

# Optimization of Arithmetic and MQ Coding

Krishna Bharath Kolluru

University of Wisconsin Madison

**Abstract**—This work discusses about the optimization of MQ coding using ARM architecture and optimization of Arithmetic Coding for parallel processing. The former involves the optimization of the algorithm using the features of the platform while the latter involves optimizing with respect to the algorithm. The work also presents a novel idea of reformulating the Sequential Arithmetic Coding for Parallel Architectures.

**Index Terms**—MQ coding, Arithmetic Coding, Sequential AC and Parallel AC.

## I. INTRODUCTION

THE ever increasing demand for network communication and the need for increasing space have lead to drastic improvements in coding to reduce the data rate and also conserve space which increasing at a rate lesser than the demand.

Data compression or source coding is the process of encoding information using fewer bits (or other information-bearing units) than an un-encoded representation would use through use of specific encoding schemes. The most important data compression techniques that are used currently are Huffman and Arithmetic Coding.

Data can be compressed whenever some symbols are more likely than the other. Shannon showed that for the best possible compression code (in the sense of minimum average code length), the output length contains a contribution of  $-\log p$  bits from the encoding of each symbol whose probability of occurrence is  $p$ . If we can provide an accurate model for the probability of occurrence of each possible symbol at every point in a file, we can use arithmetic coding to encode the symbols that actually occur; the number of bits used by arithmetic coding to encode a symbol with probability  $p$  is very nearly  $-\log p$ , so the encoding is very nearly optimal for the given probability estimates.

The advantages that make arithmetic coding attractive are its flexibility and optimality. Flexibility means that arithmetic coding can be used in conjunction with any model that can provide a sequence of event probabilities. This advantage is significant because large compression gains can be obtained only through the use of sophisticated models of the input data.

Arithmetic coding is optimal in theory and very nearly optimal in practice, in the sense of encoding using minimal average code length. This optimality is often less important than it might seem, since Huffman coding is also very nearly optimal in most cases. When the probability of some single symbol is close to 1, however, arithmetic coding does give considerably better compression than other methods. The case of highly unbalanced probabilities occurs naturally in bi-level (black and white) image coding.

The main disadvantage of Arithmetic coding is its speed. It generally requires multiplication and division which slows down the process. However, in the recent times this has been overcome. One final minor problem is that arithmetic codes have poor error resistance, especially when used with adaptive models. In practice, the poor error resistance of adaptive coding is unimportant, since we can simply apply appropriate error correction coding to the encoded file.

The main idea of the QM-coder is to classify the input bit as **More Probable Symbol (MPS)** and **Less Probable Symbol (LPS)**. Before the next bit is input, the QM-coder uses a statistical model (using a context, typically a two-dimensional context of black and white pixel in an image) to predict which one of the bits (0 or 1) will be the *MPS*. If the predicted *MPS* bit does not match with the actual bit, then the QM-coder will classify this as *LPS*; otherwise, it will continue to be classified as *MPS*. The output of the coder is simply a compressed version of a stream of *MPS* or *LPS*, which are assigned probability values dynamically. The decoder has only the knowledge of whether the next predicted bit is *MPS* or *LPS*. It uses the same statistical model as that of the encoder to obtain the actual values of the bit.

The section II covers motivation that inspired the project. Section III covers the basics of MQ coder and its implementation is covered in Section IV. Section V covers the Sequential Arithmetic Coding and Section VI covers the Parallel Arithmetic Coding. Results are covered in section VII. Section VIII concludes the work while section IX completes the work with possible future work.

## II. MOTIVATION

The era of multiprocessor has long begun. It is important to analyze the existing algorithms and adapt them to the evolving multiprocessor architectures. This brings to the point where the current algorithms have to be studied thoroughly to

Manuscript received May 14, 2009. This work is a part of the course project work for ECE 734.

Kolluru Krishna Bharath is with the University of Wisconsin, Madison, WI 53706 USA (e-mail: kkolluru@wisc.edu).

understand the existing dependence flow and change them accordingly so that they can utilize the parallelism, if any that exists in them.

Different platforms often support different features. This requires a very good understanding of the algorithm and efficient implementation to extract the best features provided by the platform to get the best results.

### III. MQ CODER

#### A. Algorithm

The basic arithmetic coding has two major difficulties: the shrinking current interval that requires the use of high precision arithmetic, and no output is produced until the entire file has been read. The most straightforward solution to both of these problems is to output each leading bit as soon as it is known, and then to double the length of the current interval so that it reflects only the unknown part of the final interval. Witten, Neal, and Cleary [5] add a clever mechanism for preventing the current interval from shrinking too much when the end points are close to  $1/2$  but straddle  $1/2$ . In that case we do not yet know the next output bit, but we do know that what-ever it is, the following bit will have the opposite value; we merely keep track of that fact, and expand the current interval symmetrically about  $1/2$ . This follow-on procedure may be repeated any number of times, so the current interval size is always strictly longer than  $1/4$ .

#### B. Algorithm

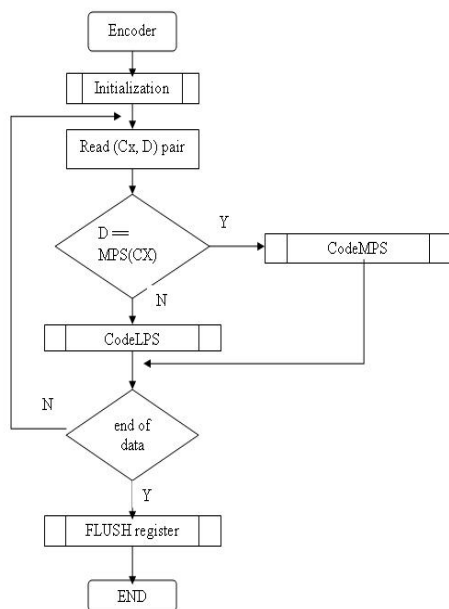


Fig. 1. The Flow Chart of MQ coding.

1. A “current interval”  $[L, H)$  is initialized to  $[0, 1)$ .
2. For each event in the file, two steps are performed.
  - (a) The current interval is subdivided into subintervals, one for each possible event. The size of an event's subinterval is proportional to the estimated probability that the event will be the next event in the file, according to the model of the input.
  - (b) The subinterval corresponding to the event that actually occurs next is selected, and it is updated as the new current interval.
  - (c) Execute the following steps repeatedly in sequence until the loop is explicitly halted:
    - (1). If the new subinterval is not entirely within one of the intervals  $[0, 1/2)$ ,  $[1/4, 3/4)$ , or  $[1/2, 1)$ , the loop is exited and returned.
    - (2). If the new subinterval lies entirely within  $[0, 1/2)$ , output 0 and any following 1s left over from previous events; then the size of the subinterval is doubled by linearly expanding  $[0, 1/2)$  to  $[0, 1)$ .
    - (3). If the new subinterval lies entirely within  $[1/2, 1)$ , output 1 and any following 0s left over from previous events; then the size of the subinterval is doubled by linearly expanding  $[1/2, 1)$  to  $[0, 1)$ .
    - (4). If the new subinterval lies entirely within  $[1/4, 3/4)$ , keep track of this fact for future output by incrementing the follow count; then the size of the subinterval is doubled by linearly expanding  $[1/4, 3/4)$  to  $[0, 1)$ .
3. Enough bits are outputted to distinguish the final current interval from all other possible final intervals.

The Fig.2 shows an example of MQ coding which explains the algorithm.

#### C. ARM Architecture

In general, an algorithm can be efficiently implemented using the special features that are provided by the particular platform. Here, Arithmetic coding is implemented on two different platforms namely, MATLAB and ARM. MATLAB is a tool that uses high level language while on ARM the algorithm is written using assembly level language.

HLL (High Level Language) programs are machine independent. They are easy to learn, easy to use, and convenient for managing complex tasks. Assembly language programs are machine specific. It is the language that the processor directly understands.

Implementation of the algorithm on different platforms calls for different implementations of the algorithm. MATLAB uses floating point arithmetic. However, not all processors support floating point processor. Consider ARM, ARM7TDMI doesn't support floating point operations. Therefore, unlike in MATLAB, where we can define the floating point precision, in ARM we have to implement the code using integers.

A few differences in the implementation the algorithm in ARM and MATLAB are as follows:

- (a) The main advantage of writing code in ARM is the optimization that is obtained by removing unnecessary if

condition which otherwise require branch instructions. In ARM, predication is used i.e. condition codes are used which removes such branches that may be required while converting a high level language to assembly language.

(b) ARM provides Thumb instructions which reduce the code space that is required to code the algorithm. This is very specific to ARM architecture.

(c) In ARM, Division is performed from the first principles unlike in MATLAB which assumes to have a DIVISION unit.

Compilers take advantage of the architecture underneath. Examples of machine specific optimizations are Predication and Software pipelining. ARM architecture supports Predication. ARM supports the option of setting the flag for arithmetic and logical instruction. This gives an additional advantage of not placing the CMP instructions before the branch instruction and thereby reducing the dependence on the branch predictor. The next section covers the implementation of the MQ coding on ARM architecture.

Action	Subintervals
Start	$[0, 1)$
Subdivide with left prob. $p\{a_1\} = \frac{2}{5}$	$[0, \frac{2}{5}), [\frac{2}{5}, 1)$
Input $b_1$ , select right subinterval	$[\frac{2}{5}, 1)$
Output 1, expand	$[\frac{1}{5}, 1)$
Subdivide with left prob. $p\{a_2\} = \frac{1}{2}$	$[\frac{1}{5}, \frac{2}{5}), [\frac{2}{5}, 1)$
Input $a_2$ , select left subinterval	$[\frac{1}{5}, \frac{2}{5})$
Increment follow count, expand	$[\frac{1}{6}, \frac{2}{6})$
Subdivide with left prob. $p\{a_3\} = \frac{3}{5}$	$[\frac{1}{6}, \frac{17}{30}), [\frac{17}{30}, \frac{2}{6})$
Input $b_3$ , select right subinterval	$[\frac{17}{30}, \frac{2}{6})$
Output 1, output 0 (follow bit), expand	$[\frac{2}{15}, \frac{2}{5})$
Output 01	$[\frac{1}{15}, \frac{1}{2})$ is entirely within $[\frac{2}{15}, \frac{2}{5})$

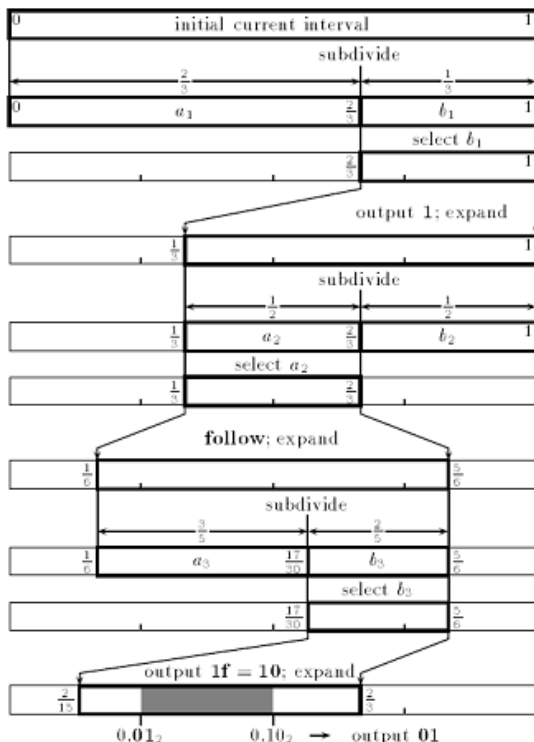


Fig. 2. The Table and the Diagram shows the steps taken during MQ coding for the example.

#### IV. IMPLEMENTATION OF MQ CODING ON ARM

The Objective of predication is to reduce eliminate hard-to-predict branches and increase ILP. It is important to notice that not all algorithms benefit from the implementation on ARM using predication. An algorithm with predictable branches and larger loops may not benefit from predication.

(a)

```
//UPDATING THE COUNT
LDRB R4,[R1] ;R4 HAS THE SYMBOL.
LDRB R5,[R2] ;R5 HAS THE COUNT OF NUMBER OF ZEROS.
LDRB R6,[R3] ;R6 HAS THE COUNT OF NUMBER OF ONES.
CMP R4,#0 ;CHECK IF THE VALUE THAT IS READ FROM THE
SOURCE IS 1/0.
ADDNE R5,R5,#1 ;IF ZERO, ADD 1 TO THE COUNT OF
COUNT_0.
ADDEQ R6,R6,#1 ;IF ONE, ADD 1 TO THE COUNT OF COUNT_1.
STRB R5,[R2] ;UPDATING COUNT_1
STRB R6,[R3] ;THE COUNT_0 HAS BEEN UPDATED WITH THE
NEW VALUE.
```

(b)

```
//EVALUATING THE MPS AND THE LPS USING
PREDICATION.
CMP R5,R6 ;CHECK IF THE COUNT_0>COUNT_1.
MOVGE R4,#1 ;IF YES, MOVE 1 INTO R4.
MOVLT R4,#0 ;IF NO, MOVE 0 INTO R4.
LDRR0,=(MPS)
STRB R4,[R0]
```

(c)

```
//IF NO PREDICATION ; THE CODE WOULD LOOK
LIKE
LDR R0,=(MPS)
CMP R5,R6
BGE LOOP1 ;HIGHLY UNPREDICTABLE BRANCH.
MOV R4,#1
STRB R4,[R0]
B EXIT
LOOP1: MOV R4,#0
STRB R4,[R0]
EXIT
```

Fig. 3. (a) Code shows the initialization of the variables. (b)The code shows the implementation of evaluating MPS and LPS with predication. (c) Performs the same function as I (b) but is implemented without predication. Both the memory required reduces and also improves the performance.

##### A. Predication

The idea of predication is to check for the condition once and update a flag which is then used by later instructions to decide whether to execute the instruction depending on the value of the flag. For example, from the above Fig. 3 which shows the small portion of the code implemented on ARM platform for MQ coding, we can see the application of predication. Using CMP instruction in Fig.3(b) sets the predicate and this is then used by the MOV instruction to decide whether to move or not by checking if the flag for Less

Than or Greater than or Equal is set. In Fig.3(c) CMP is used to branch out depending on whether the flag was set or not. The predicate can also be set by arithmetic and logical instructions.

### B. Integer Arithmetic

In practice, the arithmetic can be done by storing the endpoints of the current interval as sufficiently large integers rather than in floating point or exact rational numbers. We also use integers for the frequency counts used to estimate event probabilities. The subdivision process involves selecting non-overlapping intervals (of length at least 1) with lengths approximately proportional to the counts. Table 3 illustrates the use of integer arithmetic using a full interval of  $[0;N) = [0;1024)$ . The length of the current interval is always at least  $N=4+2$ , 258 in this case, so we can always use probabilities precise to at least  $1=258$ ; often the precision will be near  $1=1024$ . In practice we use even larger integers; the interval  $[0; 65\ 536)$  is a common choice, and gives a practically continuous choice of probabilities at each step. The subdivisions in this example are not quite the same as those in Fig.4 because the resulting intervals are rounded to integers. The encoded file is 11001 as before, but for a longer input file the encodings would eventually diverge.

Action	Subintervals
Start	$[0, 1024)$
$p\{a_1\} = \frac{2}{3}$ ; subdivide with left probability = $\frac{683}{1024} \approx 0.66699$	$[0, 683), [683, 1024)$
Input $b_1$ , select right subinterval	$[683, 1024)$
Output 1, expand	$[342, 1024)$
Subdivide with left prob. $p\{a_2\} = \frac{1}{2}$	$[342, 683), [683, 1024)$
Input $a_2$ , select left subinterval	$[342, 683)$
Increment <b>follow</b> count, expand	$[172, 854)$
$p\{a_1\} = \frac{3}{5}$ ; subdivide with left probability = $\frac{409}{682} \approx 0.59971$	$[172, 581), [581, 854)$
Input $b_3$ , select right subinterval	$[581, 854)$
Output 1, output 0 ( <b>follow</b> bit), expand	$[138, 654)$
Output 01	$[256, 512)$ is entirely within $[138, 654)$

Fig. 6. MQ coding using integer Arithmetic.

### C. Adaptive probability Calculation

There are 2 methods to calculate the probability of the MPS and LPS. The first method is static and the other is dynamic. In static, we perform 2 passes, in the first pass the frequency of the bits 0 and 1 is calculated and then depending the count, the MPS is decided. In the second pass, the encoding is performed. In this method, the drawback is that, run time calculation of the probability cannot be performed and therefore the complete information needs to be available to perform the encoding.

In the second method, adaptive probability calculation method, dynamic calculation of the MPS is performed. In this method, the probability calculation and the encoding is done in one pass. The process of encoding could be slow in this case due to the dynamic updating of the MPS.

## V. SEQUENTIAL ARITHMETIC CODING

### A. Sequential Arithmetic Coding

The pseudo code for sequential arithmetic coding is as follows.

#### Algorithm: Sequential Arithmetic Coding

```

begin
  L=0;
  H=1;
  F(0)=0;
  for(j=1 to N)
  {
    i=index_of_symbol(j);
    L(j+1)=L(j)+(H(j)-L(j))*F(i-1);
    H(j+1)=L(j)+(H(j)-L(j))*F(i);
  }
  output((L+H)/2);
end

```

From the algorithm, it is evident that the present range depends on the previous range at any iteration. There is inter-loop dependence. This can be removed by splitting the given sequence of symbols and distributing it to the various processors. The data flow diagram of the sequential arithmetic coding is given in Fig.4.

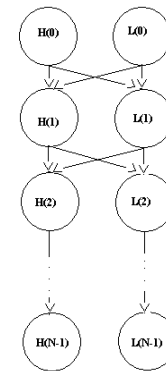


Fig. 5. Data flow diagram of sequential arithmetic coding.

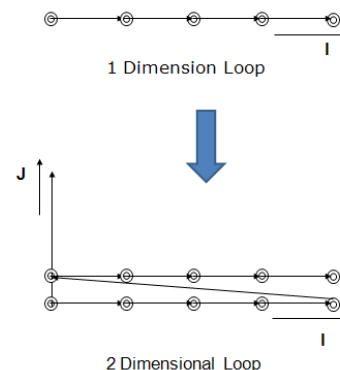


Fig. 6.(Top) Dependence graph for 1D loop. (Down) Dependence graph for 2D loop.

The Fig.5 shows that converting from 1D loop into 2D loop doesn't aid in achieving parallelism because the inner-loop and outer-loop parallelism, and loop interchange to achieve parallelism are absent.

The Dependence Matrix is given by  

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

The matrix shows that we cannot extract parallelism from the given formulation of the algorithm.

## VI. PARALLEL ARITHMETIC CODING

The parallelism that is inherently absent in the arithmetic coding can be extracted by splitting the sequence into each processor and assuming each sub-sequence to be separate sequence and then applying arithmetic coding to each one. This reduces the execution of the arithmetic coding by a factor equal to the number of processors. However, if we consider the overhead of moving the data into each processor, the speed up gets reduced from the ideal value. The result obtained from each processor can then be assumed as a new symbol and this new sequence of symbols with their ranges are then used to calculate the final range of the original sequence. Fig.6 shows the block diagram of the parallel arithmetic coding.

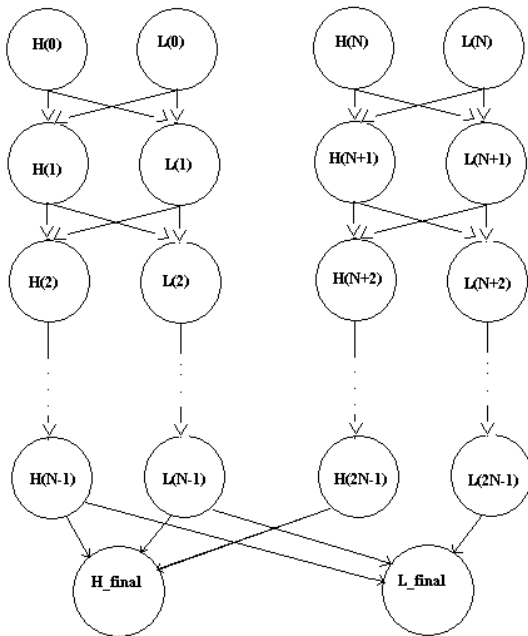


Fig. 7. Data flow diagram of parallel arithmetic coding for 2N symbols.

On changing the formulation which still gives the same result, we see that the dependence graph changes showing parallelism. Fig.7 shows the dependence graph for parallel arithmetic coding.

The dependence matrix for the above graph is

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

Since the 2<sup>nd</sup> row of the matrix is zero, as indicated by the graph has outer-loop parallelism. A pseudo code of the

parallel arithmetic coding implementation is as shown in Fig.8.

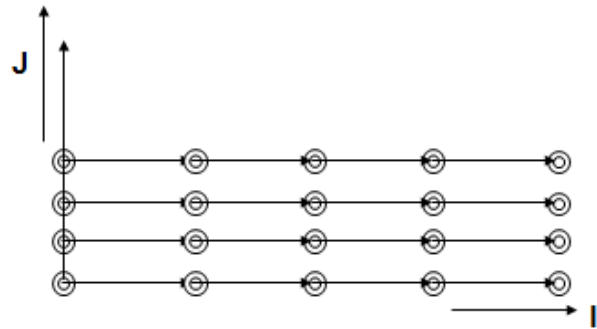


Fig. 8. Dependence graph for parallel arithmetic coding.

```

Do all i=1 to 2
Do j=1 to 2
  {
    l=index_of_symbol(j,i);
    L(j,i)=L(j,i)+(H(j,i)-L(j,i))*F(l-1);
    H(j,i)=L(j,i)+(H(j,i)-L(j,i))*F(l);
  }
Enddo
Enddoall
L_final = L12 + (H12-L12)*L22;
H_final = L12 + (H12-L12)*H22;

```

Fig. 9. Pseudo code for parallel arithmetic coding.

### Overhead

The overhead that is inherent in parallel processing is that the subsequence along with its probability has to be loaded into each processor. This overhead is large when the sequence is small. This results in poor speedup. But when the sequence is very large we can see that the speed up achieved is good.

## VII. RESULTS

### A. MQ coding

Implementation of the MQ coding on MATLAB and ARM architecture show a difference in the performance of the algorithm. On ARM architecture, utilizing *machine specific optimization* "predication", performance analysis shows that using Predication, we get a fractional speed up of 2.75 on replacing a conditional branch instruction, i.e. a reduction from 0.264us to 0.096us for a clock frequency of 41MHz (24ns). Each time a symbol (1/0) is encoded we save 7 cycles (i.e.7 cycles/run) for every predicated instruction used instead of branch instruction. When this is executed, say, on a black & white image of size 256x256, we save ~0.5M cycles.

### B. Parallel Arithmetic Coding

The parallel Arithmetic Coding for a text message of length 1800 showed the follow speed up of 1.66 (with the overhead)

with 4 processor architecture. Further, the speed up increased to 2 when the length increased to a length of 10000.

### VIII. CONCLUSION

The implementation of the MQ coding on ARM results in optimization of the algorithm to best suit the platform. Utilizing the features of the platform efficiently shows that the speed up of the system can be improved. At the same time, implementing “novel” parallel arithmetic coding shows reformulating the arithmetic coding can result improving the performance of the algorithm.

This work covers only encoding of MQ coding and arithmetic coding.

### IX. FUTURE WORK

There are certain limitations with the parallel arithmetic coding. The problem with sequential arithmetic coding still exists. Mainly, the precision required increases with increase in the sequence length. However, a possible solution to this

problem which needs further research would be limiting the precision to say 32 bit and then transmitting the information as soon the precision required for further encoding exceeds 32 bits. Once the information is transmitted, the range calculator can be reset to continue calculating for the rest of the sequence. Implementation of a parallel arithmetic decoding could be a challenging problem.

### REFERENCES

- [1] Howard & Vitter, “Arithmetic Coding for Data Compression”.
- [2] David Sehr, Jay Bharadwaj, Jim Pierce, Priti Shrivastav, Carole Dulong, “IA-64 Compiler Technology”.
- [3] Utpal Bannerjee, “Loop Parallelization”.
- [4] Pierre Boulet, Darté and Silber, “Loop parallelization Algorithms: from parallelism extraction to code generation”.
- [5] I. H. Witten, R. M. Neal and J. G. Cleary, “Arithmetic Coding for Data Compression,” *Comm. ACM*, 30, no. 6, pp. 520{540, June 1987.}
- [6] Supol and Melichar, “Arithmetic Coding in Parallel”.

## APPENDIX

### ARM CODE FOR MQ CODER

MQ CODER: THE FOLLOWING SECTION IS THE MAIN CODE THAT PERFORMS THE MQ ENCODING.  
ARM

```

AREA FLASH,CODE,READONLY

INCLUDE ADuC7026.inc
EXPORT __main
IMPORT __ADAPTIVE_PROBABILITY
IMPORT Reset_Handler
IMPORT MPS
IMPORT PROB_LPS
IMPORT PROB_MPS
IMPORT __MQCODING
IMPORT COUNT_1
IMPORT COUNT_0
EXPORT RANGE_LOW
EXPORT RANGE_HIGH
EXPORT SOURCE
IMPORT FOLLOW
IMPORT ENCODED
IMPORT __ENCODING_VALUE_FINAL_RANGE

; FIRST, WE HAVE TO CALCULATE THE PROBABILITY OF THE INPUT TEXT ADAPTIVELY. THIS IS DONE BY
; READING THE DATA
; FROM THE FILE AND THEN CALCULATING THE PROBABILITY OF THE SYMBOL.
; HERE WE CONSIDER CALCULATING THE PROBABILITY OF THE MOST PROBABLE SYMBOL AND THE LEAST
; PROBABLE SYMBOL
; THE INITIAL COUNT OF THE MPS AND THE LPS IS ONE. THERE ARE ONLY 2 POSSIBLE INPUTS. THEY ARE 1
; AND 0.

SOURCEDCB 1,1,0,9

    ALIGN
__main

    LDR R0,=(RANGE_LOW)
    LDR R1,=(RANGE_HIGH)
    MOV R2,#0
    STR R2,[R0]           ;RANGE_LOW IS LOADED WITH THE INITIAL VALUE OF INTERVAL LOW WHICH IS
0.
    LDR R2,=(1024)
    STR R2,[R1]           ;RANGE_HIGH IS LOADED WITH THE INITIAL VALUE OF INTERVAL HIGH WHICH
IS 1023.
    LDR R0,=(FOLLOW)
    MOV R1,#0
    STR R1,[R0]           ;INITIALIZING THE 'FOLLOW' COUNT TO ZERO.
    LDR R2,=(COUNT_1)   ;R2 HAS THE COUNT OF NUMBER OF 1s
    LDR R3,=(COUNT_0)   ;R3 HAS THE COUNT OF NUMBER OF 0s
    MOV R5,#1             ;THE INITIAL VALUE OF COUNT_1 IS ONE.
    MOV R6,#2             ;THE INITIAL VALUE OF COUNT_0 IS ONE.
    STRB R5,[R2]
    STRB R6,[R3]
    LDR R1,=(SOURCE)     ;R1 HAS THE STARTING ADDRESS OF THE MESSAGE i.e. SOURCE.

```

```

LDR R12,=(ENCODED)
LOOP
BL __ADAPTIVE_PROBABILITY
BL __MQCODING
ADD R1,R1,#1 ;R1 IS INCREMENTED BY ONE AFTER THE ADAPTIVE PROBABILITY CALCULATION
AND MQCODING IS PERFORMED.
LDRB R11,[R1] ;CHECK IF THE SYMBOL IS '9' WHICH INDICATES IT IS THE EOF OF THE FILE.
CMP R11,#9
BNE LOOP
BL __ENCODING_VALUE_FINAL_RANGE ;AFTER THE FINAL RANGE IS CALCULATED, IT IS IMPORTANT
TO TRANSMIT BITS THAT CORRESPOND TO THE FINAL RANGE.

WAIT
B WAIT

AREA SRAM,DATA,READWRITE
RANGE_LOW SPACE 4
RANGE_HIGH SPACE 4
END

```

---

**SUBROUTINE ADAPTIVE\_PROBABILITY: THIS SECTION PERFORMS THE RUNTIME PROBABILITY OF THE MPS AND LPS AND THEN ASSIGNS 1 OR 0 DYNAMICALLY TO MPS.**

```

ARM

AREA FLASH,CODE,READONLY

INCLUDE ADuC7026.inc
EXPORT __ADAPTIVE_PROBABILITY
EXPORT MPS
EXPORT PROB_LPS
EXPORT PROB_MPS
EXPORT COUNT_1
EXPORT COUNT_0
IMPORT SOURCE
IMPORT Reset_Handler

; FIRST, WE HAVE TO CALCULATE THE PROBABILITY OF THE INPUT TEXT ADAPTIVELY. THIS IS DONE BY
READING THE DATA
; FROM THE FILE AND THEN CALCULATING THE PROBABILITY OF THE SYMBOL.
; HERE WE CONSIDER CALCULATING THE PROBABILITY OF THE MOST PROBABLE SYMBOL AND THE LEAST
PROBABLE SYMBOL
; THE INITIAL COUNT OF THE MPS AND THE LPS IS ONE. THERE ARE ONLY 2 POSSIBLE INPUTS. THEY ARE 1
AND 0.

__ADAPTIVE_PROBABILITY
PUSH {R0,R2-R12,LR} ;CONTEXT SAVING
LDR R7,=(PROB_LPS) ;R7 HAS THE ADDRESS OF THE PROBABILITY OF LPS
LDR R8,=(PROB_MPS) ;R8 HAS THE ADDRESS OF THE PROBABILITY OF MPS
MOV R9,#0

ADAPTIVE_PROBABILITY

LDRB R4,[R1] ;R4 HAS THE SYMBOL
LDRB R5,[R2]
LDRB R6,[R3]
;EVALUATING THE MPS AND THE LPS.
CMP R5,R6 ;CHECK IF THE COUNT_0>COUNT_1.
MOVGE R4,#1 ;IF YES, MOVE 1 INTO R4.

```

```

MOVLT R4,#0      ;IF NO, MOVE 0 INTO R4.
LDR  R0,=(MPS)
STRB R4,[R0]

;CALCULATION OF THE PROBABILITY OF ZERO AND ONE.
;PROBABILITY(1)=COUNT_1/(COUNT_1+COUNT_0)
;PROBABILITY(0)=COUNT_0/(COUNT_1+COUNT_0)
ADD  R0,R5,R6 ;R0 HAS THE VALUE (COUNT_1+COUNT_0)
LSL  R5,R5,#10 ;THE COUNT_1 IS MULTIPLIED BY 1024.
LSL  R6,R6,#10 ;THE COUNT_0 IS MULTIPLIED BY 1024.
DIVISION_1
SUB  R5,R5,R0
ADD  R9,R9,#1
CMP  R5,R0
BGE  DIVISION_1
LSL  R5,R5,#1 ;NOW THAT R5 IS LESS THAN R0, R5 IS MULTIPLIED BY 2 AND CHECKED WHETHER IT IS
GREATER THAN R0,
CMP  R5,R0 ;IF YES,
ADDGT R9,R9,#1 ;THE VALUE IS ROUNDED OFF TO THE NEXT INTEGER VALUE BY ADDING A VALUE
OF 1.
;UPDATING THE PROBABILITY OF MPS AND LPS
CMP  R4,#1
BNE  UPDATING_LPS
STR  R9,[R8] ;THE PROBABILITY(1) IS STORED IN PROB_MPS
MOV  R9,#0
B  DIVISION_0
UPDATING_LPS
STR  R9,[R7] ;THE PROBABILITY(1) IS STORED IN PROB_LPS
MOV  R9,#0

DIVISION_0
SUB  R6,R6,R0
ADD  R9,R9,#1
CMP  R6,R0
BGE  DIVISION_0
LSL  R6,R6,#1 ;NOW THAT R5 IS LESS THAN R0, R5 IS MULTIPLIED BY 2 AND CHECKED WHETHER IT IS
GREATER THAN R0,
CMP  R6,R0 ;IF YES,
ADDGT R9,R9,#1 ;THE VALUE IS ROUNDED OFF TO THE NEXT INTEGER VALUE BY ADDING A VALUE OF
1.
;UPDATING THE PROBABILITY OF MPS AND LPS
CMP  R4,#0
BNE  UPDATING_LPS_1
STR  R9,[R8] ;THE PROBABILITY(0) IS STORED IN PROB_MPS

B  EXIT
UPDATING_LPS_1
STR  R9,[R7] ;THE PROBABILITY(0) IS STORED IN PROB_LPS
EXIT

;THIS IS DONE AFTER THE PROBABILITY CALCULATION FOR THE NEXT SYMBOL.
;UPDATING THE COUNT
LDRB R4,[R1] ;R4 HAS THE SYMBOL.
LDRB R5,[R2] ;R5 HAS THE COUNT OF NUMBER OF ZEROS.
LDRB R6,[R3] ;R6 HAS THE COUNT OF NUMBER OF ONES.
CMP  R4,#0 ;CHECK IF THE VALUE THAT IS READ FROM THE SOURCE IS 1 OR 0.
ADDNE R5,R5,#1 ;IF ZERO, ADD 1 TO THE COUNT OF COUNT_0
ADDEQ R6,R6,#1 ;IF ONE, ADD 1 TO THE COUNT OF COUNT_1
STRB R5,[R2] ;THE COUNT_1 HAS BEEN UPDATED WITH THE NEW VALUE
STRB R6,[R3] ;THE COUNT_0 HAS BEEN UPDATED WITH THE NEW VALUE

```

```
POP {R0,R2-R12,PC}
```

```
AREA SRAM,DATA,READWRITE
```

```
PROB_LPS SPACE 4
PROB_MPS SPACE 4
ALIGN
COUNT_1 SPACE 1
COUNT_0 SPACE 1
MPS SPACE 1
END
```

---

**SUBROUTINE MQCODING: THIS SECTION OF THE CODE PERFORMS THE MQCODING.**

```
;FIRST, WE NEED TO CHECK WHETHER THE SYMBOL IS MPS OR LPS.(DONE)
;SECOND, ON DETECTING WHETHER MPS AND LPS, WE HAVE TO CALCULATE THE RANGE THAT IT WOULD
CURRENTLY BELONG TO.(DONE)
;THIRD, CHECK WHETHER THE RANGE IS WITH IN ANY OF THE 3 DEFINED REGIONS, I.E.
(0,0.5);(0.25,0.75);(0.5,1);
;FOURTH, IF IT LIES IN ANY OF THE GIVEN REGIONS THEN WE SHOULD UPDATE THE CURRENT RANGE
ACCORDINGLY.
;FIFTH, IF IT DOES NOT LIE IN ANY OF THE GIVEN REGIONS THEN WE HAVE TO INCREMENT THE "FOLLOW"
COUNT.
;SIXTH, HAVE TO BROADCAST THE COMPLEMENT OF THE VALUE "FOLLOW" NUMBER OF TIMES WHEN IT
LIES IN (0,0.5);(0.5,1);
;THE FINAL RESULT WILL LIE IN THE ARRAY NAME ENCODED MESSAGE.
```

```
ARM
```

```
AREA FLASH,CODE,READONLY
INCLUDE ADuC7026.inc
IMPORT MPS
IMPORT PROB_LPS
IMPORT PROB_MPS
IMPORT SOURCE
IMPORT RANGE_LOW
IMPORT RANGE_HIGH
EXPORT __MQCODING
EXPORT FOLLOW
EXPORT ENCODED
```

```
__MQCODING
```

```
PUSH {R0-R11,LR}

LDR R2,=(RANGE_LOW)
LDR R3,=(RANGE_HIGH)
LDR R8,=(PROB_MPS)
LDR R9,[R8] ;R9 HAS THE PROB_MPS(PROB_MPS IS ASSUMED TO BE LOCATED BELOW PROB_LPS IN
THE RANGE SCALE).
LDR R6,[R2] ;R6 HAS THE LOWER LIMIT OF THE CURRENT RANGE.
LDR R7,[R3] ;R7 HAS THE HIGHER LIMIT OF THE CURRENT RANGE.
LDR R4,=(MPS) ;R4 IS LOADED WITH THE STARTING ADDRESS OF MPS.
LDRB R4,[R4] ;R4 HAS THE MOST PROBABLE SYMBOL.
LDRB R5,[R1] ;LOADING R5 WITH THE FIRST SYMBOL OF THE MESSAGE.
LDR R1,=(FOLLOW)

;COMPARISON TO CHECK WHETHER THE CURRENT SYMBOL LOADED(R5) IS MPS OR LPS.
CMP R4,R5
```

```

BNE LPS_CALC
;SINCE THE COMPARISON RETURNS A ZERO, I.E. THEY ARE EQUAL, WE KNOW THAT THE CURRENT SYMBOL
IS MPS.
STR R6,[R2] ;AS LONG THE SYMBOL ENCODED IS MPS, LOWER LIMIT WONT CHANGE.
;RANGE_HIGH(NEW)=RANGE_LOW(OLD)+(RANGE_HIGH(OLD)-RANGE_LOW(OLD))*PROB_MPS
SUB R7,R7,R6 ;R7=(RANGE_HIGH(OLD)-RANGE_LOW(OLD))
MUL R7,R7,R9 ;R7=(R7*(RANGE_HIGH(OLD)-RANGE_LOW(OLD)))*PROB_MPS
ANDS R11,R7,#0X1 ;CHECK IF THE LAST BIT IS ONE OR ZERO. IF ONE IT IS ODD.
ADDNE R7,R7,#1 ;IF R7 IS ODD, ADD 1 TO THE VALUE OF R7.(THIS IS DONE SO THAT PRECISION IS NOT
LOST(TO MAJOR EXTENT.))
LSR R7,R7,#10 ;DIVIDING THE VALUE OF R7 BY 1024, TO GET THE RANGE IN (0,1024).
ADD R7,R7,R6 ;R7=RANGE_HIGH(NEW)
STR R7,[R3] ;RANGE_HIGH IS LOADED WITH THE NEW VALUE.
B EXIT_RANGE_CALCULATION
;LPS_CALC PERFORMS THE CALCULATION WHEN THE SYMBOL LOADED IS LPS
LPS_CALC
STR R7,[R3] ;AS LONG THE SYMBOL ENCODED IS LPS, HIGHER LIMIT WONT CHANGE.
;RANGE_LOW(NEW)=RANGE_LOW(OLD)+(RANGE_HIGH(OLD)-RANGE_LOW(OLD))*PROB_LPS
SUB R7,R7,R6 ;R7=(RANGE_HIGH(OLD)-RANGE_LOW(OLD))
MUL R7,R7,R9 ;R7=(R7*(RANGE_HIGH(OLD)-RANGE_LOW(OLD)))*PROB_MPS
LSR R7,R7,#10 ;DIVIDING THE VALUE OF R7 BY 1024, TO GET THE RANGE IN (0,1024).
ADD R7,R7,R6 ;R7=RANGE_LOW(NEW)
STR R7,[R2] ;RANGE_LOW IS LOADED WITH THE NEW VALUE.

EXIT_RANGE_CALCULATION
.....
;NOW COMES THE PART WHERE THE RANGE HAS TO COMPARED WITH THE 3 REGIONS TO TRANSMIT A 'ONE'
OR 'ZERO' AND 'FOLLOW'.
;CHECK IF THE RANGE LIES IN [0,512]
;INTRODUCING A COUNT(R11), TO CHECK WHICH REGION IT BELONG TO.
MOV R11,#0
LDR R6,[R2]
LDR R7,[R3]
CMP R7,#512 ;CHECK IF RANGE_HIGH<=512.
ADDLE R11,R11,#1
CMP R11,#1
BEQ STAGE1
MOV R11,#0
CMP R6,#512 ;CHECK IF RANGE_LOW>=512.
ADDGE R11,R11,#2
CMP R11,#2
BEQ STAGE2
MOV R11,#0
CMP R6,#256 ;CHECK IF RANGE_LOW>=256.
ADDGE R11,R11,#3
CMP R7,#768 ;CHECK IF RANGE_HIGH<=768.
ADDLE R11,R11,#3
CMP R11,#6
BEQ STAGE3
B EXIT ;IF THE RANGE DOESNT LIE IN ANY OF THE 3 INTERVALS THEN SIMPLY EXIT!!!
.....
;STAGE1 IS CALCULATION IN THE REGION (0,512)
STAGE1
LDR R11,[R1] ;LOADING R11 WITH THE CURRENT 'FOLLOW' COUNT.
MOV R0,#0
STRB R0,[R12],#1 ;OUTPUT ZERO.
ORR R0,R0,#0X1 ;TO OUTPUT THE COMPLEMENT OF ZERO, IF THE 'FOLLOW' COUNT IS NOT ZERO.
FOLLOW_STAGE1
CMP R11,#0 ;CHECK IF THE 'FOLLOW' COUNT IS ZERO
BEQ RESCALING1 ;IF YES, BRANCH OUT OF FOLLOW_STAGE1

```

```

STRB R0,[R12],#1 ;IF NOT, OUTPUT ONE, UNTIL THE 'FOLLOW' COUNT GOES TO ZERO.
SUB R11,R11,#1 ;DECREMENT 'FOLLOW' COUNT EACH TIME A ZERO IS OUTPUTTED.
B FOLLOW_STAGE1
RESCALING1
STR R11,[R1] ;STORING THE NEW VALUE OF 'FOLLOW' COUNT.
RESCALING_STAGE1_1
SUB R7,R7,R6 ;R7=(RANGE_HIGH-RANGE_LOW)
LSL R7,#1 ;R7=2*(RANGE_HIGH-RANGE_LOW)
LSL R6,#1 ;R6=RANGE_LOW(NEW)=2*(RANGE_LOW-0)
ADD R7,R7,R6;R7=RANGE_HIGH(NEW)=RANGE(NEW)+RANGE_LOW(NEW)
STR R6,[R2] ;RANGE_LOW IS LOADED WITH THE NEW VALUE.
STR R7,[R3] ;RANGE_HIGH IS LOADED WITH THE NEW VALUE.
B EXIT_RANGE_CALCULATION
.....
;STAGE2 IS CALCULATION IN THE REGION (512,1023)
STAGE2
LDR R11,[R1] ;LOADING R11 WITH THE CURRENT 'FOLLOW' COUNT.
MOV R0,#1
STRB R0,[R12],#1 ;OUTPUT ONE.
AND R0,R0,#0 ;TO OUTPUT THE COMPLEMENT OF ONE, IF THE 'FOLLOW' COUNT IS NOT ZERO.
FOLLOW_STAGE2
CMP R11,#0
BEQ RESCALING2
STRB R0,[R12],#1
SUB R11,R11,#1
B FOLLOW_STAGE2
RESCALING2
STR R11,[R1] ;STORING THE NEW VALUE OF 'FOLLOW' COUNT.
RESCALING_STAGE2_1
SUB R7,R7,R6 ;R7=(RANGE_HIGH-RANGE_LOW)
LSL R7,#1 ;R7=2*(RANGE_HIGH-RANGE_LOW)
SUB R6,R6,#512 ;R6=RANGE_LOW-512
LSL R6,#1 ;R6=RANGE_LOW(NEW)=2*(RANGE_LOW-512)
ADD R7,R7,R6;R7=RANGE_HIGH(NEW)=RANGE(NEW)+RANGE_LOW(NEW)
STR R6,[R2] ;RANGE_LOW IS LOADED WITH THE NEW VALUE.
STR R7,[R3] ;RANGE_HIGH IS LOADED WITH THE NEW VALUE.
B EXIT_RANGE_CALCULATION
.....
;STAGE3 IS CALCULATION IN THE REGION (256,768)
STAGE3
LDR R0,[R1] ;LOADING R0 WITH THE CURRENT 'FOLLOW' COUNT.
RESCALING_STAGE3_1
ADD R0,R0,#1 ;INCREMENTING THE 'FOLLOW' COUNT BY ONE.
SUB R7,R7,R6 ;R7=(RANGE_HIGH-RANGE_LOW)
LSL R7,#1 ;R7=2*(RANGE_HIGH-RANGE_LOW)
SUB R6,R6,#256 ;R6=RANGE_LOW-256
LSL R6,#1 ;R6=RANGE_LOW(NEW)=2*(RANGE_LOW-256)
ADD R7,R7,R6;R7=RANGE_HIGH(NEW)=RANGE(NEW)+RANGE_LOW(NEW)
STR R6,[R2] ;RANGE_LOW IS LOADED WITH THE NEW VALUE.
STR R7,[R3] ;RANGE_HIGH IS LOADED WITH THE NEW VALUE.
STR R0,[R1] ;LOAD THE 'FOLLOW' WITH THE NEW 'FOLLOW' COUNT.
B EXIT_RANGE_CALCULATION
.....
EXIT
POP {R0-R11,PC}

AREA SRAM,DATA,READWRITE
ENCODED SPACE 1000

```

```
FOLLOW SPACE 4
END
```

---

**SUBROUTINE \_\_ENCODING\_VALUE\_FINAL\_RANGE: CALCULATING THE FINAL RANGE**  
ARM

```
AREA FLASH,CODE,READONLY
```

```
INCLUDE ADuC7026.inc
IMPORT RANGE_LOW
IMPORT ENCODED
EXPORT __ENCODING_VALUE_FINAL_RANGE
```

```
__ENCODING_VALUE_FINAL_RANGE
```

```
PUSH {R0-R11,LR}
MOV R3,#10
MOV R0,#512
LDR R11,=(RANGE_LOW)
LDR R0,[R11]
CMP R0,#256
MOVLE R0,#256
BLE LOOP
CMP R0,#512
MOVLE R0,#512
```

```
LOOP
```

```
AND R2,R0,#0X200
LSR R2,#9
STRB R2,[R12],#1
LSL R0,#1
SUBS R3,R3,#1
BNE LOOP
```

```
POP {R0-R11,PC}
END
```

---

## MATLAB CODE FOR MQ ENCODING

(Reference Prof. Yu Hen Hu's code)

```
function arithscale()
clc;
fid=fopen('seq1.txt','r');
seq=fread(fid,'*char');
fclose(fid);
seq=reshape(seq,1,length(seq));
[alpha prob]=probmodel(seq);
if ~isempty(alpha)
    [tag mnm]=arithscalecod(alpha,prob,seq);
    disp(strcat('Tag =',tag));
    seq=arithscaledecod(tag,alpha,prob,length(seq),mnm);
    disp('Sequence = ');
    disp(seq);
else
    display('Empty Sequence....');
end
end
```

```

function [alpha prob]=probmodel(seq)
if ~isempty(seq)
    alpha(1)=seq(1);
    prob(1)=1;
    l=length(seq);
    k=2;
    for i=2:l
        idx=find(seq(i)==alpha);
        if isempty(idx)
            alpha(k)=seq(i);
            prob(k)=1;
            k=k+1;
        else
            prob(idx)=prob(idx)+1;
        end
    end
    prob=prob;
    prob=prob./l;
    disp(prob);
else
    alpha=[];
    prob=[];
end
end

function [tag mnm]=arithscalecod(alpha,prob,seq)
ls=length(seq);
l=0;u=1;
Fx(1)=0;
for i=1:length(prob)
    Fx(i+1)=Fx(i)+prob(i);
end
tag=[];
dif=[];
while ~isempty(seq)
    dif(end+1)=u-1;
    if l>=0 & u<0.5
        tag=strcat(tag,'0');
        l=2*l;
        u=2*u;
    elseif l>=0.5 & u<1
        tag=strcat(tag,'1');
        l=2*(l-0.5);
        u=2*(u-0.5);
    else
        p=find(seq(1)==alpha);
        l1=l+(u-1)*Fx(p);
        u=1+(u-1)*Fx(p+1);
        l=l1;
        seq(1)='';
    end
end
end
wl=8;
b=numb2bin(u,wl);
tag=strcat(tag,b);
mnm=min(dif);
end

function b=numb2bin(l,wl)

```

```

b=[];
for i=1:wl
    v=l*2;
    f=floor(v);
    b=strcat(b,num2str(f));
    l=v-f;
end
end

function seq=arithscaledecod(tag,alpha,prob,lgt,mnm)
l=0;u=1;
Fx(1)=0;
for i=1:length(prob)
    Fx(i+1)=Fx(i)+prob(i);
end
seq='';
k=ceil(log2(1/mnm));
k=2*k;
ln=length(tag);
if k>ln
    k=ln;
end
while lgt>0
    if l>=0 & u<0.5
        tag(1)='';
        tag(end+1)='0';
        l=2*l;
        u=2*u;
    elseif l>=0.5 & u<1
        tag(1)='';
        tag(end+1)='0';
        l=2*(l-0.5);
        u=2*(u-0.5);
    else
        b=tag(1:k);
        tg=bin2numb(b);
        t=(tg-1)/(u-1);
        for j=1:length(prob)
            if t>=Fx(j) & t<Fx(j+1)
                break
            end
        end
        seq=[seq alpha(j)];
        l1=1+(u-1)*Fx(j);
        u=1+(u-1)*Fx(j+1);
        l=l1;
        lgt=lgt-1;
    end
end
end

function d=bin2numb(b)
d=0;
for i=1:length(b)
    bt=str2num(b(i));
    d=d+bt*2^(-i);
end
end

```

# SEQUENTIAL AND PARALLEL ARITHMETIC CODING

## Main function:

### Sequential AC:

```
function coding(message,prob_alphabet,alphabet)

LowRange(1)=0;
HighRange(1)=1024;

for j=1:length(message)
    for i=1:length(alphabet)
        if(alphabet(i)==message(j))
            lx=sum(prob_alphabet(1:1:i-1));
            hx=sum(prob_alphabet(1:1:i));
            LowRange(j+1)=(LowRange(j) + (HighRange(j) - LowRange(j))*lx);
            HighRange(j+1)=(LowRange(j) + (HighRange(j) - LowRange(j))*hx);
        end
    end
end
LowRange=vpa(LowRange,50)*10^50;
encoded=dec2bin(LowRange(length(LowRange)));
```

---

### Parallel AC:

```
clc
clear all
format long;
fid=fopen('source.txt','r');
message=fread(fid,'*char');
fclose(fid);
message=reshape(message,1,length(message));
%display(message);
[alphabet prob_alphabet]=probmodel(message);
NP=input('The number of parallel processors ? :(1-5) ','s');
NP=str2num(NP);
tic
for i=1:NP
    if(i~=NP)
        for j=1:ceil(length(message)/NP)
            sequence(j,i)=message(j+(i-1)*ceil(length(message)/NP));
            for l=1:length(alphabet)
                if (alphabet(l)==sequence(j,i))
                    lx(j,i)=sum(prob_alphabet(1:1:l-1));
                    hx(j,i)=sum(prob_alphabet(1:1:l));
                end
            end
        end
    else
        for j=1:length(message) - (NP-1)*ceil(length(message)/NP)
            sequence(j,i)=message((NP-1)*ceil(length(message)/NP)+j);
```





