H.264/AVC용 CAVLC 디코더의 설계

정덕영* · 손승일**

Design of CAVLC Decoder for H.264/AVC

Duck Young Jung* · Seung Il Sonh**

이 논문은 2006년 정부(교육인적자원부)의 재원으로 학술진흥재단의 지원을 받아 수행된 연구임(KRF-2006-521-D00339)

요 약

디지털 비디오 압축 기술은 대역폭과 저장 공간이 제한되는 멀티미디어 데이터의 효율적인 전송과 저장을 가능하게 하는 중요한 역할을 해왔다. JVT가 제안한 새로운 비디오 코딩 표준인 H.264/AVC는 압축 성능에 있어서 이전의 표준들을 훨씬 능가하고 있다. 특히 비디오 및 이미지 압축 응용에서 가변길이 부호는 중요한 역할을 한다. H.264/AVC 표준은 엔트로피 코딩 방식으로서 CAVLC를 채택하였다.

H.264/AVC의 CAVLC는 많은 메모리 액세스를 필요로 한다. 이는 메모리 액세스 시에 상당한 전력을 소비하기 때문에 DMB와 비디오 폰 서비스와 같은 응용을 위해서는 심각한 문제가 될 수 있다. 본 논문에서는 이러한 문제점을 극복하기 위해, 산술연산에 근거한 메모리 사용없는 coeff_token, level 및 run_before 디코딩을 구현하고, total_zero 가변길이 디코딩시에만 필요한 메모리의 70%만 사용하여 구현하는 가변길이 디코딩 기법을 제안한다.

ABSTRACT

Digital video compression technique has played an important role that enables efficient transmission and storage of multimedia data where bandwidth and storage space are limited. The new video coding standard, H.264/AVC, developed by Joint Video Team(JVT) significantly outperforms previous standards in compression performance. Especially, variable length code(VLC) plays a crucial part in video and image compression applications. H.264/AVC standard adopted Context-based Adaptive Variable Length Coding(CAVLC) as the entropy coding method.

CAVLC of H.264/AVC requires a large number of the memory accesses. This is a serious problem for applications such as DMB and video phone service because of the considerable amount of power that is consumed in accessing the memory. In order to overcome this problem in this paper, we propose a variable length technique that implements memory-free coeff_token, level, and run_before decoding based on arithmetic operations and using only 70% of the required memory at total_zero variable length decoding.

키워드

H.264/AVC, VLC, CAVLC, Compression

* C&S 테크놀로지 반도체연구소
** 한신대학교 정보통신학과

접수일자 : 2007. 3. 6

Ⅰ. 서 론

현재 영상과 디지털 컨텐츠에 대한 관심이 높아지고 있다. 이에 많은 데이터를 빠른 시간에 전송해야하는 문제가 발생하게 되면서 압축에 대한 관심도 높아졌다.

이런 영상 압축표준 중에 하나이며 현재 지상파 DMB와 카메라 핸드폰 그리고 각종 영상에서 표준으로 사용하고 있는 H.264는 기존의 어떠한 비디오 압축 표준보다 획기적인 화질 개선 수단의 제공을 목적으로 개발되었고, 움직임 예측 및 보상의 경우를 살펴보면 모양과 크기가 다른 블록들을 제공하고, 1/4 픽셀(pixel) 정밀도의 움직임 보상을 지원하므로 정수 픽셀 단위의 움직임 보상보다 좋은 압축효율을 제공한다. 또한 압축 기법에서 사용되는 VLC(가변 길이 코딩)는 MPEG 비디오 및 이미지 코딩 응용에서 매우 중요한 역할을 한다[1,2]. VLC의 핵심 개념은 평균 코드워드 길이(Average Codeword Length)를 최소화하는 것이다. 짧은 코드워드는 자주 발생되는 데이터에 할당하고, 긴 코드워드는 자주 발생되지 않는 데이터에 할당하여 데이터 압축을 수행한다. 그런데 데이터 압축률을 한층 증가시키기 위해 H.264/AVC에서는 4x4(혹은 2x2) 블록으로 구성된 변환계수의 데이터를 부호화하기 위해 CAVLC를 채택하고 있다. 특히 H.264/AVC의 실시간 처리 요구는 CAVLC의 전용 하드웨어 구현을 필수적으로 요구하고 있다[3,4].

CAVLC 디코더는 일반적으로 Coeff_token 디코더 블록, Trailing ones 디코더 블록, 레벨 디코더 블록, Total_zeros 디코더 블록, Run_before 디코더 블록 등으로 구성되어 있다. 특히 H.264/AVC용 CAVLC 디코더를 구현함에 있어 각 단계별로 많은 테이블 룩업(Table Look-up)을 수행하게 된다. Coeff_token 블록의 경우 64개의 엔트리를 갖는 4개의 테이블 룩업과 14개의 엔트리를 갖는 1개의 테이블 룩업을 수행하게 된다. Total_zeros 블록의 경우 최대 16개의 엔트리를 갖는 가변 엔트리의 15개 테이블 룩업을 수행해야한다. Run_before 블록은 최대 15개의 엔트리를 갖는 가변 길이 엔트리의 7개 테이블 룩업을 수행해야 된다[5-7].

따라서 CAVLC 구현 시 많은 메모리와 메모리 참조가 요구되고 칩 크기가 증가되는 문제점이 발생한다. 그러므로 본 연구에서는 최대한 메모리 액세스를 줄이면서도 간단히 구현이 가능한 알고리즘을 개발하여 이를 FPGA로 구현 및 검증함으로써 H.264/AVC 모듈에서 핵심 소자로 사용할 수 있도록 설계하였다.

Ⅱ. 관련 연구

Wu Di 등이 2003년 발표한 논문에 따르면 CAVLC에서 사용하는 모든 테이블을 저장하고 전송된 비트스트림 데이터에서 최초 1을 만나기 전까지의 0의 개수에 따라 테이블 내의 데이터를 선택하여 최초 1 이후의 데이터를 비교하여 수행하였다[8]. 모든 테이블을 저장하여 수행하므로 테이블을 저장하기 위해 칩 면적이 커지는 단점이 있지만, 메모리를 모두 접근하지 않고 최초 1을 만나기 전까지의 0의 개수에 따라 접근하기 때문에 접근 횟수는 테이블을 이용하여 수행하는 것보다 접근 횟수를 줄일 수 있다.

2005년 발표된 CAVLC의 효율적인 디코딩 기법에서는 coeff_token단계에 대해 간단한 식을 이용하여 전송된 비트스트림에 대해 0의 개수를 구하고 규정된 조건에 따라 일부는 메모리를 참조하고 일부는 테이블을 참조하여 수행하도록 하였다[9]. 이로 인해 저장해야 하는 메모리를 줄이고 그에 따른 메모리 접근 횟수도 줄일 수 있다. 하지만 논문에서 규정한 조건에 만족하지 못할 경우가 자주 발생할 경우 그에 따른 메모리 접근이 늘어나게 되고, 일부 테이블을 저장해야 하기 때문에 그만큼의 메모리는 존재해야 한다. 또한 run_before단계에서는 run_before에서 사용하는 7개의 테이블에 대해 논문에 정해진 상태에서 사용하는 알고리즘을 적용하면 모든 경우는 테이블 없이 복호화가 가능하다. 그리고 하드웨어 설계할 때 곱셈기를 하나만 사용하여 coeff_token을 설계할 때 보다 칩 면적은 적게 들지만 각각의 상태에서 적용되지 않는 부분이 발생하는 단점이 있다.

2005년 Guo 등이 발표한 고성능 CAVLC VLSI 아키텍처에 따르면, coeff_token단계에 대해 높은 발생확률의 코드워드들을 모아서 테이블화하고, 나머지 낮은 발생확률의 코드워드는 다른 테이블로 나누어 ROM으로 저장하고 발생확률이 높은 것부터 활성화하여 수행하므로 테이블을 메모리에 저장하는 칩 면적은 변함이 없지만, 메모리 접근 횟수를 줄일 수 있다[10]. 하지만 발생확률이 낮은 테이블에 대한 접근이 늘어날 경우 메모리 접근 횟수가 많이 발생하게 되어 모든 테이블을 저장하고 수행해야 하기 때문에 최악의 경우 모든 테이블을 저장

하여 사용할 때와 동일한 수행을 하게 된다. 이 논문의 핵심 구성은 HLLT(Hierarchical logic for look-up table)과 PCCF(Partial combination component Freezing) 부분으로 이루어져 있다. 각각의 VLC 테이블은 최대 65536번 들어가지만, 여기서 제안한 방식에서는 오직 62번만 들어간다. 또한 run_before 단계에서는 전체 테이블에서 오직 45번만 들어가며, ZTEBA(Zero-left table elimination by arithmetic) 방법을 적용하여 설계하였다[10].

2004년 Amer 등이 발표한 논문에서는 테이블에서 사용되는 코드의 확률을 이용한 CAVLC 알고리즘을 제안하였다[11]. Coeff_token단계에서는 4개의 테이블에 대해 각 코드의 발생확률을 구하고 발생확률이 높은 코드는 테이블을 참조하지 않고 알고리즘을 적용하여 수행하며, 발생확률이 낮은 것은 테이블화하여 구현하였다. 발생확률이 낮은 것만 저장하므로 그만큼의 칩 면적과 메모리 접근 횟수를 줄일 수 있으며, 발생 확률과 동일할 경우 메모리 접근 없는 수행도 가능하다. 하지만 일부 메모리를 저장해야 하므로 그에 대한 칩 면적을 차지하게 되며, 최악의 경우 발생확률과 다르면 모든 메모리를 접근해야하는 경우가 발생하게 된다. 또한 run_before단계에서는 total_zero 값이 0이 될 때까지 각 0이 아닌 계수의 run_before를 테이블을 전혀 사용하지 않고 FSM(Finite State Machine)을 이용하여 복호화 함으로써 run_before 복호화에 필요한 메모리 접근을 없앨 수 있다[12].

Ⅲ. CAVLC 디코딩 개요

CAVLC 디코딩은 인코더와 같은 단계를 동일한 순서로 디코딩을 수행한다.

3.1 coeff_token 단계

4x4블록의 '0'이 아닌 계수의 개수와 trailing_ones의 개수를 의미한다. coeff_token의 부호화는 4개의 테이블 중에서 이웃 블록(이전 블록과 윗 블록)들의 total_coeff 개수에 따라 한 개의 테이블을 선택하며, 그 코드 테이블과 코드에 해당하는 코드 길이 테이블로 구성된 2개의 테이블이 있어 복호화 시에는 코드 길이 테이블의 데이터만큼의 입력 비트열을 읽었을 때 똑같은 코드 테이블 값이 나올 경우 이때의 테이블 인덱스 값을 total_coeff와 trailing_ones 값으로 복원한다[2,4,12].

3.2 trailing_ones 단계

coeff_token에서 얻은 trailing_ones 값에 따라 한 비트씩 읽어 0이면 -1, 1이면 1로 복원한다[2,4,12].

3.3 level 단계

0이 아닌 계수를 부호화 하는 곳으로 각 레벨에 대한 코드는 접두사(최초 1을 만나기 전까지의 0의 개수)와 접미사로 구성되며, 접미사의 길이는 0~6비트사이이다. 이를 처음에 total_coeff가 10보다 크고 trailing_ones가 3보다 작으면 1로 시작하고 그렇지 않으면 0으로 접미사를 시작하여 수행한다. 복호화는 최초로 1을 만나기 전까지의 0의 개수와 최초 1 다음부터 접미사의 길이만큼의 데이터를 이용하여 복원한다[2,4].

3.4 total_zeros 단계

복원될 4x4 블록에서 지그재그 스캔하고 역으로 최초로 0이 아닌 계수부터 마지막 계수 전까지의 0의 개수를 복원하는 부분이다. 그 수행은 coeff_token에서 얻은 total_coeff 값과 코드 테이블과 그 코드에 해당하는 코드 길이 테이블로 구성된 2개의 테이블에서 코드 길이 테이블의 데이터만큼의 입력 비트열을 읽었을 때 똑같은 코드 테이블 값이 나올 경우 이때의 테이블 인덱스 값이 total_zeros값이 된다[2,4,12].

3.5 run_before 단계

run_before는 지그재그 스캔의 역 순서에서 각각의 0이 아닌 계수 앞에 있는 0의 개수를 의미한다. 7개의 테이블을 사용하여 부호화하는데 복호화는 total_zero에서 구한 0의 개수에 따라 7개의 테이블 중에서 선택되며 그 테이블 내의 값과 코드 길이 테이블 및 입력 비트열과 비교하여 같은 값이 나왔을 경우 그 때의 테이블의 위치 값이 run_before값이 된다. 수행의 종료는 total_zero의 개수가 0이 될 때 또는 total_coeff의 위치를 모두 찾았을 때 종료되며 그렇지 않으면 선택된 0 개수를 현재 0 개수에서 제외하고 다시 수행한다[2,4,12].

Ⅳ. 제안한 CAVLC 디코더 설계

4.1 CAVLC 디코더 블록도

CAVLC 디코더는 인코더에서 처리한 순서와 동일한

순서로 복원하며, 이는 coeff_token, trailing_ones, level, total_zeros, run_before의 총 5단계로 이루어져있다. 그 중에서 테이블을 사용하여 부호화한 coeff_token과 total_zeros 그리고 run_before는 복호화시에도 테이블을 사용하여 복원해야 한다. 이렇게 테이블을 사용하여 복원할 경우 사용할 테이블을 메모리로 저장하고 있어야 하며, 각 단계별로 메모리 접근이 발생하고 최악의 경우 선택된 메모리에서 모든 경우를 다 수행해야 하기 때문에 성능저하가 발생하게 된다. 이에 본 논문에서는 테이블의 규칙성을 찾아 각 단계에서 사용되는 테이블 중에서 total_zeros에서 사용하는 테이블의 70%만 사용하여 수행하는 알고리즘을 제안하였으며, 이를 적용하여 CAVLC 디코더를 설계하였다. 그림 1은 CAVLC 디코더의 전체 블록도를 보여주고 있다.

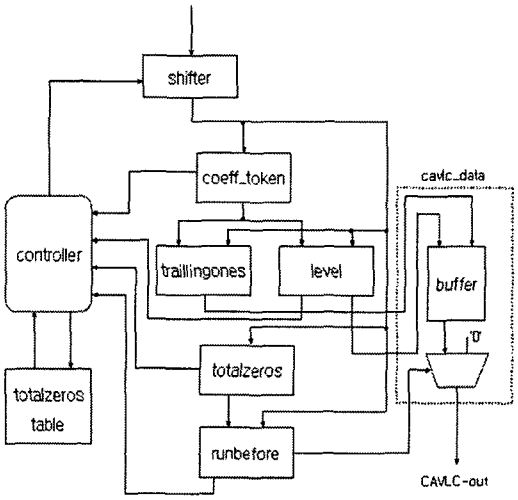

그림 1. CAVLC 디코더의 블록도
Fig. 1. Blockdiagram of CAVLC decoder

CAVLC 디코더의 수행은 쉬프터를 통해 16비트단위로 전송받은 비트스트림 데이터에서 최초 1을 만나기 전까지의 0의 개수를 구하고, 그 이후부터 0의 개수와 함께 coeff_token으로 전송한다. 이 단계에서 수행된 결과에 따라 trailing_ones단계와 level단계 그리고 total_zeros단계를 수행하며, trailing_ones단계와 level단계에서 처리된 결과를 버퍼에 저장하여 최종 복원된 CAVLC 데이터로 처리할 때 사용된다. 마지막 단계인 run_before단계는 total_zeros단계의 결과에 따라 수행된다. 최종 복원된 CAVLC 데이터는 run_before단계에서의 처리결과에 따라 버퍼에 있는 데이터 또는 0을 출력한다. 그리고 이 모든 단계의 처리에 관한 제어는 제어기(Controller)에서 처리한다.

4.2 쉬프터

쉬프터의 valid가 활성화되면 16비트로 전송된 비트스트림 데이터를 먼저 buffer1과 buffer2에 저장한다. 저장된 데이터를 32비트로 확장하여 첫 번째 1을 만나기 전까지 0의 개수를 구하고, 구해진 0의 개수와 최초의 1을 만난 부분부터 16비트 데이터를 전송한다. 포인터의 값은 모듈로 16으로 동작하는데, 16을 넘으면 carry_out이 발생한다. 이때 buffer2의 데이터가 buffer1으로 전송되며 buffer2에는 처리 대기중인 입력된 데이터를 저장한다. 이러한 과정은 블록 데이터에 대한 디코딩을 완료할 때까지 이루어진다.

모든 단계의 수행이 완료되고 복원된 CAVLC 데이터가 모두 전송되면 쉬프터의 모든 값은 리셋되며 쉬프터의 valid가 활성화 되면 다시 수행을 시작한다. 그림 2는 본 논문에서 설계된 쉬프터의 블록도이다.

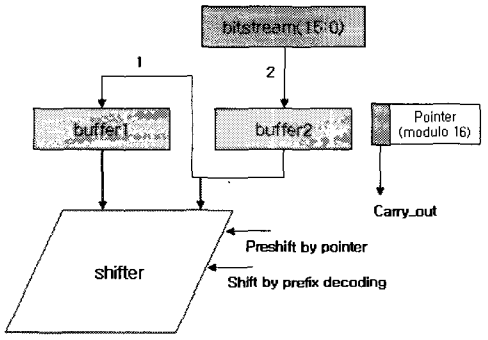

그림 2. 쉬프터 블록도
Fig. 2. Shifter blockdiagram

4.3 coeff_token

본 논문과 관련하여 향후에 나타는 용어를 간략하게 요약하면 다음과 같다. Lod_num[3:0]은 최초 1을 만날 때까지 0을 포함한 최초 1까지의 개수를 의미하며, remainin_bit[1:0]은 최초 1이후의 유효한 코드워드 비트를 의미한다. 따라서 전체 코드워드 비트 수는 "lod_num+ remaining_bit"의 결과와 같다. H.264/AVC의 표준은 coeff_token 값을 디코딩하기위해 4개의 VLC

ROM 테이블을 사용하도록 하였는데, 이는 VLC0, VLC1, VLC2 및 VLC3으로 구성되어 있다. 본 논문에서는 이러한 테이블에 대한 분석을 바탕으로 수학적 연산으로 해결할 수 있는 규칙성을 도출하였으며, 이를 표1부터 표4까지 요약하여 정리하였다.

표 1. 테이블을 사용하지 않은 VLC0 알고리즘
Table 1. Table-free VLC0 algorithm

Lod_num[3:0]	Remaining _bits[1:0]	Total_coeff[4:0]	Trailing_ones[1:0]	사용된 전체 비트수 t_bits[4:0]	i[2:0]
0(1)	0	lod_num	0	Lod_num + remaining_ bits	선행 1 이후의 비트 값 i[2]i[1]i[0]
1(01)	0		1		
2(001)	0		2		
3(000011,000100, 000101)	1(2) 2	3(2) 1	3(1) 0		
4(0000101,000011, 0000100)	2, (1), 2	3 (4) 5	2 (3) 3		
5 – 8	2	i[2:1]=0 → lod_num+1 i[2:1]!=0 → lod_num=-i[2:1]	3-i[2:1]		
9	3	i[2:0]=4 → 10 else i[1:0]=1 → lod_num - i[1:0] else i[1:0]=3 → 7 else lod-num-b[2]-i[1:0]	3 – i[1:0]		
10 – 12	3	i[1:0]=3 → lod_num=i[2]+(lod_num-9) i[1:0]=2 → lod_num < 12 then i[1:0]=3서 사용 else i[1:0]=1서 사용 i[1:0]=1 → lod_num=(i[2]+(lod_num-8) i[1:0]=0 → lod_num=(i[2]+(lod_num+7)	3 – i[1:0]		
13	2	i[2:1]=3 → lod_num + 2 i[2:1]!=3 → lod_num + 3	3-i[2:1] 예외 i[2:1]=0 → 0		
14	0	13	1		

표 2. 테이블을 사용하지 않은 VLC1 알고리즘
Table 2. Table-free VLC1 algorithm

Lod_num[3:0]	Remaining _bits[1:0]	Total_coeff[4:0]	Trailing_ones [1:0]	사용된 전체 비트수 t_bits[4:0]	i[2:0]
0(11)	0	0	0	Lod_num + remaining_bi ts	선행 1 이후의 비트 값 i[2]i[1]i[0]
0(10)	0	1	1		
1(011)	1	2	2		
1(0101, 0100)	2	3 (4)	3 (3)		
2(001011,001010, 001001,001000)	3	(1) 3 (3) 6	(0),1 (2) 3 3 – i[1:0]		
2(001110, 001111)	2	5, (2)	3, (1)		
3(0001...)~5(00000 001...)	2	Lod_num=5 && i[2:1]=0 → 5 else i[1:0]=0 → lod_num+4 i[1:0]=3 → lod_num-1 i[1:0]=1or2 → lod_num+1 Lod_num=6 && i[2:1]=0 → 9, i[2:1]=3 → 6 else 7	0 첫번째 조건을 제외하고는 3-i[1:0]		
7 – 9	3	i[1:0]=3 → lod_num – i[2] + (lod_num –6) i[1:0]=1 or 2 → lod_num-i[2] + (lod_num-5) i[1:0]=0 && lod_num=8 && i[2]=0 → 11 else lod_num-i[2]+4 + (lod_num/9)	3-i[1:0]		
10	2	i[2:1] = 2 or 3 → lod_num + 4	i[2:1]=2 → 2 i[2:1]=3 → 0		
10	3	i[2:1]= 0 or 1 → lod_num + 5	i[1:0]=1 →0 i[1:0]=0 or 3 → 1 i[1:0]=2 →2		
11	2	Lod_num + 5	3-i[2:1]		
12	0	Lod_num + 3	3		

coeff_token 단계에서는 0이 아닌 계수의 개수와 최초의 0이 아닌 계수가 ±1일 때 0을 포함해서 연속적으로 3개까지의 개수를 부호화하여 전송된 비트스트림 데이터를 테이블을 이용하여 복원하는 부분이다. coeff_token에서는 테이블이 4개 사용되며, 이를 설계에 사용할 경우 각각의 테이블을 실제 데이터 부분과 비트 크기를 저장한 테이블이 필요하다. 또한 흐름도 내에 "식i"로 표현된 것은 lod_num을 십진수로 읽은 값과 관련이 있다.

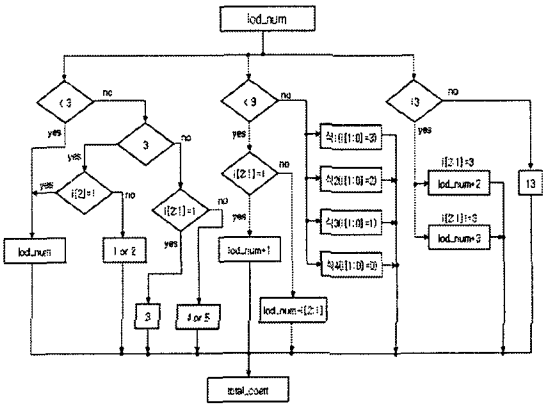

그림 3. 테이블을 사용하지 않은 VLC0 알고리즘의 흐름도
Fig. 3. Flow chart of table-free VLC0 algorithm

표 3. 테이블을 사용하지 않은 VLC2 알고리즘
Table 3. Table-free VLC2 algorithm

Lod_num [3:0]	Remaining _bits[1:0]	Total_coeff[4:0]	Trailing_ones[1:0]	사용된 전체 비트수 t_bits[4:0]	i[2:0]
0	3	i[2]=1 → i[1:0] else i[2]=0 → 4 + i[1:0]	3-i[1:0] i[2]=0 → 3	Lod_num + remaining_ bits	선행 1 이후의 비트 값 i[2]i[1]i[0]
1	3	i[2:0]=5 → 8 else i[2:0] < 5 → 5 – i[2:1] else i[2:0] > 5 → 5 – i[1:0]	i[0]=0 && i[2:1]=3 → 1 else i[2:0]=6 or i[2]=0 && i[0]=1 → 2 else i[2:0]=5 → 3		
2	3	i=0 → 3 / i=3 → 2 i=7 → 1 / i=4 → 9 i[2:1]=0 → 7 i[2]=1 && (i[1] xor i[0]) → 6	i[2:0]=0 → 0 3-i[1:0] i=4 → 3		
3	3	i=0 or 6 → 7 – i[1:0] , i=1 → 5, i=2 → 9, i=3 or 7 → 5 – i[2], i=4 → 10, i=5,8 → 8	i[2:0]=(0or1) or i[1:0]=3 →0, i[2:0]=6 →1 , i[2:0]=5 or 2→2 , i[2:0]=4→3		
4	3	12-i[1:0] - i[2]	3 - i[1:0]		
5	3	i[2]=1 → 13 – i[1:0] / i[2:0]=0 → 12 else i[2]=0 → 14 – i[1:0] = 13 – i[1:0] + b[2]	i[2:0]=0 → 0 else 3 – i[1:0]		
6	i[2:1]=3 → 2 Else 3	i[2:1]=3 → 13 else i[2:0]=0 → 15 else i[2:0]=5 → 13 else i=1,2,3,4 → 14	i[1:0]=1→ 0, i[1:0]=2→ 3, i[1:0]=3→2 (대체 식과 동일함) else 1		
7	2	i[2:1]=0 → 16 else 15	i[2:1]=2→ 3, i[2:1]=3→2 else i[1]		
8	1	16	i[2]=1 → 2, i[2]=0 → 3		
9	0	16	0		

표 4 테이블을 사용하지 않은 VLC3 알고리즘
Table 4. Table-free VLC3 algorithm

Lod_num[3:0]	Remaining _bits[1:0]	Total_coeff[4:0]	Trailing_ones[1:0]	사용된 전체 비트수 t_bits[4:0]	i[2:0]
Lod_num 값을 사용하지 않고 구현 가능함.	사용하지 않음	i[5:1]=1 → 0 else i[5:2] + 1	i[5:1]=1 → 0 else i[1:0]	Lod_num + remaining_bits + 6 비트로 고정됨	선행 1 이후의 비트 값 1i[2]i[1]i[0]

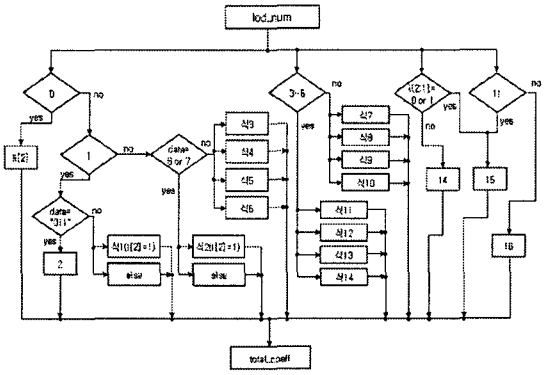

그림 4. 테이블을 사용하지 않은 VLC1 알고리즘의 흐름도

Fig. 4. Flow chart of table-free VLC1 algorithm

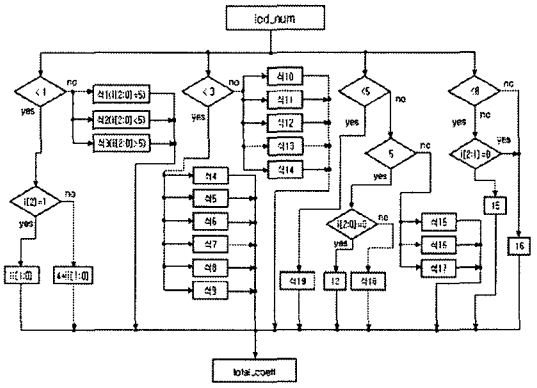

그림 5. 테이블을 사용하지 않은 VLC2 알고리즘의 흐름도

Fig. 5. Flow chart of table-free VLC2 algorithm

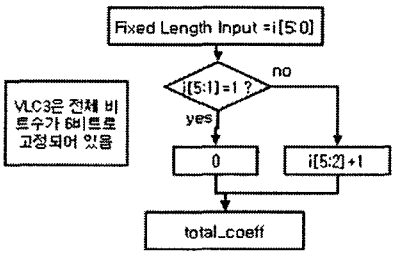

그림 6. 테이블을 사용하지 않은 VLC3 알고리즘의 흐름도

Fig. 6. Flow chart of table-free VLC3 algorithm

각각의 테이블은 62개의 코드워드를 가지고 있으며 456개를 저장할 수 있는 저장 공간이 필요하다. 이에 본 논문에서는 테이블에서 사용되는 코드워드에서 최초 1을 만나기 전까지의 0의 개수에 따라 간단한 식을 이용하여 테이블 없이 한 클럭에 total_coeffs와 trailing_ones를 연산하고 Remaining_bits를 구하여 전송함으로써 다음 단계의 시작 입력을 찾는데 이용하였다. 이때 이용되는 0의 개수와 데이터는 쉬프터에서 전송된다. 단, VLC3의 경우는 Remaining_bits는 항상 6비트로 한다. 이와 같이 본 논문에서 제안한 coeff_token에서 사용되는 4개의 테이블을 표1, 표2, 표3, 표4에서 보여주고 있다. 그리고 각 테이블에 대한 흐름도는 그림3, 그림4, 그림5, 그림6에서 보여 주고 있다.

4.4 level

level 단계는 coeff_token에서 trailing_ones를 제외한 0이 아닌 계수를 복원하는 부분으로써 테이블0~6까지 총 7개의 테이블이 존재하고, 이를 이용하여 복원해야 하지만 본 논문에서는 간단한 식을 이용하여 테이블 없이 수행하였다. 이 테이블은 처음에는 식(1)을 만족하면 테이블1이고 그렇지 않으면 테이블0에서 참조한다. 이후 테이블의 참조는 처음에 테이블0을 참조했을 경우 테이블1로 참조하는데 만약에 식(2)를 만족하면 하나씩 증가하게 된다.

total_coeff > 10 and trailing_ones < 3 (식1)
((abs(복원된 data) > (3 << (참조 table - 1)))
and (참조 table < 6) 식(2)

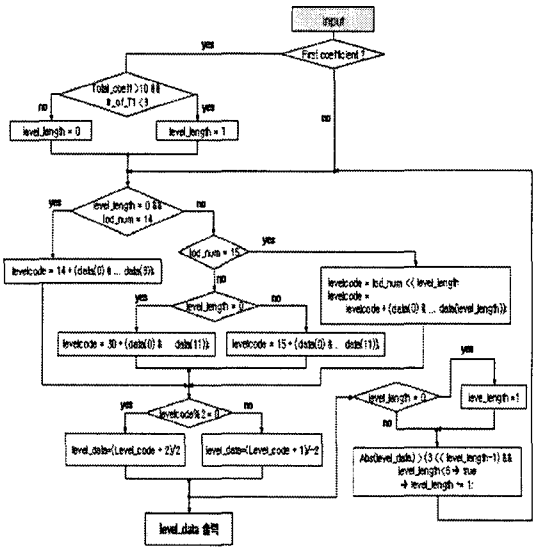

그림 7. level 디코딩의 흐름도

Fig. 7. Flow chart of level decoding

테이블0~6까지를 이용하여 복원하는 level 단계의 흐름도는 그림 7에 나타나 있다. 먼저 level 단계의 첫 수행일 경우와 아닐 경우를 구분하고 첫 수행 단계일 경우 식(1)을 이용하여 테이블을 선택하고, 선택한 테이블을 level_length라고 했을 때 level_length가 0인지 아닌지에 따라 수행하며 0일 경우 level 단계로 입력된 0의 개수가 14일 경우와 15일 경우에 따라 수행하게 된다. 그리고 level_length가 0이고 위 조건이 아닐 경우와 level_length가 0이 아닐 경우에 대한 수행을 하고 입력된 데이터를 이용하여 처리된 데이터가 홀수일 경우와 짝수일 경우를 구분하여 최종 복원 데이터를 출력하고 식(2)에 따라 level_length를 증가시키고 coeff_token - trailing_ones 만큼 반복 수행한다. 이런 알고리즘을 이용하여 level 단계의 테이블을 사용하지 않고 복원하였다.

4.5 total_zeros

total_zeros 단계는 지그재그 스캔에서 최초 0이 아닌 데이터부터 역 순서로 4x4의 블록에서 처음부분까지의 0의개수를 부호화한 비트스트림 데이터를 전송된 데이터를 테이블을 이용하여 복원하는 부분이다. 이 단계에서도 total_coeff의 개수에 따라 코드워드가 구성되어 있지만 불규칙하게 구성되어 있다. 본 논문에서는 전체 코드워드 중에서 total_coeff < 3일 때와 total_coeff > 12일 때를 제외한 나머지 70%만 테이블을 사용하여 수행하였으며, 테이블을 사용하지 않는 경우의 연산은 최초 1을 만나기전까지의 0의 개수와 데이터를 쉬프터에서 입력받아 표 5, 표6에서 보는 바와 같이 간단한 식을 이용하여 수행한다. 그림8은 제안한 total_zeros 단계의 알고리즘에 대한 흐름도를 보여주고 있다.

표 5. coeff_token < 3 경우의 알고리즘
Table 5. Algorithm in case of coeff_token < 3

Lod_num[3:0]	Remaining_bits[1:0]	Total_zeros[3:0]	사용된 전체 비트수 t_bits[4:0]	i[2:0]
0(1)	0	0		선행 1 이후의 비트 값 i[2]i[1]i[0]
1	1	Lod_num + not(i[2])		
2~7	1	Lod_num + not(i[2]) + (lod_num-1)		
8	1	15		

표 6 coeff_token > 12 경우의 알고리즘
Table 6. Algorithm in case of coeff_token > 12

Lod_num[3:0]	Remaining_bits[1:0]	Total_zeros[3:0]	사용된 전체 비트수 t_bits[4:0]	i[2:0]
1(1)	0	0		선행 1 이후의 비트 값 i[2]i[1]i[0]
2	1,2	i[2] = 1 → 4 else i[1] = 1 → 5 i[1] = 0 → 6		
3-5	1	Lod_num + 4 + i[2] + (lod_num/3)		
6	1	13		
7	0	14		

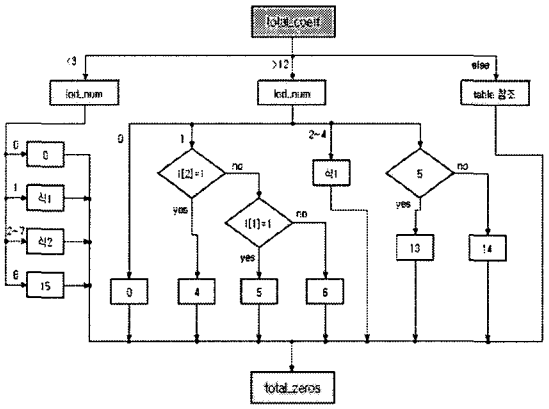

그림 8. 일부만 테이블을 참조한 coeff_token 알고리즘
Fig. 8. Coeff_token algorithm with a partial table reference

4.6 run_before

run_before 단계는 전송된 비트스트림 데이터와 total_zeros 단계에서 처리된 결과와 total_coeffs를 이용하여 trailing_ones와 level 단계에서 복원된 최초 0이 아닌 데이터부터 각각의 복원된 데이터 사이의 0의 개수를 복원하여 최종 CAVLC 데이터를 복원한다.

이 단계에서도 부호화할 때 사용된 42개의 코드워드로 구성된 테이블을 이용하여 데이터를 복원하며, 테이블에 대해 total_zeros에서 처리된 데이터에 따라 총 4단계로 수행된다. 아직 디코딩되지 않은 0의 개수를 "zeroleft"라 하는데, 디코더는 이 값을 참조하여 run_before 값을 알아낸다. 이 값은 total_zeros의 값과 동일하다. 그림 9은 본 논문에서 제안하는 메모리를 사용하지 않는 run_before 디코딩을 위한 흐름도를 보여주고 있다.

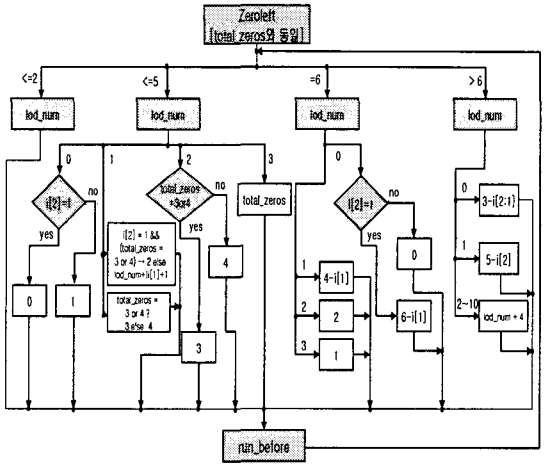

그림 9. 메모리 사용하지 않는 run_before 디코딩 알고리즘
Fig. 9. Memory-free run_before decoding algorithm

4.7 controller와 CAVLC_data 블록

Controller는 각 단계별로 수행을 5단계의 상태로 구분하여 수행하고, 각 단계에서 필요한 0의 개수(lod_num)를 쉬프터를 활성화시킴으로서 구해진 데이터를 전송하게 하며, 쉬프터에서 필요한 입력 데이터를 저장하고 출력하는 역할을 수행한다. 그리고 CAVLC_data에서는 trailing_ones단계에서 수행된 결과와 level단계에서 수행된 결과를 저장하는 역할도 수행한다. run_before 단계의 수행결과에 따라 복원되어 저장된 데이터 또는 0을 출력한다. 이 수행을 run_before 단계가 모두 완료될 때까지 수행한다.

V. 시뮬레이션 및 FPGA 검증

CAVLC 디코더의 입력은 16비트단위로 전송되며 4x4블록단위로 처리된 데이터를 valid가 활성화 되면 입력받아 수행한다. 입력받은 데이터를 이용하여 CAVLC 디코더를 수행하고 첫 데이터를 전송받고 38클록 이후에 첫 복원된 4x4블록의 데이터가 출력된다.

전송이 완료된 후 45클록 이후에 두 번째 복원된 4x4 블록의 데이터를 출력한다. 이렇게 연속적으로 처리된 결과를 그림10에서 보여주고 있다.

본 논문에서 제안한 CAVLC 디코더를 설계하기 위해 사용된 툴은 Xilinx 6.2i ISE를 사용하여 합성 및 P&R 등을 수행하였으며, 시뮬레이터는 Model_Sim 5.7g를 사용하였다.

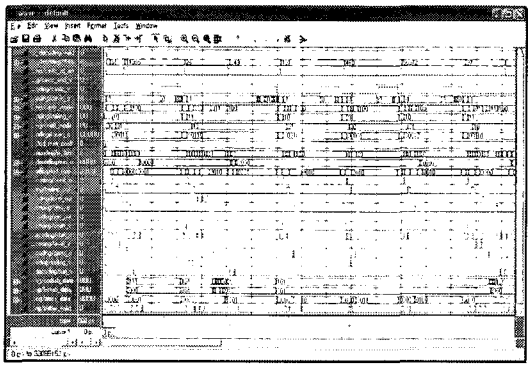

그림 10. 출력 파형
Fig. 10. Output waveform

검증을 완료한 후 CAVLC Top 모듈에서 .bit 파일을 생성하여 FPGA에 다운로드하여 보드 수준의 검증을 수행하였다. 보드 수준의 검증을 위해 설계된 평가 보드와 PC간에 PCI 인터페이스를 통해 동작을 확인하였다. 그림 11은 본 논문에서 사용한 평가 보드와 PC와의 인터페이스를 보여주고 있다.

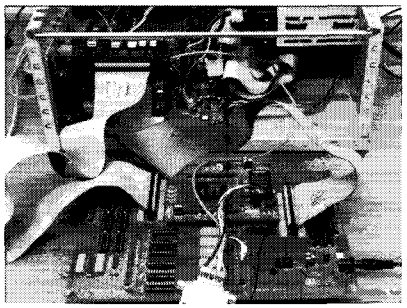

그림 11. PCI 인터페이스를 이용한 타겟보드와 PC의 연결도
Fig. 11. Connection between target board and PC using PCI interface

본 설계의 검증을 위해 Visual 프로그램을 이용하여 원영상, 소프트웨어 처리 영상 및 타겟 보드에서 처리된 영상에 대한 디스플레이할 수 있도록 다이얼로그 박스를 제작하였다.

검증을 위해 사용한 테스트 이미지의 크기는 176x144의 QCIF 영상이며, 테스트 파일은 flower 영상을 사용하

였다. 이 때 사용된 QP 값은 24로 고정하였다. 그림 12는 원영상에 대해 소프트웨어 처리된 영상과 FPGA 타겟 보드를 통해 처리된 영상을 보여주고 있다. 2가지 영상이 동일한 것을 알 수 있다.

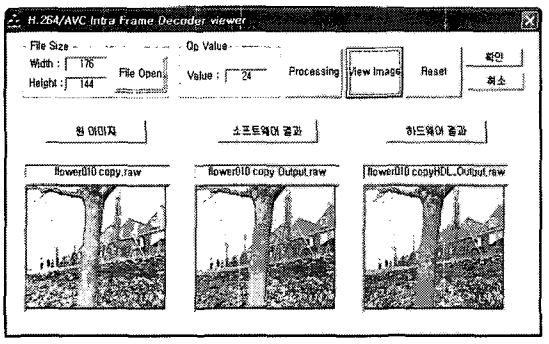

그림 12. QP=10일 때 소프트웨어 및 하드웨어 처리된 영상 비교
Fig. 12. Image comparison processed by software and hardware(QP=10)

VI. 설계 결과 분석

Wu Di 등이 제안한 논문[8]과 본 논문에서 제안한 알고리즘에서 메모리 접근의 감소율을 표7에서 보여주고 있다. 비교를 위해 176x144의 QCIF 4개의 표준 영상을 사용하였다. 표에서 보는 바와 같이 coeff_token에서 사용하는 일부 메모리 중 발생빈도가 높은 것은 테이블을 사용하지 않고 논문[8]에서 제안한 알고리즘을 이용하여 복원함으로써 메모리 접근 67~95%의 감소를 보이고 있지만, 본 논문에서는 메모리를 사용하지 않고 coeff_token의 모든 데이터를 논문에서 제안한 알고리즘으로 수행함으로써 메모리 접근이 100%의 감소를 보인다.

표 8은 run_before 수행에 대해 4개의 표준 영상에 대한 메모리 액세스 회수를 Kim 등이 제안한 논문[9]와 본 논문에서 제안한 알고리즘을 비교한 것이다. 표에 나타난 바와 같이 본 논문에서 제안한 방식은 메모리 액세스를 전혀 하지 않으나, 논문[9]의 방식은 QP값에 따라 변하기는 하지만 많은 메모리 액세스가 있음을 알 수 있다.

또한 제안한 알고리즘의 수행 사이클과 테이블을 참조한 알고리즘의 수행 사이클을 비교하였으며, 그 결과 표준영상 4개에 대해 50~70% 정도의 수행 사이클 감소를 확인하였다. 그림 13에서는 비교 결과를 보여 주고 있다[8,10]. 이러한 비교를 종합하여 판단할 때, 본 논문에서 제안한 알고리즘을 적용하여 CAVLC 디코더를 설계하면 기존의 CAVLC 디코더에 비해 우수한 성능을 얻을 수 있음을 알 수 있었다.

표 7. [9]의 coeff_token의 메모리 액세스 감소율 비교
Table 7. Comparison of Memory access reduction rate for coeff_token

coeff_token 단계 ~ 단위 : %

비교 항목 / Qp	Foreman		News		Container		Silent	
	논문[8]	제안 알고리즘	논문[8]	제안 알고리즘	논문[8]	제안 알고리즘	논문[8]	제안 알고리즘
24	71.2	100	67.6	100	81.4	100	71.6	100
28	77.7	100	70.0	100	79.5	100	78.9	100
32	84.5	100	74.0	100	79.9	100	85.5	100
36	90.7	100	81.2	100	82.9	100	92.2	100
40	95.1	100	88.6	100	88.0	100	94.8	100

※ 위 비교는 메모리 접근 횟수의 감소율 비교로 100은 메모리 접근 횟수가 없음을 의미함.

표 8. 본 논문과 논문[9]의 run_before 수행을 위한 메모리 액세스 횟수의 비교
Table 8. Comparison of the number of memory accesses required for run_before execution between paper[9] and the proposed method

비교 항목 / Qp	Foreman		News		Container		Silent	
	논문[9]	제안 알고리즘	논문[9]	제안 알고리즘	논문[9]	제안 알고리즘	논문[9]	제안 알고리즘
24	387418	0	243038	0	166392	0	185757	0
28	150034	0	131232	0	71258	0	85420	0
32	60244	0	52304	0	32508	0	33784	0
36	19966	0	26270	0	14528	0	10982	0
40	6346	0	10233	0	6070	0	3440	0

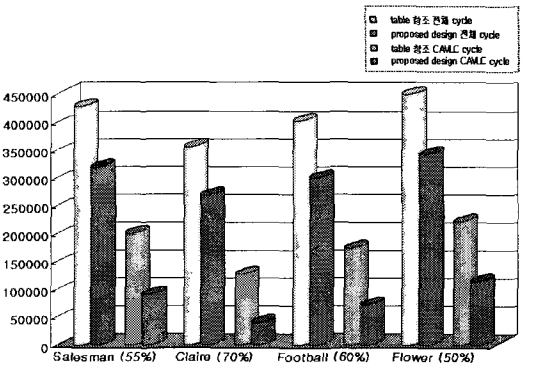

그림 13. 제안한 알고리즘과 테이블 참조 알고리즘의 수행 사이클 비교
Fig. 13. Execution cycle comparison between the proposed algorithm and table reference algorithms

표 9. 제안한 CAVLC 디코더의 합성 결과
Table 9. Synthesized result of the proposed CAVLC decoder

NO. of Slices	No. of Gates	Timing	Target Device
448	17,389	Mininum Period: 6.485ns Maximum Frequency: 154.214MHz	XC2V1000 -6fg256

표 9는 본 논문에서 설계한 CAVLC 디코더의 합결 결과를 보여주고 있다. 타겟 디바이스는 Xilinx의 XC2V1000 FPGA 칩을 사용하였으며, 동작 속도는 약 154MHz이다.

Ⅶ. 결 론

CAVLC 디코더를 수행할 경우 5단계 중에서 coeff_token단계와 level단계, total_zeros단계 그리고 run_before단계에서 테이블을 사용하여 수행한다. 이 모든 단계의 테이블을 사용하여 디코더를 설계할 경우 테이블을 메모리에 모두 저장하여야 하며, 최악의 경우 선택된 테이블에서 입력 데이터와 맞는 데이터를 찾기 위해 모든 전급하게 되는 단점이 발생한다. 이에 칩 면적이 크게 되며, 처리 속도는 감소하고 전력 소모는 증가하게 된다. 이에 본 논문에서 coeff_token단계에서 사용하는 4개의 테이블과 level단계에서 사용하는 7개의 테이블 그리고 run_before에서 사용하는 테이블을 모두 사용하지 않고 total_zeros에서 사용하는 테이블 중에서 70%만을 메모리로 저장하여 처리하였다. 결과적으로 기존의 논문과 성능을 비교할 때, QCIF 표준영상 4개에 대해 50~70% 정도의 수행 사이클 줄임으로서 성능 향상을 보였다. 그 결과 H.264/AVC 모듈에서 핵심 소자로 사용이 가능할 것으로 사료된다.

오늘날 발표된 모든 영상 코덱은 압축을 위해 가변 길이 코딩을 사용하고 있다. 향후 본 논문에서 설계한 CAVLC 디코더를 H.264/AVC의 디코더에 합체하여 검증을 수행할 예정이며, AVS, VC-1 코덱에 대한 가변 길이 코딩의 연구를 진행할 예정이다.

참고문헌

[1] T. Wiegand, *Study of Final Committee Draft of Joint Video Specification Draft 2*, Doc. JVT-F100d2, Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, Dec. 2002.

[2] G. Bjontegaard and K. Lillevold. "Context-adaptive VLC(CAVLC) Coding of Coefficients," JVT of ISO/IEC MPEG & ITU-T VCEG 3rd Meeting, Fairfax, Virginia, May 2002.

[3] Thomas Wiegand, Gray J. Sullivan, Gisle Bjontegaard, and Ajay Luthra, "Overview of the H.264/AVC Video Coding Standard," IEEE Trans. Circuits and systems for video technology, vol.9, pp.287-290, July. 2003.

[4] Iain E.G. Richardson, *H.264 and MPEG-4 Video Compression*, WILEY, 2003

[5] T. Wiegand, *Draft ITU-T Recommendation and Final Draft International Standard of Joint Viedo Specification*, Doc. JVT-G050r1, JVT of ISO/IEC MPEG & ITU-T VCEG 8th Meeting, Geneva, Switzerland, May 2003

[6] Saied, R. Chakrabrati, "Scheduling for minimizing the Number of Memory Access in Low Power Applications," Workshop on VLSI Signal Processing, pp.169-178, Nov. 1996.

[7] ITU-T Rec. H.264 / ISO /IEC 11496 - 10, "Advanced Video Coding," Final Committee Draft, Document JVT-E022, Sep. 2002.

[8] Wu Di, Gao Wen, Hu Mingzeng, and Ji Zhenzhou, " A VLSI Architecture Design of CAVLC Decoder," Proceedings of 5th international conference on ASIC, pp.962-965, Oct. 2003

[9] Yong Ho Moon, Gyu Yeong Kim, and Jae Ho Kim, "An Efficient Decoding of CAVLC in H.264/AVC Video Coding Standard," IEEE Transactions on Consumer Electronics, Vol.51, pp.933-938, Aug. 2005.

[10] Hsiu-Cheng Chang, Chien-Chang Lin, and Jiun-In Guo, "A Novel Low-Cost High Performance VLSI Architecture for MPEG-4 AVC/H.264 CAVLC Decoding," ISCAS 2005, Vol.6, pp.6110-6113, May 2005.

[11] Ihab Amer, Wael Badawy, and Graham Jullien, "Towards MPEG-4 Part 10 System on Chip: A VLSI Prototype for Context-based Adaptive Variable Length Coding(CAVLC)," IEEE workshop on signal processing system, pp.13-15, Oct. 2004.

[12] 이은정, *H.264/AVC 동영상 압축 표준에서의 효율적인 CAVLC 구현*, 부산대학교 대학원 석사 학위논문, Feb. 2005

저자소개

정 덕 영(Duck-Young Jung)

2005년 한신대학교 정보통신학과 (학사)
2007년 한신대학교 대학원 정보통신학과(석사)
2007년~현재 C&S 테크놀로지 반도체 연구소
※ 관심분야 : 영상처리 프로세서 설계, ASIC 설계

손 승 일(Seung-Il Sonh)

1989년 연세대학교 전자공학과 (학사)
1991년 연세대학교 대학원 전자공학과(석사)
1998년 연세대학교 대학원 전자공학과(박사)
2002년~현재 한신대학교 정보통신학과 부교수
※관심분야 : ASIC 설계(네트워크, 영상 칩 설계)