Unified Implementation of Discrete Cosine Transform for Supporting Multiple Video Coding Standards

Joonpyo Pyo[1], In-Cheol Park [2]
Department of Electrical Engineering and Computer Science, KAIST
jppyo@ics.kaist.ac.kr

Abstract – *Transform coding has been widely used in video coding standards to reduce the amount of information to be transferred. MPEG2/H.261 uses 8 × 8 discrete cosine transform (DCT), and H.264 and VC9 employ 4 × 4 integer transform and variable-size integer transforms, respectively. This paper presents a unified hardware architecture to support all the transforms with a single unit. As many blocks are shared among the transforms, the proposed architecture reduces hardware resources significantly.*

Keywords: Discrete cosine transform, integer transform, distributed arithmetic.

1 Introduction

영상 표준 부호기에서 영상 데이터 블록의 상관성을 줄이고 데이터의 분포를 조밀화 하기 위한 Transform 은 뒤에서 수행되는 양자화 과정 및 압축 과정과 더불어 데이터의 양을 줄이는 매우 중요한 역할을 한다.

MPEG-2 는 현재 DVD, 디지털 TV 방송, 위성 방송 등의 분야에서 널리 쓰인다. 이 MPEG-2 의 영상 변환(Transform)은 8 × 8 DCT(Discrete Cosine Transform) [1]이다. 반면 차세대 표준으로 각광받고 있는 MPEG-4 AVC/H.264 는 4 × 4 DCT 의 근사 형태인 4 × 4 Integer Transform [2]을 사용한다. 또 차세대 표준의 자리를 놓고 H.264 와 각축을 벌이고 있는 VC-9 은 Integer Transform 을 사용하지만 앞의 경우와는 다르게 입력 데이터의 크기를 고정하지 않고 네 가지 크기의 입력 데이터를 사용한다. 이러한 다양한 영상 변환이 있으므로, 한 영상 변환만을 수행하는 영상 변환 프로세서보다 여러 영상 변환을 모두 수행하는 통합된 영상 변환 프로세서가 필요하다. 그러므로 본 논문에서는 각 영상 변환을 모두 지원하는 통합된 프로세서의 구조를 제안한다.

본 논문에서 제안하는 영상 변환 프로세서는 알려진 여러 구현 방법 [3]-[7] 중에서 분산연산 곱셈방식(Distributed Arithmetic)을 이용하며, 각 영상 변환을 구성하는 블록들을 공유하여 사용되는 하드웨어를 줄인다.

본 논문은 다음과 같이 구성하였다. 섹션 2 는 여러 영상 표준이 정의하는 몇 가지 영상 변환에 대하여 살펴본다. 섹션 3 은 구현에 사용하는 분산연산 곱셈방식을 다루고, 섹션 4 에서는 전체 프로세서의 구조를 제안하고 구성 블록들의 구조를 설명한다. 섹션 5 에서는 프로세서의 특성과 장점에 대하여 알아보고, 섹션 6 은 결론을 언급한다.

2 Transform in Video Coding Standards

2D 변환은 식 (1)으로 정의한다.

$$\mathbf{Z} = \mathbf{AXA}^{\mathbf{T}} \qquad (1)$$

$\mathbf{X}$ 는 입력 데이터, $\mathbf{A}$ 는 변환 행렬(transform matrix)이다. 식 (1)의 계산 알고리즘은 다음과 같다. 우선 $\mathbf{Y} = \mathbf{AX}^{\mathbf{T}}$ 을 구한다. 이는 $\mathbf{X}^{\mathbf{T}}$ 의 열, 즉 $\mathbf{X}$ 의 행을 1 차원 변환하는 것이다. 이 결과를 전치행렬(transpose matrix)로 바꾸고, 다시 1 차원 변환을 하면 변환식 $\mathbf{Z} = \mathbf{AY}^{\mathbf{T}} = \mathbf{AXA}^{\mathbf{T}}$ 을 얻는다.

2.1 8 × 8 DCT in Previous Standards

8 × 8 DCT 영상 변환에 대한 변환 행렬은 식 (2) 형태이다. 정확한 값은 식 (3)을 참고하면 된다 [8].

$$A = \begin{bmatrix} a & a & a & a & a & a & a & a \\ b & d & e & g & -g & -e & -d & -b \\ c & f & -f & -c & -c & -f & f & c \\ d & -g & -b & -e & e & b & g & -d \\ a & -a & -a & a & a & -a & -a & a \\ e & -b & g & d & -d & -g & b & -e \\ f & -c & c & -f & -f & c & -c & f \\ g & -e & d & -b & b & -d & e & -g \end{bmatrix} \qquad (2)$$

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \end{bmatrix} = \sqrt{\frac{2}{N}} \begin{bmatrix} \cos\frac{\pi}{4} \\ \cos\frac{\pi}{16} \\ \cos\frac{\pi}{8} \\ \cos\frac{3\pi}{16} \\ \cos\frac{5\pi}{16} \\ \cos\frac{3\pi}{8} \\ \cos\frac{7\pi}{16} \end{bmatrix} = \begin{bmatrix} 0.35355 \\ 0.49039 \\ 0.46194 \\ 0.41552 \\ 0.27734 \\ 0.19091 \\ 0.0976 \end{bmatrix} \qquad (3)$$

2.2 4 × 4 Integer Transform in H.264

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \quad (4)$$

식 (4)는 H.264 의 4 × 4 integer transform 변환 행렬이다. 이 영상 변환은 4 × 4 DCT 를 근사하였기 때문에 행렬의 원소들이 모두 간단한 정수이다 [9].

2.3 Integer Transforms in VC-9

VC-9 의 영상 변환은 입력 데이터 크기가 4 × 4, 4 × 8, 8 × 4, 8 × 8 이다. 따라서 총 4 가지 종류의 영상 변환이 있다 [10]. 표 1 에는 입력 데이터 블록 크기에 따른 변환 식을 정리하였다. $\mathbf{A_4}$ 와 $\mathbf{A_8}$ 은 각각 4 × 4 와 8 × 8 DCT 의 변환 행렬을 근사한 행렬이다.

$$\mathbf{A_8} = \begin{bmatrix} 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 16 & 15 & 9 & 4 & -4 & -9 & -15 & -16 \\ 16 & 6 & -6 & -16 & -16 & -6 & 6 & 16 \\ 15 & -4 & -16 & -9 & 9 & 16 & 4 & -15 \\ 12 & -12 & -12 & 12 & 12 & -12 & -12 & 12 \\ 9 & -16 & 4 & 15 & -15 & -4 & 16 & -9 \\ 6 & -16 & 16 & -6 & -6 & 16 & -16 & 6 \\ 4 & -9 & 15 & -16 & 16 & -15 & 9 & -4 \end{bmatrix} \quad (5)$$

$$\mathbf{A_4} = \begin{bmatrix} 17 & 17 & 17 & 17 \\ 22 & 10 & -10 & -22 \\ 17 & -17 & -17 & 17 \\ 10 & -22 & 22 & -10 \end{bmatrix} \quad (6)$$

<표 1> 입력 데이터 종류에 따른 transform equation

입력 데이터 블록 크기	식
4 × 4	$\mathbf{Z = A_4 X A_4^T}$
4 × 8	$\mathbf{Z = A_4 X A_8^T}$
8 × 4	$\mathbf{Z = A_8 X A_4^T}$
8 × 8	$\mathbf{Z = A_8 X A_8^T}$

3 Distributed Arithmetic

분산연산 곱셈방식을 사용하면 곱의 합(Sum of Product)을 곱셈기 없이 구현할 수 있다. 따라서 본 논문에서는 1D 영상 변환의 구현을 위해서 비트-시리얼 방식의 분산연산 곱셈방식을 사용한다. 식 (7)과 같이 m 번의 곱셈이 필요한 곱의 합에 대하여 분산연산 곱셈 방식을 사용하여 결과를 구하는 과정은 다음과 같다.

$$y = \sum_{m=1}^{N} a_m x_m \quad (7)$$

a_m 은 고정점 상수이며 입력 벡터 x_m 는 B 비트 2's 보수체계의 이진수로 아래와 같이 표현한다.

$$x_m = -x_m^{(0)} + \sum_{j=1}^{B-1} x_m^{(j)} 2^{-j} \quad (8)$$

여기에서 $x_m^{(j)}$ 은 x_m 의 j 번째 비트이다. (8)의 식을 (7)에 대입하면

$$\begin{aligned} y &= \sum_{m=1}^{N} a_m \left[-x_m^{(0)} + \sum_{j=1}^{B-1} x_m^{(j)} 2^{-j} \right] \\ &= \sum_{j=1}^{B-1} \left[\sum_{m=1}^{N} a_m x_m^{(j)} \right] 2^{-j} - \sum_{m=1}^{N} a_m x_m^{(0)} \end{aligned} \quad (9)$$

위의 식 중 $F_N(a, x^{(j)}) = \sum_{m=1}^{N} a_m x_m^{(j)}$ 이라 치환하면

$$y = \sum_{j=1}^{B-1} F_N(a, x^{(j)}) 2^{-j} - F_N(a, x^{(0)}) \quad (10)$$

식 (10)으로 정리할 수 있다. a 가 고정된 상수이므로 모든 $x^{(j)}$ 에 대하여 $F_N(a, x^{(j)})$ 을 미리 계산하고 그 결과를 롬에 넣어 둔다. 실제 계산할 때에는 롬 데이터를 읽어서 더하고 이를 오른쪽으로 한 비트 쉬프트 하여서 다시 더하는 것을 계속 반복해 결과값을 얻는다. 단 롬의 입력이 부호 비트(MSB)일 때에는 덧셈 대신에 뺄셈을 수행한다. 롬과 누산기로 이루어진 이 하드웨어를 롬-어큐뮬레이터라 한다. 그림 1 은 구현된 롬-어큐뮬레이터의 블록 다이어그램이다.

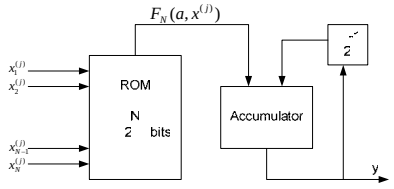


그림 1. ROM-accumulator block diagram.

4 Proposed Architecture

4.1 Overall Configuration

그림 2 는 영상 변환 프로세서의 전체 구조를 보여준다. 2 개의 1 차원 영상 변환 블록이 있다. 그리고 두 영상 변환 블록 사이에서 행과 열을 바꾸어 주는 역할을 하는 전치(transpose)블록이 있다. 또 연산 도중 데이터가 범람하는 것을 막기 위해 라운딩 블록이 비트 수 조절 역할을 한다.

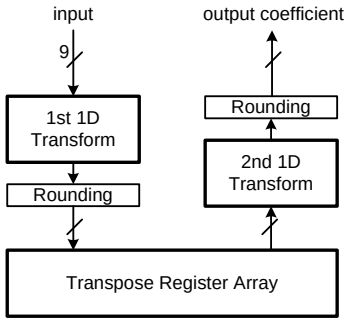


그림 2. Block diagram of overall processor.

4.2 1D Transform Processing Unit

8 × 8 DCT 는 8 포인트 1 차원 영상 변환을 2 번 수행한다. 그리고 4 × 4 integer transform 은 4 포인트 1 차원 영상 변환을 2 번 수행한다. VC-9 의 4 종류의 integer transform 은 종류별로 다르지만 모두 합하여 8 포인트 영상 변환을 4 번, 4 포인트 영상 변환을 4 번 수행한다. 따라서 하드웨어를 통합하여 구현하기 위하여 영상 변환 구조는 4 포인트 1 차원 영상 변환과 8 포인트 1 차원 영상 변환을 모두 수행할 수 있어야 한다.

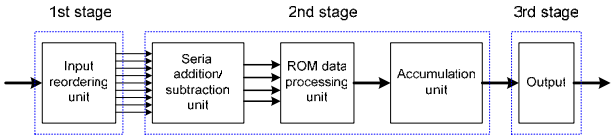
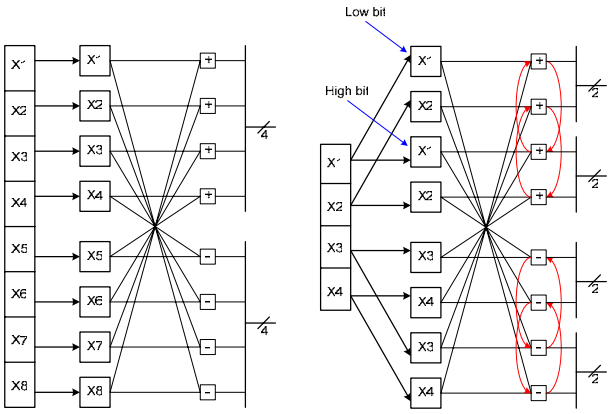


그림 3. Block diagram of 1D transform unit.

1 차원 영상 변환 구조는 크게 3 단계로 나눌 수 있다. 첫 단계에서는 입력 데이터 벡터를 받아 데이터의 순서를 재배치하여 레지스터에 넣는다. 두 번째 단계에서는 레지스터의 데이터를 비트 단위로 꺼내어서 시리얼하게 연산(덧셈/뺄셈)을 수행한다. 이 시리얼 연산 결과를 묶어서 ROM 의 address 로 넣고, 얻어지는 ROM 의 출력을 적절하게 처리한다. 그리고 처리된 결과를 어큐뮬레이터에 누적 연산한다. 마지막 세 번째 단계에서는 어큐뮬레이터의 결과를 출력 벡터로 내보낸다. 그림 3 은 1 차원 영상 변환 구조에 대한 블록 다이어그램이다. 각 단계는 pipeline 형태로 독립적으로 실행한다.

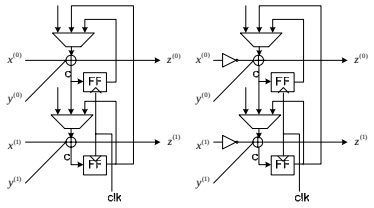
A. 입력 재배치 및 시리얼 연산 블록

그림 4.(a)는 8 포인트 1 차원 영상 변환을 위한 입력 재배치 블록과 시리얼 연산 블록이다. B 비트의 8 × 1 입력 데이터 벡터를 받아서 데이터의 순서를 재배치하여 레지스터 8 개에 넣는다. 그리고 이 레지스터 데이터들을 B 사이클 동안 LSB 에서 MSB 의 순서로 한 비트씩 비트-시리얼 방식으로 처리한다.



(a) (b)

그림 4. (a) 8-point input reordering and serial processing block. (b) 4-point input reordering and serial processing block.



(a) (b)

그림 5. (a) bit-serial adder for both 4 point and 8 point cases. (b) bit-serial subtractor for both 4 point and 8 point cases

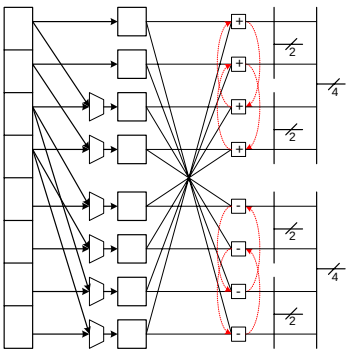


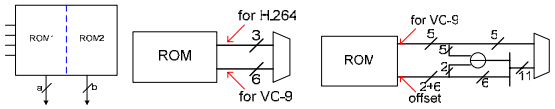
그림 6. Unified input reordering and serial processing block

그림 4.(b)는 4 포인트 1 차원 영상 변환을 위한 구조이다. 8 포인트 변환과의 차이점을 살펴보면, 입력이 4 × 1 벡터이다. 레지스터의 개수가 8 개이므로 입력 받은 데이터 4 개를 2 번 전달할 수 있다. 따라서 데이터를 한 사이클에 2 비트씩 처리하여 속도를 높일 수 있다. 입력이 9 비트이므로 비트 0(LSB)과 비트 1 을

동시에, 2 와 3 을 동시에, 4 와 5 을 동시에, 6 과 7 을 동시에 처리하고 마지막으로 비트 8(MSB)만 처리한다. 2 비트씩 처리하기 때문에 비트-시리얼 덧셈기/뺄셈기 2 개를 사용하여 각각의 carry out 을 carry in 으로 넣어준다. 그림 4.(b)에서 각 덧셈기나 뺄셈기들을 연결하는 선은 carry 의 이동을 의미한다. 그림 5 와 같이 구현하면 8 포인트를 위한 bit serial 계산과 4 포인트를 위한 2-bit serial 계산을 모두 수행할 수 있다.

통합된 입력 재배열과 시리얼 프로세스 구조는 그림 6 에 있다.

B. 롬-어큐뮬레이터 블록



(a) (b) (c)

그림 7. (a) Unification of ROMs which has same input. (b) ROM for 4 point transform. (c) ROM for 8 point transform.

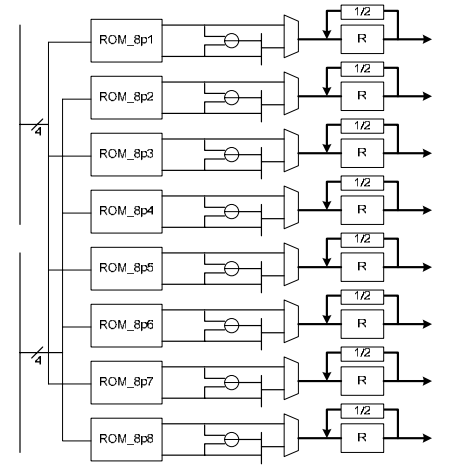
<표 2> DCT 과 VC-9 의 coefficient 비교

DCT coefficient	Value	12 bit binary	MSB 6bit	VC-9 coefficient	Offset (Difference)
a	0.35355	0.01011010100	001011→11	12	1
b	0.49039	0.01111101100	001111→15	16	1
c	0.46194	0.01110110010	001110→14	16	2
d	0.41573	0.01101010011	001101→13	15	2
e	0.27779	0.01000111000	001000→8	9	1
f	0.19134	0.00110000111	000110→6	6	0
g	0.09755	0.00011000111	000011→3	4	1

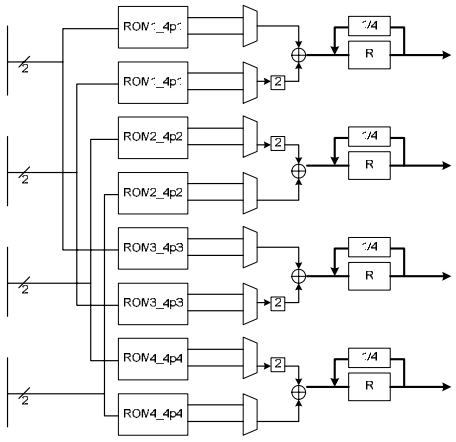
입력이 같고 데이터가 다른 두 롬은 그림 7.(a)와 같이 하나의 롬으로 합할 수 있다. 그리하여 하나의 롬에서 두 출력을 얻는다.

4 포인트 변환의 경우, 롬 부분은 그림 7.(b)의 구조이다. H.264 와 VC-9 의 두 데이터를 갖는 롬 2 개를 그림 7.(a)처럼 하나로 합하고 H.264 값과 VC-9 값 중 하나를 부호화기를 이용하여 선택한다.

8 포인트 변환의 경우, 롬 크기를 줄인 롬-어큐뮬레이터 구조는 그림 7.(c)이다. Coefficient 의 특성을 이용하여 롬 데이터의 비트 수를 감소시켜 크기를 줄인다. 표 2 는 DCT coefficient 의 값과 8 × 8 integer transform coefficient 의 값을 비교한다. 표 2 를 통하여 DCT coefficient 의 MSB 6bit 값과 8 × 8 integer transform coefficient 값에서 유사성을 발견할 수 있다. 두 값의 차이를 오프셋으로 한다.



(a)



(b)

그림 8. (a) ROM-accumulator of 8-point 1D transform. (b) ROM-accumulator of 4-point 1D transform.

DCT coefficient 비트 수에 비하여 VC-9 의 coefficient 비트 수가 더 작으므로 롬 1 에는 VC-9 을 위한 계산 결과를 저장한다. 그리고 롬 2 에는, 오프셋 2 비트와 DCT coefficient 의 LSB 6 비트를 concatenation 하여 롬의 값으로 넣어둔다. 실제로 VC-9 의 연산 시에는 롬 1 을 그대로 사용하고, DCT 의 연산 시에는 롬 1 의 데이터를 롬 2 의 상위 2 비트(오프셋)로 뺀 값에 롬 2 의 하위 6 비트를 concatenation 하여 사용한다. DCT 롬 출력 비트 수에 비하여 롬 2 출력 비트 수가 작기 때문에 뺄셈기를 사용하면서 전체 롬 크기를 줄일 수 있다. 역시 그림 7.(a) 형식을 이용한다.

그림 8.(a)는 8 포인트 영상 변환에 대한 롬-어큐뮬레이터 구조이며, 그림 8.(b)는 4 포인트 영상 변환에 대한 롬-어큐뮬레이터 구조이다.

4.3 Transpose Register Array

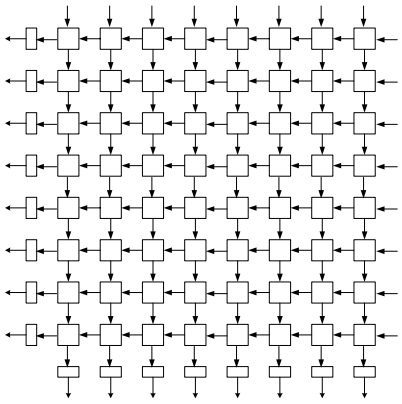


그림 9. Transpose register array

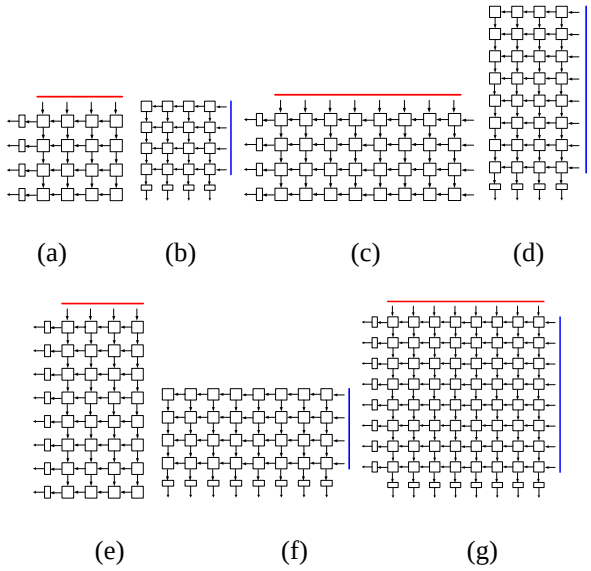


그림 10. Usage of transpose register array in different data block size.

전치블록은 첫 번째 영상 변환 블록과 두 번째 영상 변환 블록 사이에서 처음 블록의 결과를 두 번째 블록의 입력으로 만들기 위하여 행과 열을 뒤바꿀 때 사용한다 [11]. 그림 9 의 레지스터 배열 블록을 사용하면 4 × 4, 4 × 8, 8 × 4, 8 × 8 크기의 모든 데이터를 transpose 할 수 있다.

그림 10 은 각 데이터 종류에 따른 전치블록의 데이터 경로를 보여준다. 4 × 4 영상 변환 데이터의 경우 4 포인트의 데이터들이 입력되고 4 번 반복 입력 후에 전치블록 밖으로 출력된다. 데이터는 위 혹은 오른쪽 방향에서 입력되며 이 방향은 전치블록 내에 있는 이전 데이터의 진행 방향에 따라 결정된다. 그림 10.(a) 또는 10.(b)에 해당한다. 4 × 8 데이터의 경우 8 포인트의 데이터들이 입력되고 4 번 반복 입력 후에

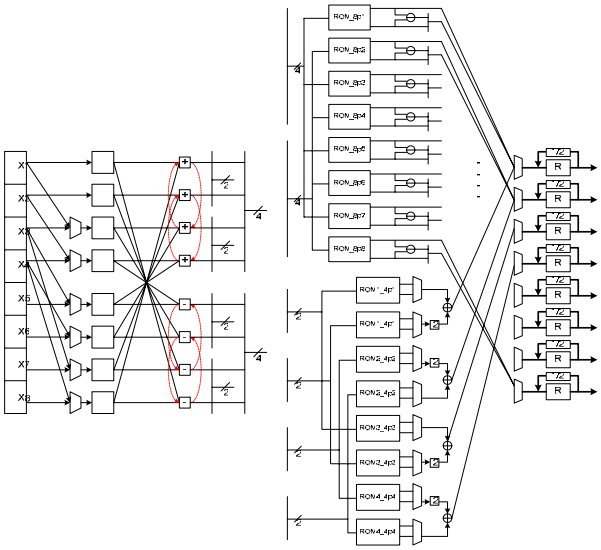


그림 11. Unified overall architecture.

데이터가 전치블록 바깥으로 나간다. 이 때 전치블록에 들어있는 이전 데이터가 어느 방향으로 출력되고 있는 가에 따라서 입력 방향이 결정되어 그림 10.(c) 또는 10.(d)에 해당한다. 8 × 4 데이터의 경우 4 포인트의 데이터들이 입력되고 이를 8 번 반복한 후에 데이터가 전치블록 바깥으로 나간다. 역시 전치블록에 들어있는 데이터의 이동 방향에 따라서 입력 방향이 결정되어 그림 10.(e) 또는 10.(f)에 해당한다. 그림 10.(g)는 8 × 8 데이터에 해당 한다. 마찬가지로 입력은 2 방향으로 들어온다.

DCT 는 모두 8 포인트 변환이고 H.264 의 integer transform 역시 모두 4 포인트 변환이므로 당연히 한 전치블록으로 transpose 를 수행한다. 뿐만 아니라 4 포인트 변환과 8 포인트 변환이 임의로 VC-9 에서 수행되어도 항상 한 전치블록으로 transpose 가능하다.

5 Result

<표 3> Comparison of required hardware

	2^2 word ROMs	2^4 word ROMs	bit-serial adder	bit-serial subtractor	accumulator	subtractor	adder	mux
4×4 H.264	4	0	2	2	4	0	0	0
8×8 DCT	0	8	4	4	8	0	0	0
Transform VC-9	8	8	4	4	8	0	0	10
Total Sum	12	16	10	10	20	0	0	10
Proposed Architecture	8	8	4	4	8	8	4	22

표 3 은 통합하여 구현한 하드웨어와 각 영상 변환을 개별 구현하는 하드웨어를 비교한다. 사용한 비트-시리얼 덧셈기/뺄셈기의 개수는 60%가 감소하였고, 어큐뮬레이터 개수 역시 60% 만큼 감소하였다. 사용한 롬의 개수는 입력 어드레스의 종류에 따라 각각 33%, 50% 감소하였다. 반면 뺄셈기와 덧셈기, 부호화기의 수는 증가하였다

6 Conclusions

이 논문에서는 3 가지 영상 변환을 하나의 하드웨어로 통합하여 수행하는 구조를 제안하였다. 이 프로세서는 분산연산 곱셈방식을 기반으로 하며, 병렬 구조에 비하여 하드웨어를 대부분 더 적게 사용하였다.

References

[1] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform, " *IEEE Trans. Computers*, vol. C-23, pp. 90-94, 1974.

[2] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerosfsky, "Low-complexity transform and quantization in H.264/AVC," *IEEE Trans. Circuits Syst. Video Technol*., vol. 13, no. 7, pp. 598-603, Jul. 2003.

[3] N. I. Cho and S. U. Lee, "Fast algorithm and implementation of 2-D discrete cosine transform," *IEEE Trans. Circuits Syst. II*, vol. CAS-38, pp.297-305, Mar. 1991.

[4] S. Uramoto *et al*., "A 100MHz 2-D discrete cosine transform core processor," *IEEE J. Solid-State Circuits*, vol. 27, pp. 492-499, Apr. 1992

[5] Y. T. Chang and C. L. Wang, "New systolic array implementation of the 2-D discrete cosine transform and its inverse," *IEEE Trans. Circuits Syst. Video Technol*., vol. 5, pp. 150-157, Apr. 1995

[6] Y. P. Lee, T. H. Chen, L. G. Chen, M. J. Chen, and C. W. Ku, "A cost-effective architecture for 8 × 8 two-dimensional DCT/IDCT using direct method," *IEEE Trans. Circuits Syst. Video Technol*., vol. 7, pp. 459-467, June 1997.

[7] M. Sun *et al*., "VLSI implementation of a 16 × 16 discrete cosine transform," *IEEE Trans. Circuits and Syst*., vol. 36, pp. 610-617, Apr. 1989.

[8] A. Madisetti and A. N. Willson, "A 100MHz 2-D 8 × 8 DCT/IDCT processor for HDTV applications," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, pp. 158-165, Apr. 1995.

[9] I. E. G. Richardson, "H.264 and MPEG-4 Video Compression," New York, NY: Wiley, 2003.

[10] Committee Draft : Video Codec VC-9

[11] Tu-chih Wang, *et al*., "Parallel 4×4 2D transform and inverse transform architecture for MPEG-4 AVC/H.264," in *Proc. of 2003 IEEE International Symposium on Circuits and Systems*, Bangkok, Thailand, May 2003, pp. 800-803.