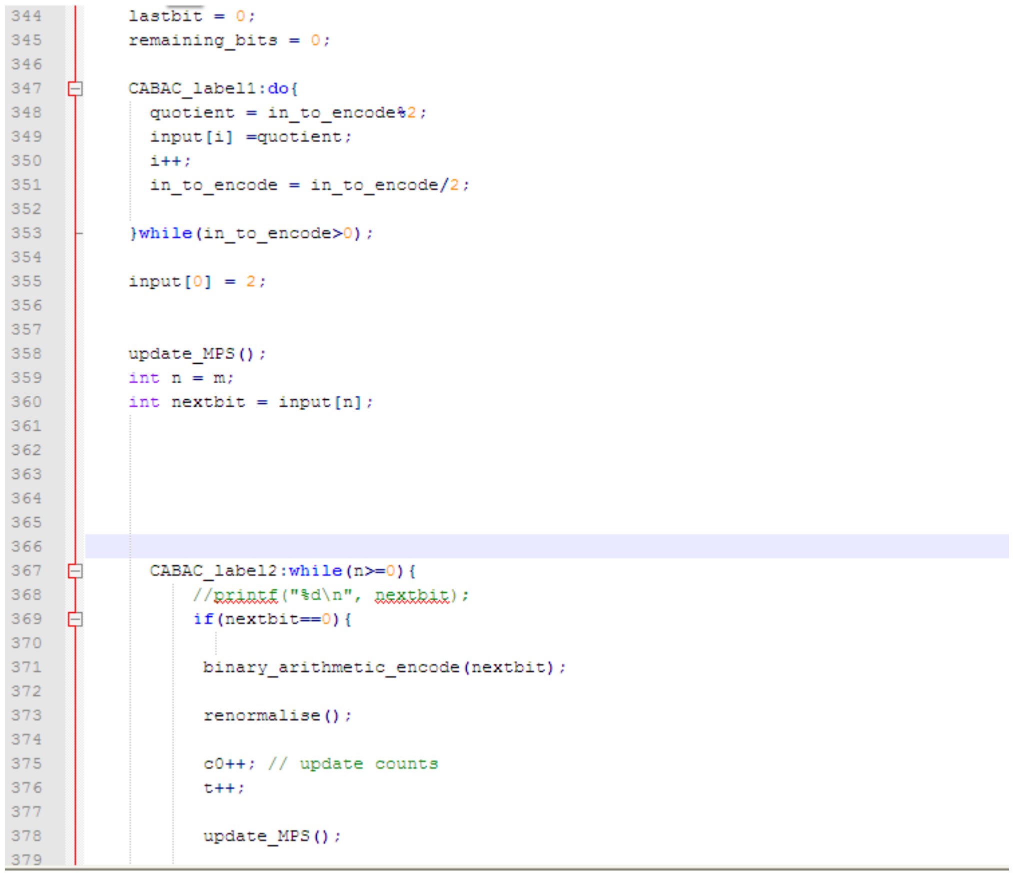


```
void done_encode()
{



    if (bo == 0)
    output_bits(10, L);
    else
    {
    bo = bo + 1;
    if (L < 512)
        bit_plus_follow(0);

    else
        bit_plus_follow(1);


    output_bits(9, L);
    }
}
```

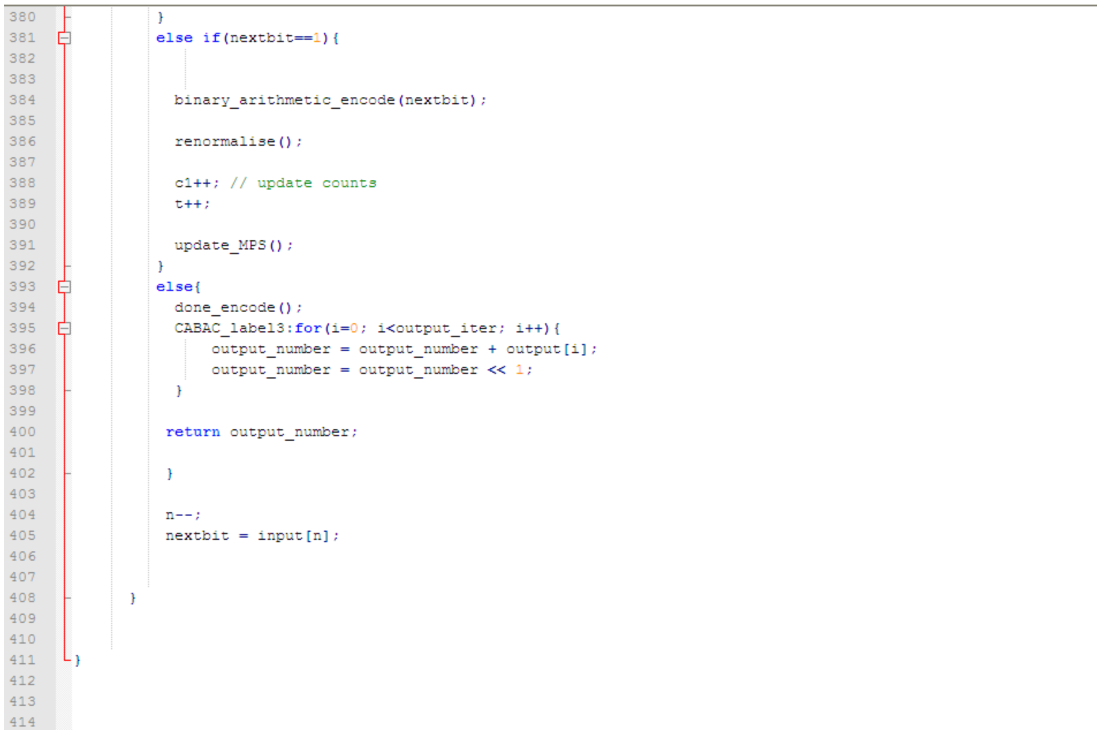
CABAC Function (top function)



```
int CABAC(int in_to_encode){
  //printf("Symbol | c0 | c1 | t | type | L | R\n");

    t = 2; // total number of symbols
    c0 = 1; // count of zero symbol
    c1 = 1; // count of one symbol

    bo = 0; // bits outstanding

    LPS = 0; // Least Probable Symbol
    MPS = 1; // Most Probable Symbol

    L = 256; // Initial L value - Offset
    R = 510; // Initial R value - Range

    output_iter = 0;
    output_number = 0;

    state = 0;
    int temp = in_to_encode;
    int quotient;
    int m=0;
    int i = 1;
    int j = 0;
    int k = 0;
    int l = 0;

   CABAC_label0:do{
      temp = temp/2;
       m++;
    }while(temp>0);

    int input[1024];

    input[0] = 2;

```



```
    lastbit = 0;
    remaining_bits = 0;

    CABAC_label1:do{
      quotient = in_to_encode%2;
      input[i] =quotient;
      i++;
      in_to_encode = in_to_encode/2;

    }while(in_to_encode>0);

    input[0] = 2;


    update_MPS();
    int n = m;
    int nextbit = input[n];






      CABAC_label2:while(n>=0){
          //printf("%d\n", nextbit);
          if(nextbit==0){

           binary_arithmetic_encode(nextbit);

           renormalise();

           c0++; // update counts
           t++;

           update_MPS();

```



```
         }
         else if(nextbit==1){


           binary_arithmetic_encode(nextbit);

           renormalise();

           c1++; // update counts
           t++;

           update_MPS();
         }
         else{
           done_encode();
           CABAC_label3:for(i=0; i<output_iter; i++){
               output_number = output_number + output[i];
               output_number = output_number << 1;
           }

          return output_number;

          }

          n--;
          nextbit = input[n];


      }


}
```