H.264/AVC를 위한 효율적인 Pipelined Arithmetic Encoder

윤재복, 박태근
가톨릭대학교 정보통신공학부
e-mail : *qjrl77@kornet.net, parktg@catholic.ac.kr*

An efficient Pipelined Arithmetic Encoder for H.264/AVC

Jae-Bok Yun, Tae-Geun Park
School of Information and Communication Engineering
Catholic University of Korea

Abstract

H.264/AVC에서 압축 효율을 향상시키기 위해 사용된 entropy coding중에 CABAC(Context-based Adaptive Binary Arithmetic Coding)은 하드웨어 복잡도가 높고 bit-serial 과정에서 data dependancy가 존재하기 때문에 빠른 연산이 어렵다. 본 논문에서는 adaptive arithmetic encoder와 정규화 과정을 효율적으로 구성하여 각 입력 심벌이 정규화 과정의 반복횟수에 관계없이 고정된 cycle에 encoding이 되도록 하였다. 제안한 구조는 pipeline으로 구성하기 용이하며, 이 경우 매 cycle에 한 입력 심벌의 encoding이 가능하다.

I. 서론

H.264/AVC는 ITU-T VCEG(Video Coding Experts Group)과 ISO/IEC MPEG(Moving Picture Experts Group)이 공동으로 개발한 최근 비디오 코딩에 대한 표준안이다[1]. H.264/AVC의 압출 효율을 향상시키기 위한 기술 중 하나가 entropy coding이다. H.264/AVC에 사용된 entropy coding 중에 CABAC은 MPS(Most Probable Symbol)와 LPS(Less Probable Symbol)의 확률을 이용하여 range(심벌이 차지하는 범위)와 low(심벌이 차지하는 범위의 하단 값)를 갱신하는 arithmetic coding에 현재의 MPS 심벌과 LPS의 확률을 syntax element에 맞게 context를 활용하고 확률 추정을 통해 압축 효율을 높였다[2]. CABAC은 slice header와 picture sequence를 제외한 모든 syntax elements에 대해 사용된다[1].

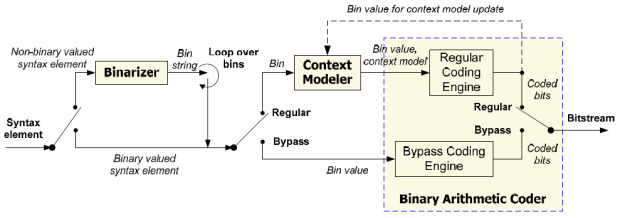

그림 1 CABAC encoding part 블록도

그림 1은 CABAC encoding part를 보여준다. CABAC encoding의 출력은 slice 단위로 이루어져서 하나의 slice의 encoding이 끝나면 마지막에 terminat- ion을 추가하여 전송한다[1]. 이때 하나의 slice는 많은 syntax element로 이루어졌다. 이 syntax element가 CABAC의 입력으로 들어간다. 이 data가 binarizer 과정이 필요한지 아닌지를 검사하여 이진 데이터로 변환하여 BAC(Binary Arithmetic Coding)를 한다. CAB-

본 연구는 한국과학재단 특정기초연구(R01-2005-000-11054-0) 지원으로 수행되었음

AC의 arithmetic coding은 Q-coder와 관련이 있다[3]. 이때, syntax element가 '0'과 '1'의 빈도가 비슷한 데이터라면 확률을 이용하지 않는 간결한 Bypass Coding을 한다[1].(syntax element의 특성에 따라 정해져있음.) 다시 말해, CABAC encoder는 binarizer와 context adaptive arithmetic encoder 이렇게 2부분으로 나눌 수 있다. 이중 context adaptive arithmetic coder에서 context를 활용하는 것은 메모리를 이용한 LUT 방식을 많이 사용하기 때문에 상대적으로 단순하지만, arithmetic coding 부분은 정규화 과정의 반복 횟수와 갱신된 Range와 Low 값을 바로 얻어내는 것이 간단하지 않고, 알아낸다 하더라도 정규화 과정의 반복에 따라서 출력 비트를 내보내는 것 또한 쉽지 않다. CABAC engine에 대한 최근 두개의 논문은 [4]과 [5]이다. 논문 [4]에서 보여주는 정규화 과정의 방법은 QM-coder 알고리즘에 기초한다. 이 방법은 복잡한 문제가 있는 "bitsOutstanding(counter)"을 정교하게 다루지 못한다. 논문 [5]는 정규화 과정에서 나타날 수 있는 각 상태를 분석하고 완전한 출력을 내는 부분을 parser block에서 효율적으로 처리하고 있다. 본 논문에서는 adaptive arithmetic encoder와 정규화 과정을 효율적으로 구성하여 각 입력 심벌이 정규화 과정의 반복횟수에 관계없이 고정된 cycle에 encoding이 되도록 하였다. 제안한 구조는 pipeline으로 구성하기 용이하며, 이 경우 매 cycle에 한 입력 심벌의 encoding이 가능하다. II 장에서는 CABAC의 arithmetic encoding 과정을 알아보고, III 장에서는 data dependancy를 효과적으로 처리하여 pipeline으로 구성 가능한 adaptive binary arithmetic encoder를 제안하고 구현한다.

II. CABAC의 Arithmetic Encoding

Binarizer 과정을 거친 syntax element는 arithmetic encoding을 한다. 이때 총 459개의 context 중, 각 syntax element마다 특정범위를 사용한다[1]. context는 1 비트의 MPS 심벌('1' 또는 '0')과 LPS의 상태를 나타내는 6 비트의 인덱스로 구성되어 있다[1]. syntax element가 context의 특정범위에서 초기화 되면 LPS의 상태와 MPS 심벌을 구할 수 있다. 이 두 값과 현재의 입력 심벌(binVal)이 입력으로 주어지면 그림 2와 같이 adaptive binary arithmetic coding을 하며, LPS의 확률 값과 현재의 MPS 심벌이 바뀌면서 context가 갱신된다. CABAC의 arithmetic coding은 전통적으로 arithmetic coding의 문제점인 range가 감소하는 문제를[6] 정규화 과정으로 해결하였고, MPS의 range가 LPS의 range 보다 작아지는 interval inversion 문제를[6] MPS 심벌을 바꿈으로써 해결하였고, 출력을 바로 해결할 수 없는 경우를 bitsOutstanding을 사용, 다음 출력이 결정되면 처리될 수 있도록 하여 효율적으로 처리되도록 구성되었다[1].

CABAC arithmetic coding에서 압축원리는 input symbol이 '1'이든 '0'이든 한쪽이 더 자주 입력되면 MPS의 확률이 더 크게 갱신되고 정규화 과정이 적게 실행됨으로써 출력 비트가 줄어들게 된다.

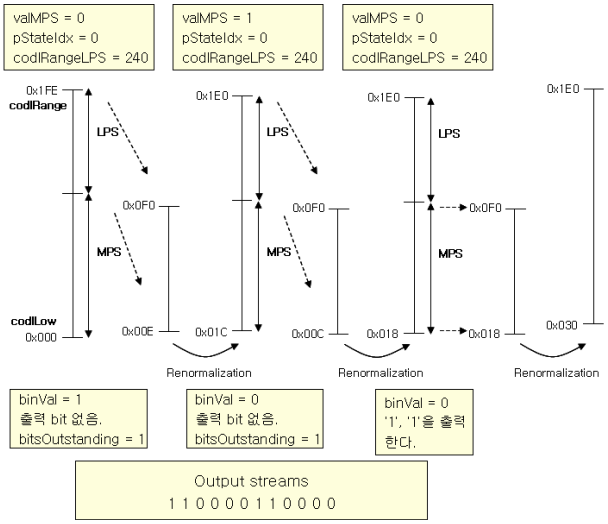

그림 2 CABAC의 Arithmetic Encoding 과정

그림 2는 '1', '0', '0' 3 비트를 encoding한 예이다. 초기 pStateIdx는 '0'이고, valMPS(현재의 MPS 심벌)는 '0', codILow(심벌이 차지하는 범위의 하단 값)는 0x000, codIRange(심벌이 차지하는 범위)는 0x1FE 로 시작한다. 처음 비트 '1'은 현재의 valMPS가 '0'이므로 LPS이고 전체 range에서 LPS가 차지하는 range가 다음 전체 range가 된다. 이때 range >= 0x100를 만족할 때 까지 정규화(1 비트 shift-left) 과정을 거친다. 이때, codILow 값에 따라서 출력을 내보내고(codILow가 0x100보다 작으면 '0'을 출력, 0x200보다 같거나 크면 '1'을 출력, 그 사이면 bitsOutstanding을 1 증가시킨다.), codILow와 codIRange가 동일한 효과가 있어야 하므로 둘 다 정규화한다. 이런 과정을 binarizer 과정을 거친 syntax element의 이진 입력 심벌에 반복적으로 적용하고, 하나의 slice의 모든 syntax element의 encoding이 끝나면 출력은 정규화 과정에서 내보낸 출력과 termination을 붙이면 하나의 slice에 대한 encoding 출력이 되고, 다음 slice에 대해 binarizer 과정을 거친 syntax element가 들어오면 좀 전의 과정을 다시 처음부터 수행하여 또 다른 slice에 대한 encoding 출력을 만들게 된다.

위의 encoding 과정을 보면 정규화 과정의 반복횟수

가 고정적이지 못하며, 다음 입력 심벌(binVal)을 encoding 하기 위해서는 갱신된 codILow와 codIRange가 있어야하는데, 이 값 또한 정규화 과정이 종료되어야 알 수 있다. 또한 출력을 결정하는 부분에서도 bitsOutstanding(아직 출력이 결정되지 않은 비트)이 있어서 단순하지 않다.

III. 제안한 CABAC의 Arithmetic Encoder

제안한 CABAC Arithmetic Encoder의 전체 블럭도는 그림 3과 같다. Adaptive Arithmetic Encoder와 Output Generator 2 부분으로 구성된다. 그림 4와 5는 Renormalization Unit에 포함되었고, 그림 6은 Bit Arrangement Unit에 포함되었다. Termination Unit은 하나의 slice의 모든 Syntax element의 encoding이 끝나면 남아있는 Termination을 붙이는 부분이다. Adaptive Arithmetic Encoder에서는 LPS가 range에서 차지하는 범위를 구하는데 있어서 곱셈 연산을 없애기 위해 ROM을 이용하여 구한다[1]. 그림 6의 Output Generator는 정규화 과정의 반복횟수에 관계없이 고정된 cycle에 하나의 입력 심벌에 대한 출력을 만들 수 있다.

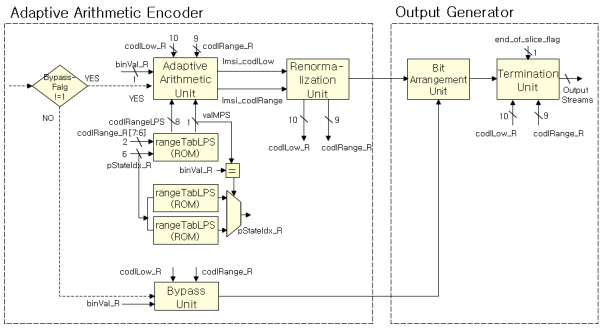

그림 3 제안한 CABAC Encoder

1. Renormalization Unit 구조

갱신된 codIRange는 그림 4과 같은 로직[7]을 사용하여 codIRange 9 비트 중에서 상위 비트로부터 '1'이 몇 번째 있느냐로 알아낼 수 있다. 그림 4의 L은 정규화 과정의 반복 횟수가 될 것이고, 갱신된 codIRange는 L만큼 shift-left하면 직접 구할 수 있다. codIRange는 최소 값이 '2'이기 때문에[1] 그림 3과 같이 상위 8 비트만 검사하면 된다.

갱신된 codILow는 정규화 과정에서 출력을 내는 것을 고려하지 않고 결과만을 직접 구한다면, 그림 5와 같이 구한다. 정규화 과정 중에 버려지는 비트는 고려하지 않으면 되고, L-1만큼 shift-left 한 후, 상위 2번

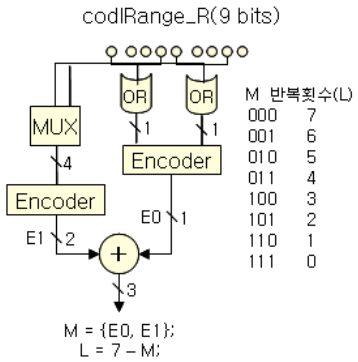

그림 4 정규화 과정의 반복 횟수를 구하는 로직

```
if(imsi_1_codILow[8] & 1 == 1)
  codILow_R <= imsi_2_codILow << 1;
else
  codILow_R <= imsi_1_codILow << 1;

imsi_1_codILow = imsi_codILow << L - 1;
imis_2_codILow = (imsi_1_codILow[8] & 1 == 1)
      ? (imsi_1_codILow & 10_1111_1111)
      : imis_1_codILow;
```

그림 5 갱신된 codILow 구하는 부분

째 비트가 1이면 그 비트만 반전하고[1] 한 번 더 shift-left 한 것이 최종 codILow가 된다.

2. Bit Arrangement Unit 구조

정규화 과정에서 codIRange에 따라서 정규화 과정의 반복횟수가 결정된다. 이 반복횟수 만큼 codILow도 정규화 되면서 출력 비트가 발생한다. 이때 완전히 결정된 출력비트만 발생하면 단순하게 출력 register에 저장만 하면 된다. 그러나 bitsOutstanding(아직 출력이 결정되지 않은 비트)이 있고[1], 이것은 다음 심벌의 encoding 출력에도 영향을 줄 수 있기 때문에 바로 처리할 수 없다. 따라서 그림 6와 같이 현재 심벌의 정규화 과정에서 출력을 결정 할 수 있는 부분과 다음 심벌의 정규화 과정에서 결정할 수 있는, 2부분으로 나누어서 처리할 수 있다. 먼저 한 심벌의 정규화 과정에서 나올 수 있는 최대 반복 횟수는 7번이다. 7번에 대한 출력 상태를 미리 만들 수 있다(2 비트로 '01'이면 완전한 출력 값 '1'이고, '00'혹은 '11'이면 완전한 출력 값 '0'이고, '10'이면 아직 출력을 결정할 수 없는 상태이다)[1]. 이때, 몇 번째 반복에서 결정할 수 없는

상태인지도 알아낼 수 있다. 이 두 정보를 가지고 현재 심벌의 정규화 과정에서 결정할 수 있는 출력을 정하여 임시 register에 저장하고, 현재 심벌에서 결정할 수 없는 부분(그림 6에서 붉은 점선 타원)은 다음 입력 심벌에서 결정할 수 있도록 bitsOutstanding에 저장한다. 이렇게 임시 register에 각 입력 심벌에 대한 결정된 출력을 저장하면서 output streams에는 각 입력 심벌의 정규화 과정이 진행되면서 결정된 출력을 저장하면, 출력을 만들 수 있다.

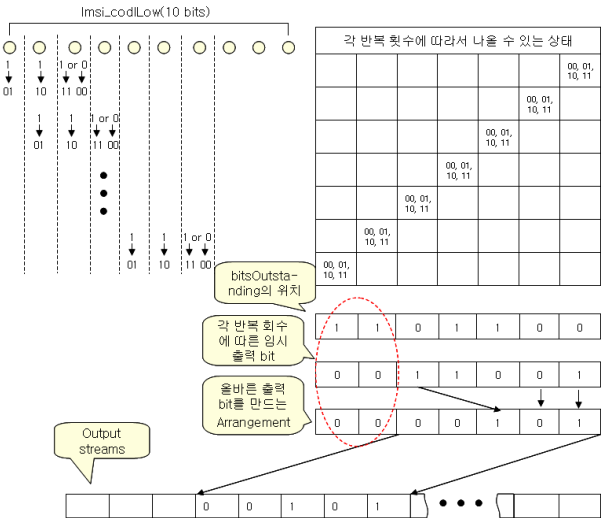

그림 6 Bit Arrangement Unit의 구조

IV. 결과 및 성능분석

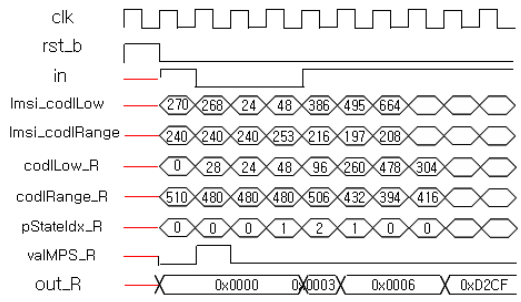

그림 7 제안한 Pipelined Adaptive Arithmetic Encoder의 타이밍 다이어그램

그림 7은 HDL로 코딩하여, '1', '0', '0', '0', '1', '1', '1', 7 비트를 입력 심벌로 준 경우의 타이밍 다이어그램이다. 본 논문에서는 context를 제외한 CABAC에서 사용되는 Adaptive Arithmetic Encoder 부분에 대한 설계이므로, pStateIdx는 '0', valMPS는 '0'으로 초기화한다. 제한한 구조에서 그림 6의 Adaptive Arithmetic Encoder 부분은 1 cycle에 끝나고, Output Generator 부분은 2 cycle에 고정적으로 끝남으로 timing 상 3단 pipeline 구조로 설계하여, 매 cycle에 입력 심벌을 encoding 할 수 있으며, 최종 출력은 마지막 입력 심벌이 입력된 후 2 cycle 후에 출력된다. 그리고 위에 제안한 구조는 각 블록이 고정 cycle에 끝남으로, 부가적인 control 블록이 필요 없다.

CABAC은 하드웨어 복잡도가 높고 bit-serial 과정에서 data dependancy가 존재하기 때문에 빠른 연산이 어렵다. 본 논문에서는 3장에서 설명한 방법을 이용하여 data dependancy를 제거하였고, CABAC의 Adaptive Arithmetic Encoder와 정규화 과정을 효율적으로 구성하여 각 입력 심벌이 정규화 과정의 반복 횟수에 관계없이 고정된 cycle에 encoding이 되도록 하였다. 제안한 구조는 pipeline으로 구성하기 용이하며, 본 논문에서는 한 cycle에 한 입력 심벌의 encoding이 가능하도록 pipeline으로 구현하고 검증하였다.

감사의 글

저자들은 본 연구를 위하여 설계 환경을 제공하여 준 IDEC(IC Design Education Center)에 감사드린다.

참고 문헌

[1] ITU-T Recommendation H.264: Advanced video coding, ITU, March 2004.

[2] D. Marpe, H. Schwarz, and T. Wiegand, "Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard", IEEE Transaction on Circuits and Systems for Video Technology, vol. 13, no. 7, pp. 620-636, July 2003.

[3] W. B. Pennebaker, J. L. Mitchel, G. G. Langdon jr., and R. B. Arps. "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder", IBM Jounal of Research and Development, 32(6):717-726, November 1988.

[4] R. Osorio, J. Bruguera, "Arithmetic coding architecture for H.264/AVC CABAC compression system", in Proc. Euromicro Symposium in Digital System Design, 2004, pp. 62-69.

[5] H. Shojania, S. Sudharsanan, "A High Performance CABAC Encoder", 2005 IEEE.

[6] D. Salomon, "Data Compression", Springer, 2004.

[7] W. Di, G. Wen, H. Mingzeng and J. Zhenzhou, "An Exp-Golomb Encoder and Decoder Architecture for JVT/AVS" , pp. 910-913, 2003 IEEE.