Data Compression

Arithmetic Coding

中央大學資工系
蘇柏齊

Problems of Huffman Coding

- Huffman code produces the best codes for the individual data symbols.
- However, the only case where it produces the ideal codes (with average size equals the entropy) is when the symbols have probabilities of negative powers of 2 (i.e., such as ½, ¼, …) since Huffman code assigns an integral number of bits to each symbol in the alphabet.
- Higher compression ratio may be achieved by combining several symbols into a single unit; however, the corresponding complexity for codeword construction will be increased. (Alphabet size grows exponentially.)
- Another problem with Huffman coding is that the coding and modeling steps are combined into a single process, and thus, adaptive coding is difficult.
  - Adaptive Huffman coding is not popular because of its decoding complexity.
  - If the probabilities of occurrence of the input symbols change, then one has to redesign the Huffman table.

Problems of Huffman Coding

- Bound
  $H(X) <= L(C_H) <= L(C_{SF}) < H(X)+1$
  $H(X) <= L(C_H) < H(X)+P_{max}+0.086$
  P_{max}: Prob. of the most frequently occurring symbol
- Example
  - Pr(A)=0.95
    Pr(B)=0.03
    Pr(C)=0.02
    Entropy=0.335 bits/symbol
  - Symbol A: 0
    Symbol B: 11
    Symbol C: 10
    Bavg=1.05 bits/symbol
    Redundancy=0.715 bits/symbol

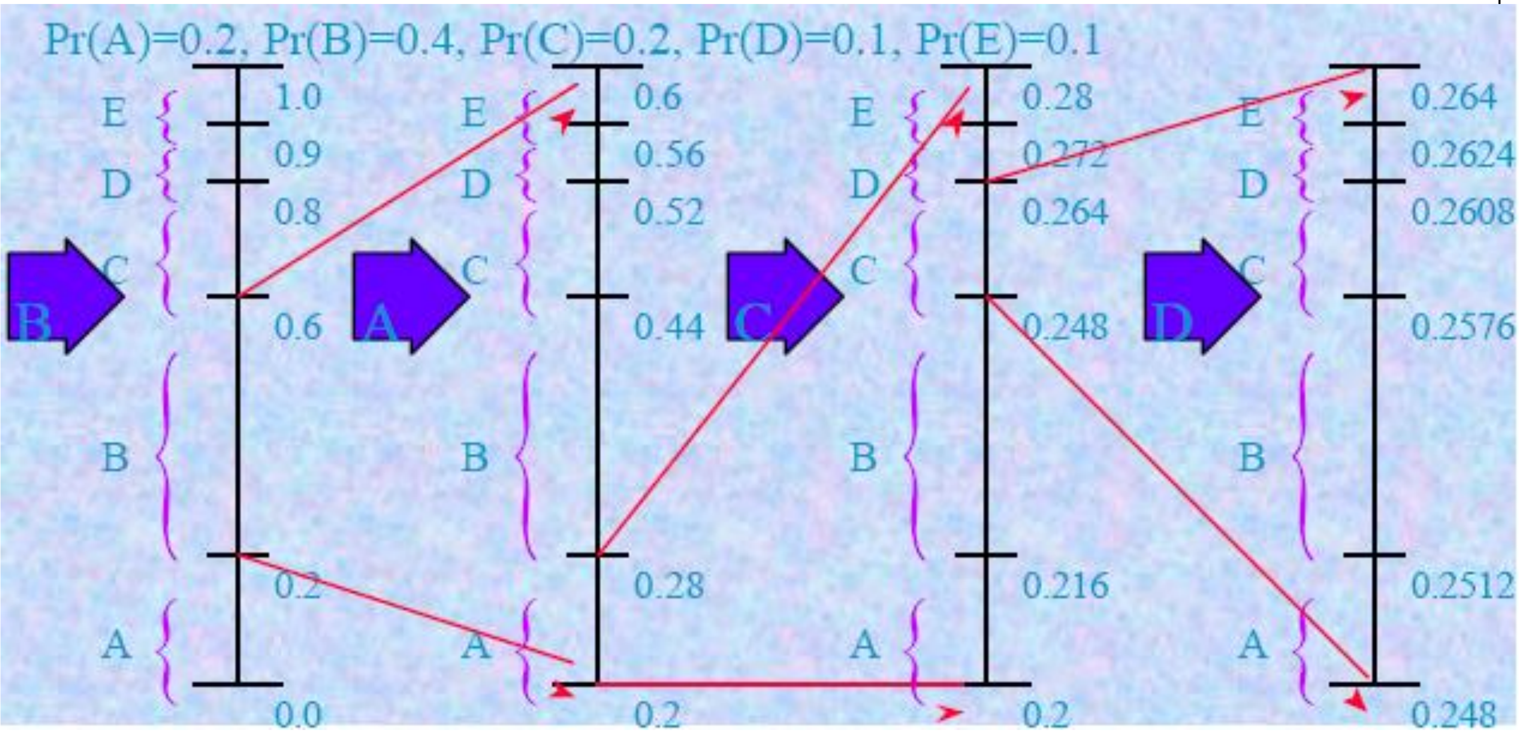
- Higher-order model still doesn't work well:
- The same example:
  - AA 0
    AB 1 1 1
    AC 1 0 0
    BA 1 1 0 1
    BB 1 1 0 0 1 1
    BC 1 1 0 0 0 1
    CA 1 0 1
    CB 1 1 0 0 1 0
    CC 1 1 0 0 0 0
    Entropy=0.669 bits/symbol
    Bavg=1.221 bits/symbol
    Redundancy=0.552 bits/symbol

Arithmetic Coding

- Arithmetic coding overcomes the problem of assigning integer codes to the individual symbols by assigning one (long) code to the entire input file.
  - Represent a stream of input symbols by a number (tag)
    - e.g. a number in [0,1).
  - The method starts with a certain interval, it reads the input file symbol by symbol, and uses the prob. of each symbol to narrow the interval.
  - Specifying a narrower interval requires more bits, so the number constructed by the algorithm grows continuously. To achieve compression, the algorithm is designed such that a high-probability symbol narrows the interval less than a low-probability symbol, with the result that high-probability symbols contribute fewer bits.
- Arithmetic coding is especially useful for dealing with sources with small alphabets, such as binary sources, and alphabets with highly skewed prob.
- Arithmetic coding can separate the coding from modeling. This allows for the dynamic adaptation of the probability model without affecting the design of the coder.

Arithmetic Coding

- Encoding procedure:



- If we want to encode the sequence “BACD”, then any number within the range of [0.2608, 0.2624) can be used to represent this sequence.
- In general, we choose the central value of the range as our output code, like 0.2616.
- Encoding is a process of narrowing down the range of possible numbers. The new range is proportional to the predefined probability

Arithmetic Coding

- Decoding procedure:
  - Find out the coding interval $[l_i,u_i)$ for the given code T and output the corresponding symbol belonging to that interval.
  - Change the value of T by:
    $T=(T-l_i)/(u_i-l_i)$
- Decoding example for given code 0.2616:

$$(1)\ \because\ T = 0.2616 \in [0.2, 0.6)\ \therefore \Rightarrow 'B',$$
$$\text{and } T = (T - 0.2)/(0.6 - 0.2) = 0.154$$
$$(2)\ \because\ T = 0.154 \in [0.0, 0.2)\ \therefore \Rightarrow 'A',$$
$$\text{and } T = (T - 0.0)/(0.2 - 0.0) = 0.77$$
$$(3)\ \because\ T = 0.77 \in [0.6, 0.8)\ \therefore \Rightarrow 'C',$$
$$\text{and } T = (T - 0.6)/(0.8 - 0.6) = 0.85$$

- Decoding is the inverse process. (The range is expanded in proportional to the probability of each extracted symbol.)

Arithmetic Coding

- Encoder:

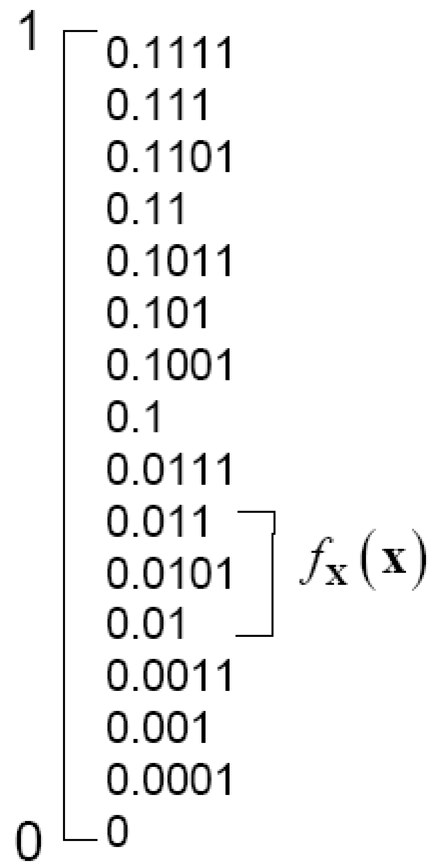
```
L=0.0;
U=1.0;
while not EOF do
   R=U-L;
   read symbol s;
   U=L+R * U(s)
   L=L+R * L(s)
enddo
output a number T in [L,U) (e.g. L, (L+U)/2)
```

- Decoder:

```
Input T
while not EOF (or while T!=0)
   Table lookup to find s, L(s)<=T<U(s)
   output symbol s;
   T=T-L(s)
   T=T/(U(s)-L(s))
enddo
```

Arithmetic Coding (in other words)

- Representation of **x** by a subinterval of the unit interval [0,1)
- Width of the subinterval is approximately equal to the probability $f_X(\mathbf{x})$
- Represent **x** by shortest binary fraction in the subinterval
- Subinterval of width $f_X(\mathbf{x})$ is guaranteed to contain one number that can be represented by L_m binary digits, with $L_m \sim -\log_2 f_X(\mathbf{x})$
- Entropy coding algorithm for sequences of symbols **x** with conditional probabilities



Properties of Arithmetic Coding

- $T_X(\mathbf{x})$ is a number in [0,1). A binary code for $T_X(\mathbf{x})$ can be obtained by taking its binary representation truncated to $l(\mathbf{x}) = ceil[\log_2(1/p(\mathbf{x}))]+1$ bits and it is a uniquely decodable code (prefix code).
  - Can be proved but we omit
- $l(\mathbf{x})$ = ceil[$\log_2(1/p(\mathbf{x}))$]+1 is the number of bits required to encode **x**. The average length of a sequence with length n is H(**x**)+2

$$l_{A^n} = \sum p(\mathbf{x}) l(\mathbf{x}) = \sum p(\mathbf{x})\left(\left\lceil \log \frac{1}{p(\mathbf{x})} \right\rceil + 1\right) \le \sum p(\mathbf{x})\left(\log \frac{1}{p(\mathbf{x})} + 1 + 1\right)$$

$$= \sum p(\mathbf{x}) \log \frac{1}{p(\mathbf{x})} + 2\sum p(\mathbf{x}) = H(\mathbf{x}) + 2$$

$$H(\mathbf{x}) \le l_{A^{(n)}} \le H(\mathbf{x}) + 2 \Rightarrow \frac{H(\mathbf{x})}{n} \le l_A \le \frac{H(\mathbf{x})}{n} + \frac{2}{n}$$

$$H(x) \le l_A < H(x) + \frac{2}{n}$$

Higher-order entropy!

$$\left(H(x) \le l_{Huffman} < H(x) + \frac{1}{n} \right)$$

Algorithm Implementation

- Recursive procedure:

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_x(x_n - 1)$$

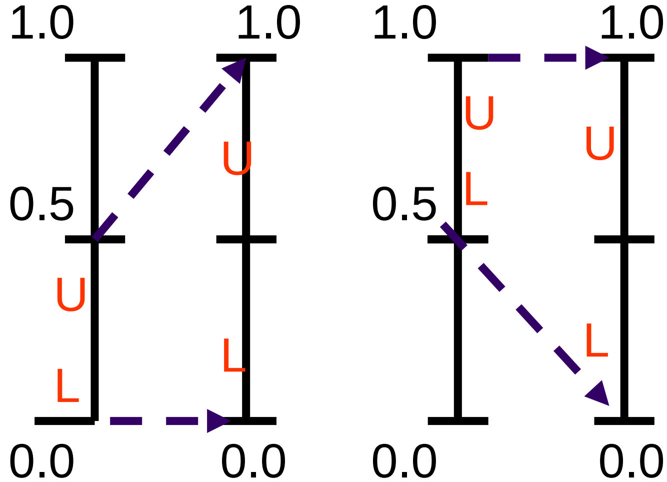
$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_x(x_n)$$

x_n: nth observed symbol

- Coding is related to the range on U and L
- One major problem: As n gets larger，U and L come closer
- Solution: scaling

Algorithm Implementation

- U & L are in [0,0.5).
  - First bits of U and L are both 0
  - E1 scaling:
    - E1(x)=2(x), [0,0.5)->[0,1)
- U & L are in [0.5,1).
  - First bits of U and L are both 1
  - E2 scaling:
    - E2(x)=2(x-0.5), [0.5,1)->[0,1)
- Incremental coding



Example:

{1, 2, 3}
P(1)=0.8,
P(2)=0.02,
P(3)=0.18
Fx(1)=0.8,
Fx(2)=0.82,
Fx(3)=1

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_x(x_n)$$

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_x(x_n - 1)$$

Note:
At each stage we are transmitting the MSB that is the same in both U and L of the tag interval. By sending the MSB's of the U and L of the tag, we are actually sending the binary representation of the tag.

```
Encode: 1 3 2 1
Encoding (1) L0=0 U0=1
             L1=0+(1-0)*0=0
             U1=0+(1-0)*0.8=0.8
Encoding (3) L2=0+(0.8-0)*0.82=0.656
             U2=0+(0.8-0)*1=0.8                SEND 1
             L2=2x(0.656-0.5)=0.312
             U2=2x(0.8-0.5)=0.6
Encoding (2) L3=0.5424
             U3=0.54816                        SEND 1
             L3= 2x(0.5424-0.5)=0.0848
             U3=2x(0.54816-0.5)=0.09632        SEND 0
             L3=2x(0.0848)=0.1696
             U3=2x(0.09632)=0.19264            SEND 0
             L3=2x(0.1696)=0.3392
             U3=2x(0.19264)=0.38528            SEND 0
             L3=2x(0.3392)=0.6784
             U3=2x(0.38528)=0.77056            SEND 1
             L3=2x(0.6784-0.5)=0.3568
             U3=2x(0.77056-0.5)=0.54112
Encoding (1) L4=0.3568
             U4=0.504256                       SEND 1
```



```
0.1100011=0.7734375 in [0.7712, 0.773505)
```

Example:

P(1)=0.8, P(2)=0.02, P(3)=0.18
Fx(1)=0.8, Fx(2)=0.82, Fx(3)=1
Decode:
110001100..0

L0=0, U0=1
110001=0.765625,
which lies in the [0,0.8)
Symbol = 1
L1=0, U1=0.8
0.765625
Which lies in the 82%~100% of [0,0.8)
Symbol = 3
L2=0.656, U2=0.8
(E2)
L2=2x(0.656-0.5)=0.312
U2=2x(0.8-0.5)=0.6
Shift one bit out of buffer and move one bit in
100011=0.546875,
which lies in 80%~82% of [0.312, 0.6)
Symbol = 2
L3=0.312+(0.6-0.312)x0.8=0.5424
U3=0.312+(0.6-0.312)x0.82=0.54816

(E2)
L3=0.0848
U3=0.09632
000110
(E1)
L3=2x0.0848=0.1696
U3=2x0.09632=0.19264
001100
(E1)
L3=0.3392
U3=0.38528
011000
(E1)
L3=0.6784
U3=0.77056
110000
(E2)
L3=0.3568
U3=0.54112
100000=0.5,
which lies in 0~80% of
[0.3568, 0.54112)
Symbol =1

E3 Scaling

- E3 scaling is applied when U & L straddle in the midpoint of the unit interval (L>0.25, U<0.75)
  - [0.25,0.75) -> [0,1), E3(x)=2(x-0.25)
- U = 10000000
  L = 01111111
- Take action when
  U = 10…
  L = 01…

E3 Scaling

- E2 scaling -> transmit '1', E1 scaling -> transmit '0'
  - E1(X) = 2X,
  - E2(X) = 2(X-0.5)
- How to inform E3 scaling?
  - E3(X) = 2(X-0.25)
  - 1010… -> 110….
- Strategy:
  - Choose not to send any information to the decoder at this time. Instead, we simply record the fact that we used an E3 scaling at the encoder (scale3)
  - Suppose after this, the tag interval gets confined to the upper half of the unit interval, we have to use an E2 scaling. The effect of the earlier E3 scaling can be mimicked at the decoder by following the E2 mapping with an E1 mapping.
  - So at the encoder, right after we send a 1 to announce the E2 scaling, we send a 0
    - E3-E2 2*(2(X-0.25)-0.5) = 2(2X-1) = 4X-2
    - E2-E1 2*2(X-0.5) = 4(X-0.5) = 4X-2
  - If the first rescaling after the E3 scaling is an E1 scaling, we do exactly the opposite.
    - E3-E1 2*2(X-0.25) = 4X-1
    - E1-E2 2(2X-0.5) = 4X-1

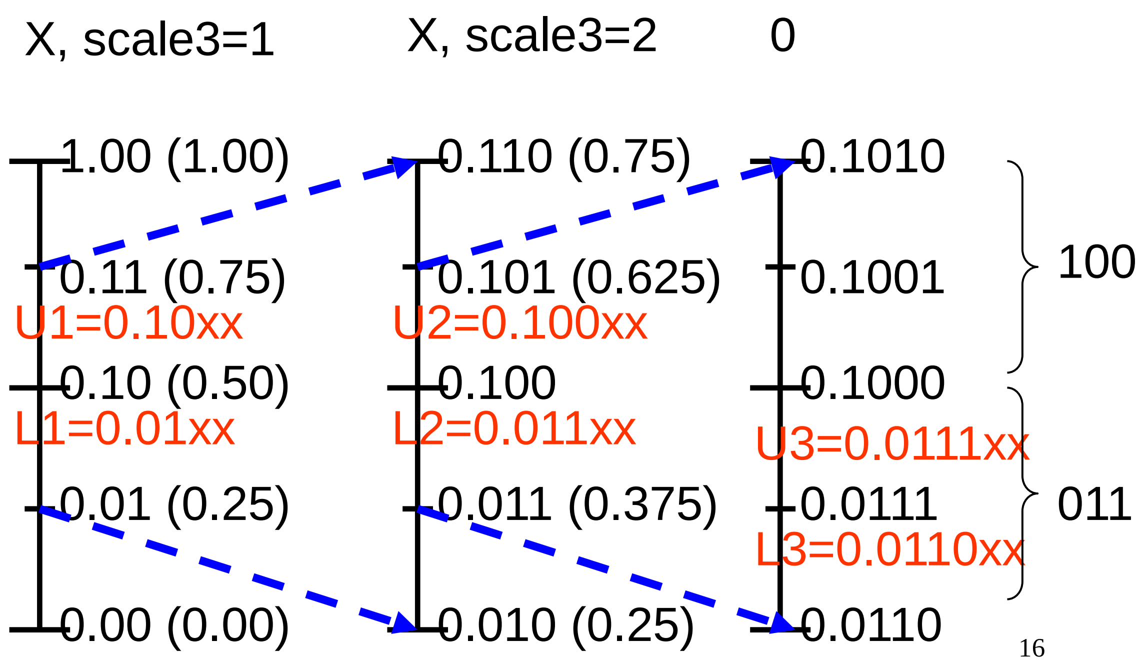
E3 Scaling

- What happens if we have to go through a series of E3 mappings at the encoder?
  - We simply keep track of the number of E3 mappings and then send that many bits of the opposite variety after the first E1 or E2 mapping
- A quicker view
  - U0.100000…

Case 1: 0.100000X

Case 2: 0.011111X

L0.011111…



Integer Implementation

$[0,1) \rightarrow 2^m_m$ binary word

$0 \rightarrow \overbrace{0.......0}^{m}$ $\quad 1 \rightarrow \overbrace{1.......1}^{m}$ $\quad 0.5 \rightarrow 0\overbrace{1.......1}^{m-1}$

$F_x(k) = \dfrac{Cum\_Sum(k)}{total\_count}$ $\quad Cum\_Sum(k) = \sum_{i=1}^{k} n_i$ $\quad n_i$: # of occurrence of symbol i

$$l^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1)Cum\_Sum(x_n - 1)}{total\_count} \right\rfloor$$

$$u^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1)Cum\_Sum(x_n)}{total\_count} \right\rfloor - 1$$

E_1 shift out MSB and shift in 1 in $u^{(n)}$

E_2 shift out MSB and shift in 0 in $l^{(n)}$

{1,2,3}
Encode:
1, 3, 2, 1

Count(1)=40
Count(2)=1
Count(3)=9
Total_Count=50

Cum_Count(0)=0
Cum_Count(1)=40
Cum_Count(2)=41
Cum_Count(3)=50
Scale3=0

L0=   0=00000000
U0=255=11111111
Encode 1
L1=   0=00000000
U1=203=11001011
Encode 3
L2=167=10100111
U2=203=11001011
E2 ***(1)***
L2=01001110= 78
U2=10010111=151
E3 scale3=1
L2=00011100= 28
U2=10101111=175
Encode 2
L3=146=10010010
U3=148=10010100
E2 ***(1)***
L3=00100100
U3=00101001
Transmit **(0)** scale3=0

E1 ***(0)***
E1 ***(0)***
E2 ***(1)***
E1 ***(0)***
L3=01000000
U3=10011111
E3, scale3=1
L3=00000000= 0
U3=10111111=191
Encode 1
L4=   0=00000000
U4=152=10011000
Send the current status of the tag
So send 00000000
(0)
But scale3=1
Transmit ***(1)*** scale3=0
Send 7 consecutive, ***(0000000)***

1100010010000000

Integer Implementation

- Init $l^{(0)} = \overbrace{0.......0}^{m}$ $u^{(0)} = \overbrace{1.......1}^{m}$
- For each k find $t^* = \dfrac{t - l^{(k-1)} + 1}{u^{(k-1)} - l^{(k-1)} + 1}$
- Find $x^{(k)}$ $\dfrac{Cum\_Count(x^{(k)} - 1)}{total\_count} \leq t^* \leq \dfrac{Cum\_Count(x^{(k)})}{total\_count}$
- Update $u^{(k)}$ $l^{(k)}$

Decode:

1100010010000000

T=11000100=196

L=0000000= 0

U=1111111=255

T* = 38

0<= T* <40 -> **1**

L= 0=00000000

U=203=11001011

T* = 48

41<= T* <50 -> **3**

L=167=10100111

U=203=11001011

E2

L=01001110

U=10010111

T=10001001

E3

L=00011100=28

U=10101111=175

T=10010010=146

T* = 40

40<= T* <41 -> **2**

L=146=10010010

U=148=10010100

E2 E1 E1 E2 E1

L=01000000

U=10011111

T=01000000

... -> **1**

QM Coder

- A binary adaptive arithmetic coder
  - Input: binary data {0,1}
- An entropy coder standard of JBIG and approved by CCITT in 1992
- A lineal descendent of the Q coder developed by IBM in 1988, but enhanced by the improvements of interval subdivision and probability estimation
  - Simplification in implementation via a state-transition diagram (Each state specifies a particular prob. And the possible transition to the next state)
  - Numerical resolution

QM Coder

- Derived before

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_x(x_n)$$

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_x(x_n - 1)$$

- $A^{(n)} = u^{(n)} - l^{(n)}$

$$A^{(n)} = A^{(n-1)}P(x_n)$$

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_x(x_n - 1)$$

- Symbols
  - MPS (More Probable Symbol)
  - LPS (Less Probable Symbol)
  - Prob(LPS)=q_c

- Procedure
  - MPS

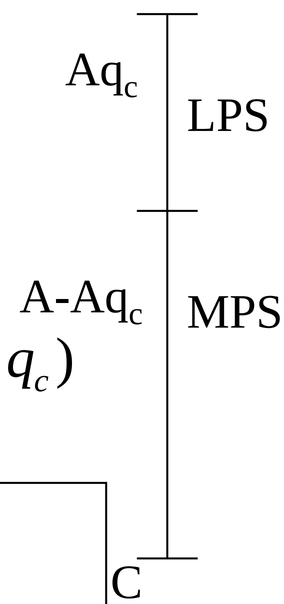
$$l^{(n)} = l^{(n-1)}$$

$$A^{(n)} = A^{(n-1)}(1 - q_c)$$

  - LPS

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}(1 - q_c)$$

$$A^{(n)} = A^{(n-1)}q_c$$



- Simplification
  - MPS

$$l^{(n)} = l^{(n-1)}$$

$$A^{(n)} = A^{(n-1)} - q_c$$

  - LPS

$$l^{(n)} = l^{(n-1)} + (A^{(n-1)} - q_c)$$

$$A^{(n)} = q_c$$

QM Coder Integer Implementation

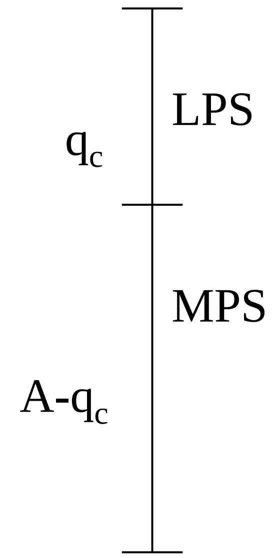
- To make sure $A^{(n)} \sim 1$, we double $A^{(n)}$ if it drops below 0.75. That is $A^{(n)}$ in [0.75,1.5)
- The same scaling is applied to $l^{(n)}$. The bits shifted out of $l^{(n)}$ make up the encoder output
- 16-bit implementation
  - 0X10000=1.5 (Upper bound for A)
  - 0X8000=0.75 (Lower bound for A)
  - 0XAAAA=1.00
- Procedure
  - After MPS
    - C is unchanged
    - $A=A-q_c$
    - If A<0X8000 (0.75)
      - Renormalize A and C
    - End
  - After LPS
    - $C=C+A-q_c$
    - $A=q_c$ (<0.5)
    - Renormalize A and C

QM Coder Conditional Exchange

Assume q_c=0.5, A=0.75 => A-q_c=0.25<q_c => Conditional exchange

- After MPS
  - C is unchanged
  - A=A-q_c
  - If A<0X8000
    - If A<q_c
      - C=C+A
      - A=q_c
    - End
    - Renormalize A and C
  - end

- After LPS
  - A=A-q_c
  - If A>q_c (OK)
    - C=C+A
    - A=q_c
  - End
  - renormalize



•Facts:

•Renormalization occurs every time an LPS occurs

•Renormalization may or may not occur when an MPS occurs

•Conditional exchange when q_c>$A^{(n)}$-q_c. Switch MPS and LPS interval at this time.

QM Coder

- q_c is not fixed. It's updated each time a rescaling take place. An ordered list of q_c is listed in a table. Every time a rescaling occurs, q_c is changed to the next lower or higher state, depending on LPS or MPS.
- Switch the roles of MPS and LPS (if MPS=1, change to MPS=0) when q_c becomes larger than 0.5 (in the state-transition table).
- We can design contexts to better predict q_c

State	q_c (Hex)	q_c (Dec)	Increase state by	Decrease state by
0	59EB	0.49582	1	S
1	5522	0.46944	1	1
2	504F	0.44283	1	1
3	4B85	0.41643	1	1
4	4639	0.38722	1	1
5	415E	0.36044	1	1
6	3C3D	0.33216	1	1
7	375E	0.30530	1	1
8	32B4	0.27958	1	2
9	2E17	0.25415	1	1
10	299A	0.22940	1	2
11	2516	0.20450	1	1
12	1EDF	0.17023	1	1
13	1AA9	0.14701	1	2
14	174E	0.12581	1	1
15	1424	0.11106	1	2
16	119C	0.09710	1	1
17	0F6B	0.08502	1	2
18	0D51	0.07343	1	2
19	0BB6	0.06458	1	1
20	0A40	0.05652	1	2
21	0861	0.04620	1	2
22	0706	0.03873	1	2
23	05CD	0.03199	1	2
24	04DE	0.02684	1	1
25	040F	0.02238	1	2
26	0363	0.01867	1	2
27	02D4	0.01559	1	2
28	025C	0.01301	1	2
29	01F8	0.01086	1	2
30	01A4	0.00905	1	2
31	0160	0.00758	1	2
32	0125	0.00631	1	2
33	00F6	0.00530	1	2
34	00CB	0.00437	1	2
35	00AB	0.00368	1	1
36	008F	0.00308	1	2
37	0068	0.00224	1	2
38	004E	0.00168	1	2
39	003B	0.00127	1	2
40	002C	0.00095	1	2
41	001A	0.00056	1	3
42	000D	0.00028	1	2
43	0006	0.00013	1	2
44	0003	0.00006	1	2
45	0001	0.00002	0	1

Model of Context

- Conditional probability -> Context-based

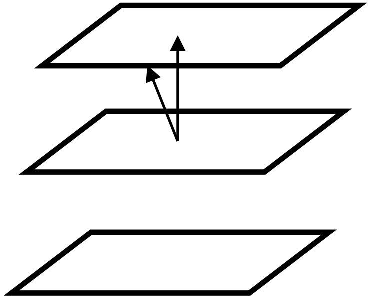
  0 0 0
  1 0 0 0 1
  0 1 0 x

  00010001010=138

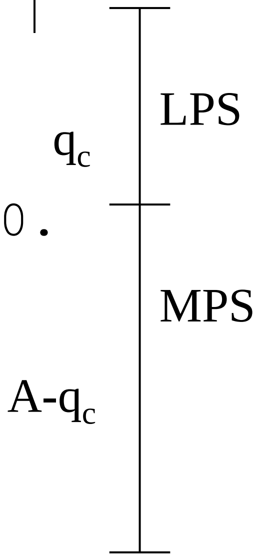
  0 0 1 1 1 0
  0 0 0 1 X

  001110001=113

- Bit-plane context
  - Context can be made bit-plane based
  - JPEG2000



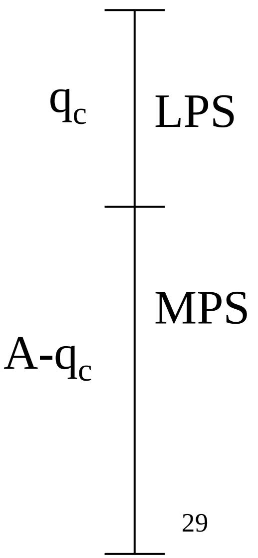
QM Encoder (Summary)




```
Initialization:
Less Probable Symbol (LPS)='1'
More Probable Symbol (MPS)='0'.
Prob(LPS)=qc=0.49582. State=0. A=0xFFFF. C=0x0000.
if encoder receives MPS
{
    A=A-qc;
    if A<0x8000
    {
        if A<qc
        {C+=A; A=qc;}
        A<<=1;
        qc changes its state according to
        Column 4 in Table1;
        //EX: Qc=01FB(state29) and changes to 01A4(state30)
        //because column 4 indicates increasing 1
        Encoder outputs MSB of C;
        C<<=1;
    }
}
```

QM Encoder (Summary)

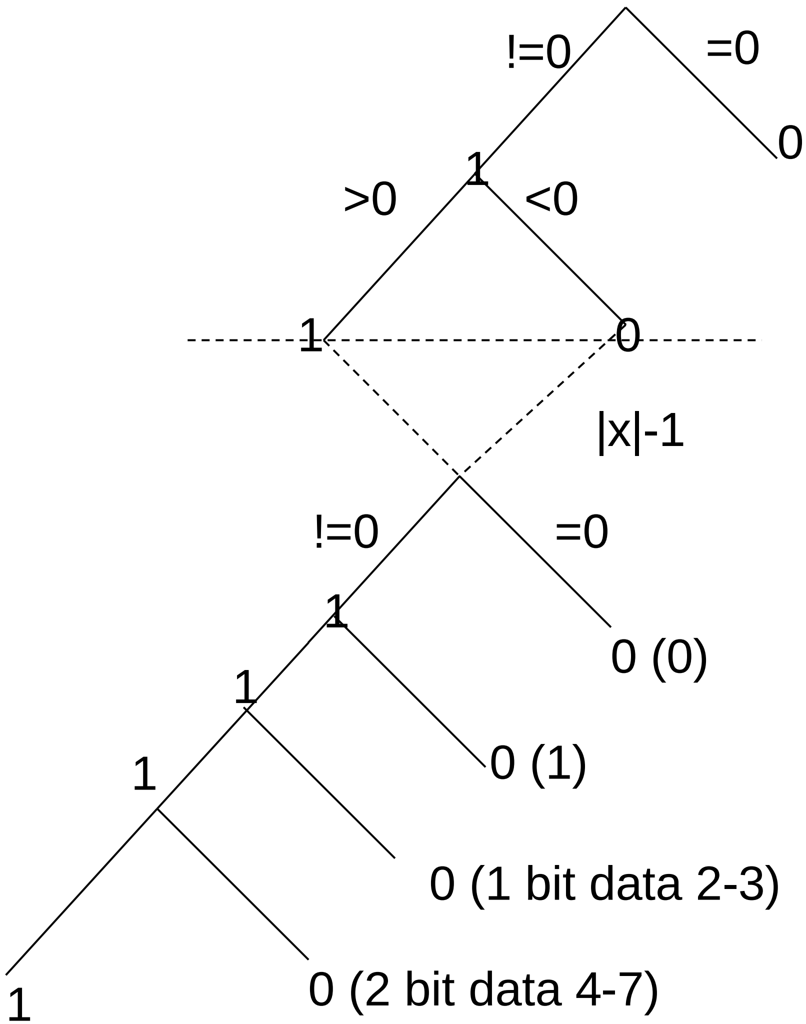
```
if encoder receives LPS
{
    A=A-qc;
    if A>=qc
    {C+=A; A=qc;}
    A<<=1;
    qc changes its state according to Column 5 in Table 1;
    // EX: qc=32B4(state08) and changes to 3C3D(state06)
    // because column 5 indicates decreasing 2*/
    encoder outputs MSB of C;
    C<<=1;
}
```



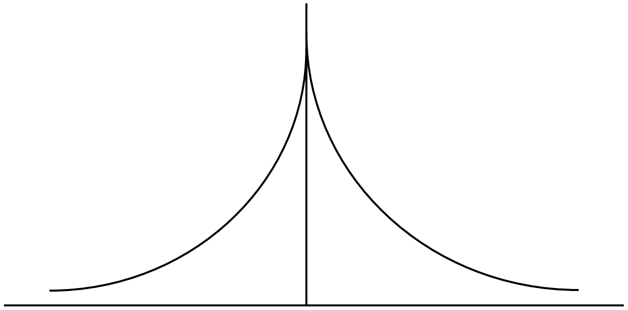
Non-Binary Data

- Redefine the symbol set s.t. more inter-pixel redundancy can be exploited effectively
- The gray level of an m-bit image can be represented as $a_{m-1}2^{m-1} + a_{m-2}2^{m-2} + \ldots + a_1 2^1 + a_0$, $a_i = 0$ or 1, $0 <= i < m$
- Thus the coefficients of the above polynomial will form m bit-planes
- Disadvantage:
  - Example: gray level 127 (01111111), 128 (10000000)
- Improvement: preprocess the image by an m-bit Gray code
  - $a_{m-1}, a_{m-2}, \ldots, a_1, a_0$
$g_{m-1}, g_{m-2}, \ldots, g_1, g_0$
$g_{m-1} = a_{m-1}$, $g_i = a_i \text{ xor } a_{i+1}$, $i = m-2, m-3 \ldots, 1, 0$
  - Example: Gray 127 (01000000), 128 (11000000)

Non-Binary Data

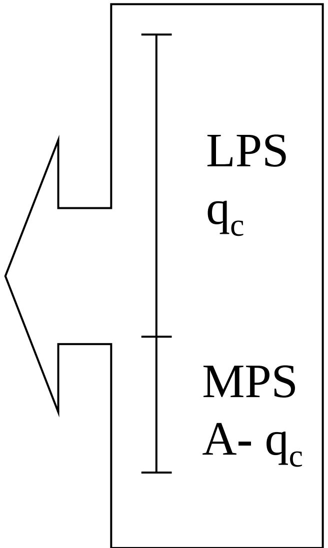


DATA	Binary Decision Tree
0*	0
0	1S0
1	1S10
2~3	1S110M
4~7	1S1110MM
8~15	1S11110MMM

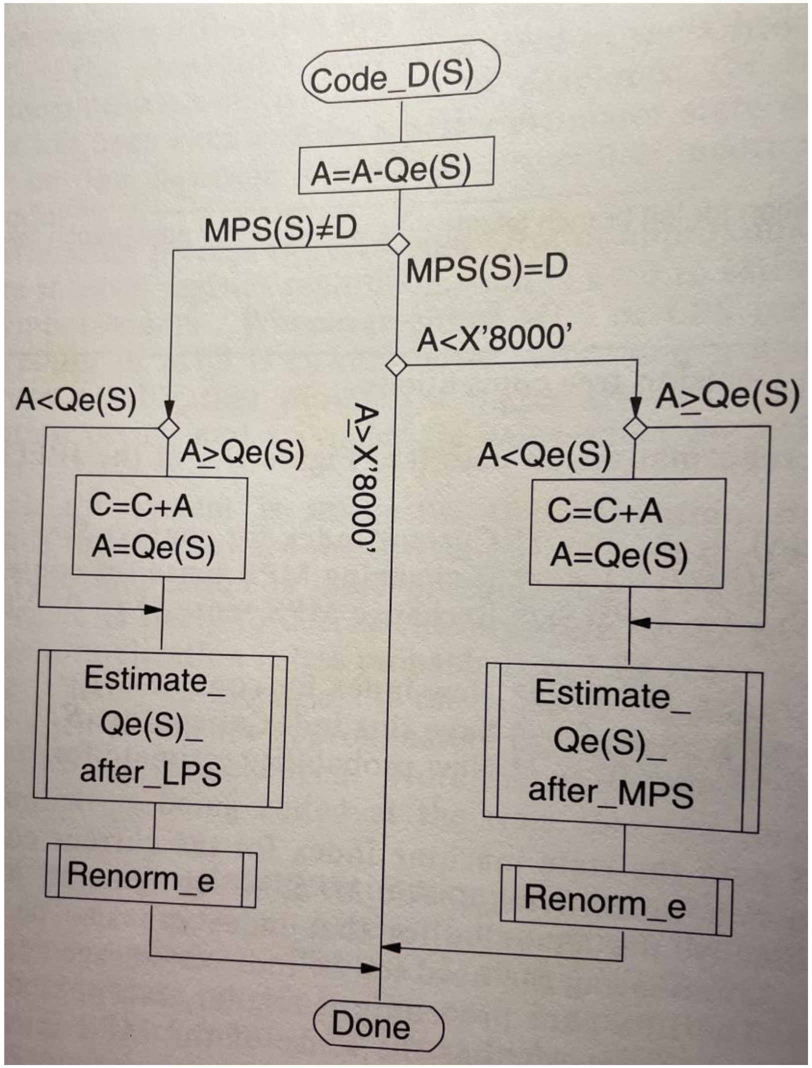


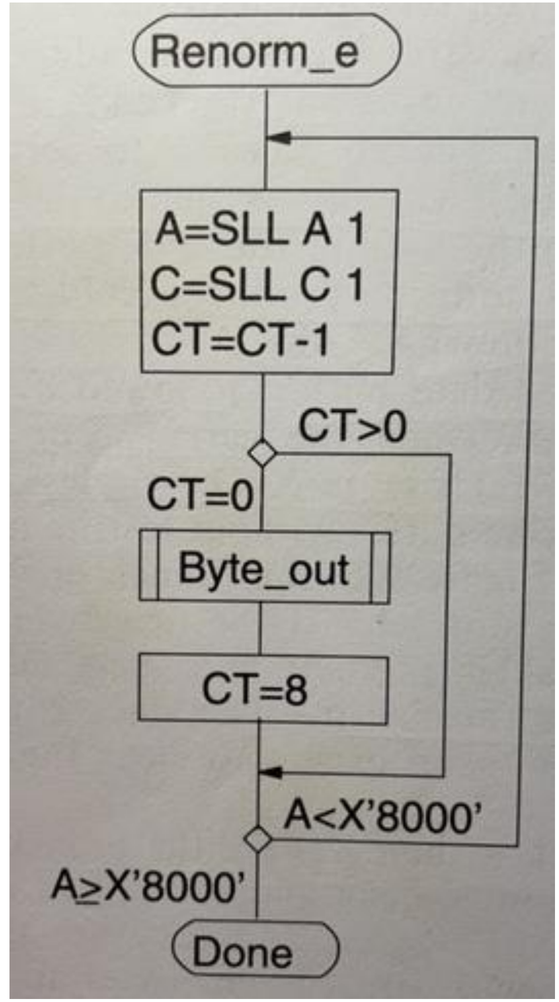
Summary of QM

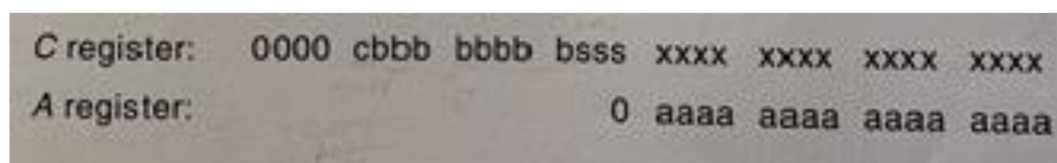
- Elimination of multiplication
- Left shift 1 bit for A & C whenever A < 0.75 (renormalization) 1.5 > A >= 0.75
- Generate new q_c (by table look-up or even with contexts) after each re-normalization
- Conditional exchange when the range of MPS is smaller than LPS ($A - q_c < q_c$)



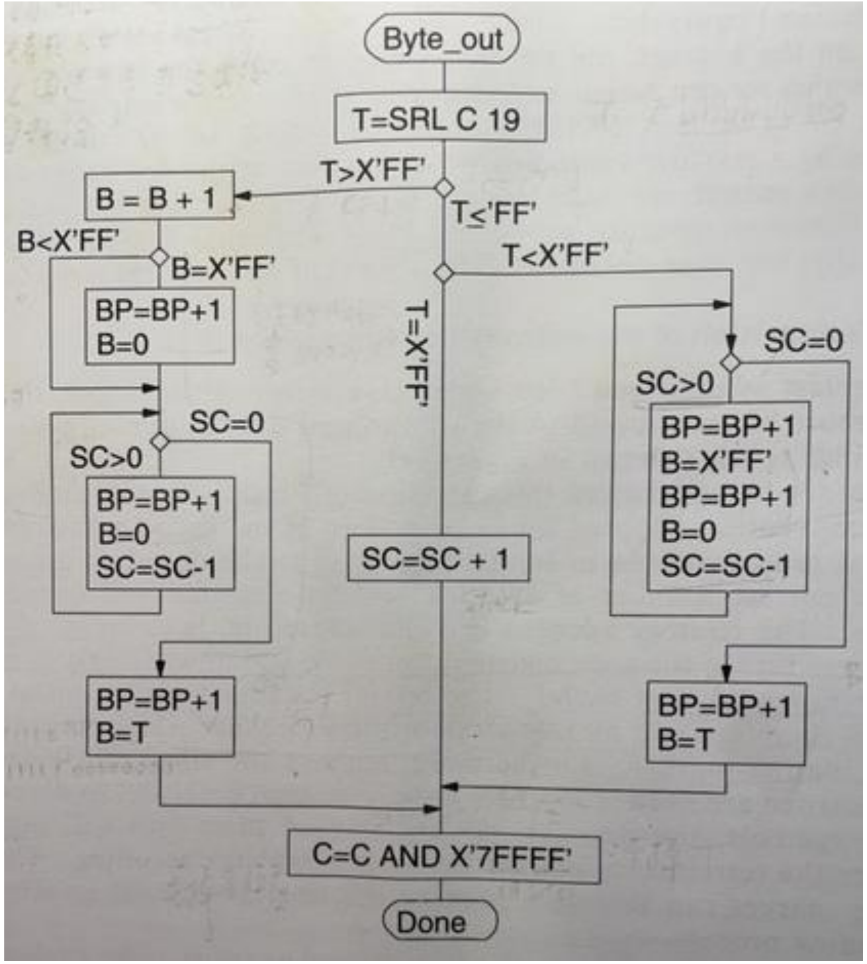
Implementation

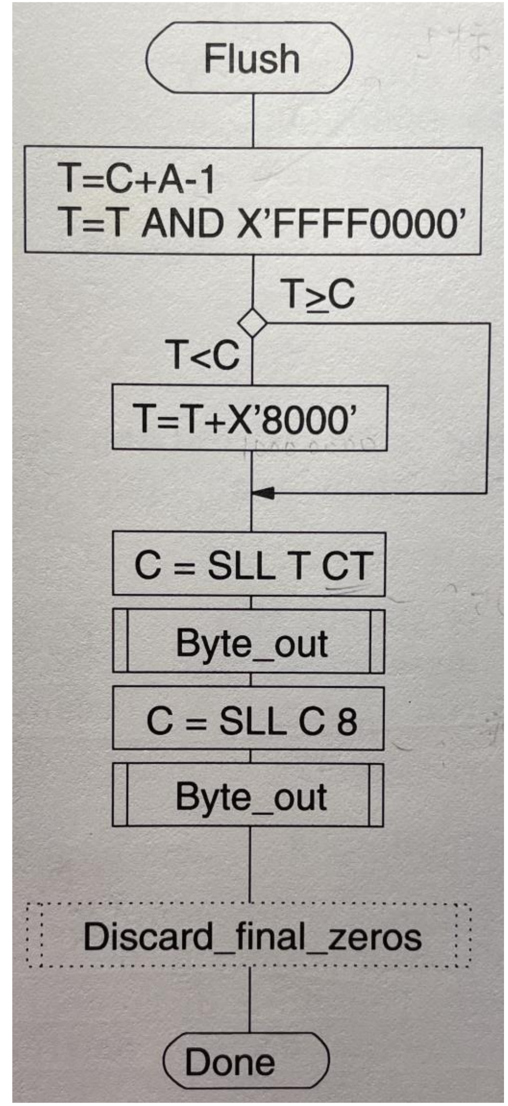






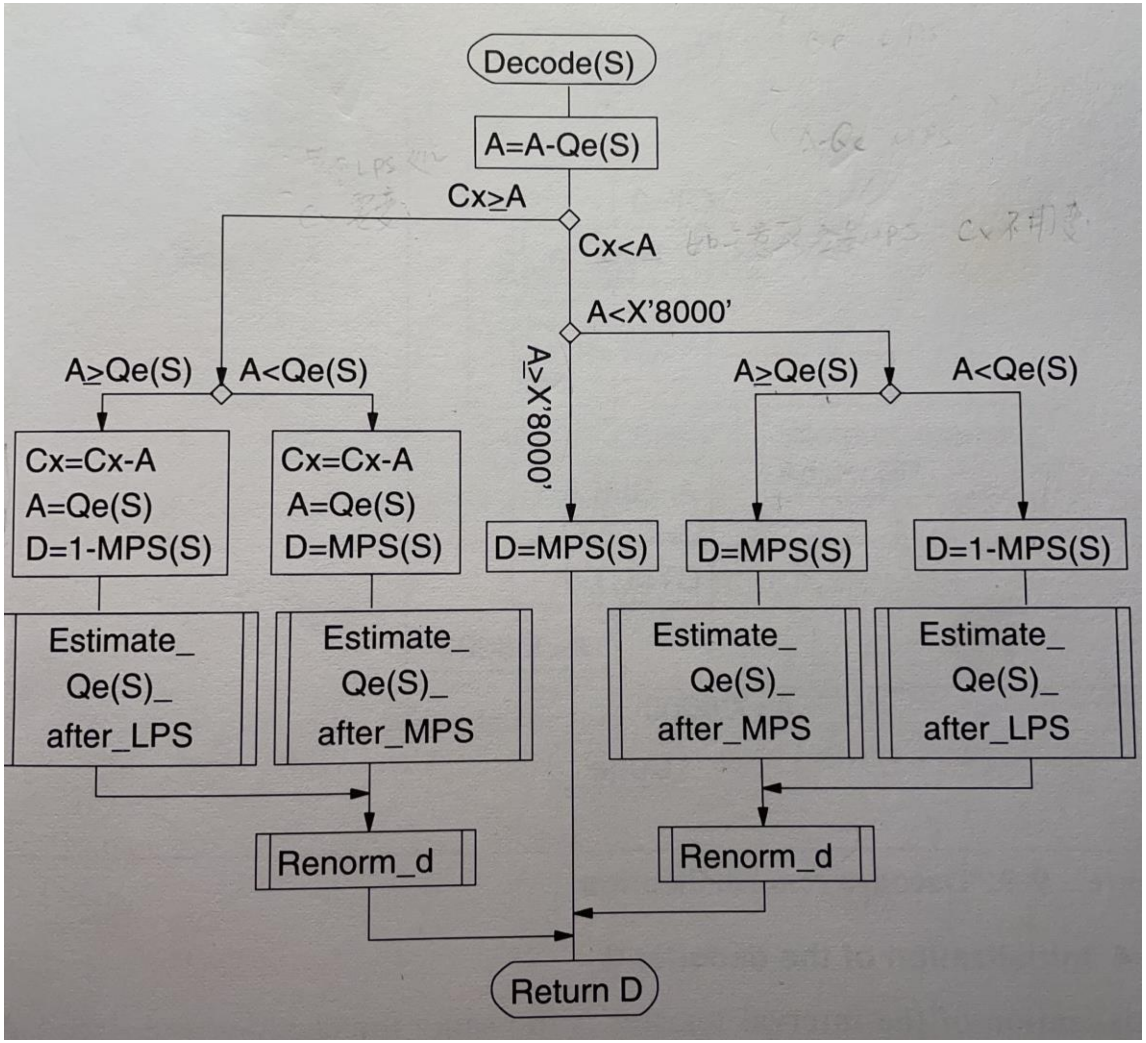
Implementation





C register:	0000	cbbb	bbbb	bsss	xxxx	xxxx	xxxx	xxxx
A register:				0	aaaa	aaaa	aaaa	aaaa

Implementation



Implementation