Homogeneous Transformation Matrix의 곱셈을 위한 병렬구조 프로세서의 설계

論 文
54D-12-6

A Parallel-Architecture Processor Design for the Fast Multiplication of Homogeneous Transformation Matrices

權 斗 兀† · 鄭 台 相**
(Do-All Kwon · Tae-Sang Chung)

Abstract - The 4×4 homogeneous transformation matrix is a compact representation of orientation and position of an object in robotics and computer graphics. A coordinate transformation is accomplished through the successive multiplications of homogeneous matrices, each of which represents the orientation and position of each corresponding link. Thus, for real time control applications in robotics or animation in computer graphics, the fast multiplication of homogeneous matrices is quite demanding. In this paper, a parallel-architecture vector processor is designed for this purpose. The processor has several key features. For the accuracy of computation for real application, the operands of the processors are floating point numbers based on the IEEE Standard 754. For the parallelism and reduction of hardware redundancy, the processor takes column vectors of homogeneous matrices as multiplication unit. To further improve the throughput, the processor structure and its control is based on a pipe-lined structure. Since the designed processor can be used as a special purpose coprocessor in robotics and computer graphics, additionally to special matrix/matrix or matrix/vector multiplication, several other useful instructions for various transformation algorithms are included for wide application of the new design. The suggested instruction set will serve as standard in future processor design for Robotics and Computer Graphics. The design is verified using FPGA implementation. Also a comparative performance improvement of the proposed design is studied compared to a uni-processor approach for possibilities of its real time application.

Key Words : Homogeneous Transformation Matrix, Matrix Multiplication, Parallel Algorithm, Vector Architecture, Floating Point Number

1. 서 론

1970년대 현대적 의미의 마이크로프로세서가 출현한 이래 프로세서는 무어의 법칙(Moore's Law)이 밝힌 바와 같이 반도체 집적기술의 발달로 인해 그 성능에 있어 눈부신 발전을 거듭해왔다. 이는 보통 동작속도의 증가로 이루어지거나 많은 양의 하드웨어 집적으로 가능한 새로운 구조의 개발로 이루어진다.

최근에는 멀티미디어 시장의 폭발적인 성장으로 인해 과거 범용 프로세서나 DSP(Digital Signal Processor)를 이용, 소프트웨어로 처리하던 각종 멀티미디어 알고리듬을 전용으로 처리하는 미디어 프로세서가 등장하고 있다. 초창기에는 범용 프로세서에 별도의 멀티미디어 처리유닛을 첨가하는 방식으로 해결하였으나 최근에는 시스템의 고사양화와 더불어 알고리듬의 병렬성과 실시간적인 응답특성으로 인해 프로세서의 설계패턴이 변하고 있는 추세이며, 이를 위해 SIMD(Single Instruction Multiple Data) 구조 등 다양한 병렬구조의 프로세서가 개발되고 있다.[1][2]

본 논문에서는 그 중 로보틱스나 컴퓨터 그래픽스에서 물체의 위치와 방향을 표현하는 방법인 Homogeneous Transformation Matrix 연산 알고리듬을 위한 프로세서를 설계하였다. 여기서 주로 필요로 하는 연산은 4×4 행렬 간 곱셈 혹은 4×4 행렬과 4×1 벡터 간 곱셈으로 이를 통해 물체의 좌표변환을 수행하며, 실제 응용에 있어서는 실시간적인 응답특성을 보여야하므로 매우 빠른 계산이 요구되는 바이다.

실제로 행렬곱셈은 다른 응용분야에서도 매우 자주 수행되는 연산으로 빠른 처리를 위한 많은 연구가 진행되어왔다. 행렬곱셈은 행렬의 각 원소 간 곱셈과 그 결과의 덧셈을 기본연산으로 하고 있으므로 MAC(Multiply- Accumulate) 유닛을 이용해 곱셈과 누적덧셈을 동시에 처리하는 방식이 대표적이다. 또한 행렬의 행 혹은 열벡터를 기본 연산단위로 하는 벡터구조를 이용해 병렬적으로 처리하는 방식도 제안되었다.[3] 본 논문에서는 기존의 벡터구조를 확장하여 행렬곱셈의 병렬적 특성을 최대한 이용하는 구조를 적용하였다. 따라서 하드웨어가 많이 소요되지만 매우 빠른 연산결과를 얻을 수 있었다. 수의 표현은 이론적인 정수표현 대신, 현장에 적용할 수 있도록 정밀도를 향상시키기 위해 IEEE

† 교신저자, 學生會員 : 中央大 工大 電子電氣工學部 碩士課程
E-mail : twoducks@hanmail.net
* 正 會 員 : 中央大 工大 電子電氣工學部 敎授 · 工博
接受日字 : 2005年 6月 16日
最終完了 : 2005年 11月 9日

754 표준[4]을 기반으로 하는 부동 소수점 수를 채택하였다. 또한 파이프라인 구조 설계를 통해 행렬곱셈의 경우 내부적으로 각 행렬/벡터 연산 간 중첩처리가 가능하므로 처리율을 크게 향상시킬 수 있었다.

설계는 하드웨어 기술언어인 VHDL을 이용하였다. HDL simulator로 동작을 검증한 후, Altera사의 FPGA 구현 툴(tool)인 MAX+plus II를 이용하여 성능을 검증하였다.

본 논문의 구성은 다음과 같다. 2장은 배경이론으로서 설계의 기본이 되는 Homogeneous Transformation Matrix 연산 알고리듬을 정리하고, 설계에 적용된 병렬연산 알고리듬을 설명한다. 3장에서는 프로세서의 설계 전반에 대해서 설명한다. 우선 적용된 프로세서의 구조와 명령어들을 정의하고, 각 유닛별로 세부적인 설계사양을 살펴본다. 4장에서는 설계한 프로세서의 동작을 검증하고, 실제 알고리듬의 적용예를 통해 전체적인 성능을 평가하며, 5장을 통해 결론을 내린다.

2. 배경이론

2.1 Homogeneous Transformation Matrix

일반적으로 3차원 공간에서 물체의 위치는 좌표축상의 원점으로부터 임의의 점까지 벡터로써 표현된다. 원점을 시점으로 하고 그 점을 종점으로 할 때, 각 축의 크기성분으로 위치를 나타내는 것이다. (그림 2.1)

실제 응용에 있어서는 연산의 편리성을 위해 이를 행렬의 형태로 표현하며, 또한 scale factor를 포함하여 4×1 열벡터로 표현하는 것이 일반적이다.

$$P=\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix},\ a_x=\frac{x}{w},\ b_y=\frac{y}{w},\ c_z=\frac{z}{w} \qquad \text{(식 2.1)}$$

만약 scale factor ω가 1보다 크다면 벡터성분은 확대된 것이며, 1보다 작다면 축소된 것이다. 1이라면 변하지 않은 상태이며, 특별히 0인 경우는 단순히 방향벡터로서의 의미만 지닌다.

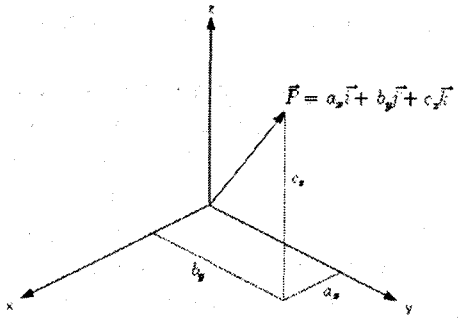

그림 2.1 3 차원 공간에서 벡터의 표현
Fig 2.1 Representation of a vector in 3D space

그러나 실제로 공간상의 물체는 하나의 점이 아니므로 프레임의 개념으로 이를 표현하게 된다. 즉, 3개의 상호 수직적인 단위벡터 $\vec{n},\vec{o},\vec{a}$ 로 대상 프레임의 방향을 나타내며, 각각은 normal, orientation, approach 벡터를 의미한다. 앞서 설명한 바와 같이 이 벡터들은 방향벡터이므로 scale factor는 0이 된다. 아울러 대상 프레임의 원점을 기준 프레임에서 벡터의 형식으로 표현할 수 있으므로 이를 통해 물체의 위치와 방향을 완전히 표현할 수 있게 된다. 이 정보를 행렬의 형태로 나타낸 것이 다음 식이다.

$$F=\begin{bmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(식 2.2)}$$

여기서 1열부터 3열까지는 기준 프레임의 각축에 대한 대상 프레임의 방향을 나타내는 방향벡터이며, 4열은 기준 프레임에 대한 대상 프레임의 원점의 위치를 나타내는 위치벡터이다. 각 열벡터에서 scale factor가 그 의미를 나타내고 있음을 확인할 수 있으며, 그림 2.2는 이를 개념적으로 보여주고 있다.

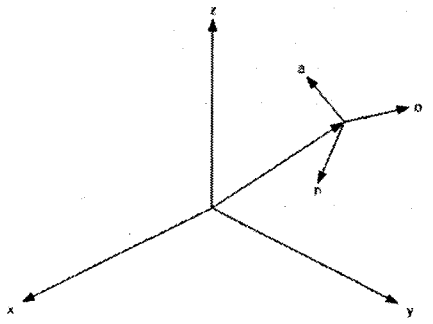

그림 2.2 3차원 공간에서 프레임의 표현
Fig 2.2 Representation of a frame in 3D space

지금까지 공간에서 물체의 위치 및 방향이 4×4 행렬로 완전히 표현됨을 살펴보았다. 여러 가지 이유로 물체의 위치를 표현하는 행렬을 정사각행렬의 형태로 유지하는 것이 요구된다. 다음 절에서 살펴보겠지만 물체의 좌표변환은 두 행렬의 곱셈으로 이루어지는데, 행렬곱셈의 특성상 첫 번째 행렬의 열의 개수와 두 번째 행렬의 행의 개수가 일치하여야 한다. 정사각행렬의 경우 곱셈 후에도 행렬의 차원이 변하지 않으므로 표현에 있어 일관성을 기할 수 있고, 연속적으로 이루어지는 변환에 있어 효율적인 연산을 기대할 수 있다. 예를 들어 식 2.2의 행렬에서 4행은 고정적인 요소로서 실제 곱셈에 있어 아무런 영향도 미치지 못하지만 이러한 이유로 삽입된 것이다. 이런 형태의 행렬을 바로 Homogeneous Transformation Matrix라고 부른다.[5]

2.2 좌표변환 알고리듬

3차원 공간에서 이루어지는 좌표의 변환은 기준 프레임에

대한 대상 프레임의 이동으로 정의된다. 이러한 변환은 공간상의 프레임의 표현과 동일한데, 프레임 자체가 물체의 변환된 좌표를 나타내고 있기 때문이다. 실제 변환은 이동, 회전, 그리고 이것의 연속적인 조합의 형태로 이루어지며, Homogeneous Matrix의 곱셈을 통해 수행된다.

2.2.1 이동변환 (Translation Transformation)

$$Trans(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(식 2.3)}$$

이동변환은 프레임의 방향은 변화시키지 않고 위치만 이동시킨다. 다시 말해 프레임의 원점의 위치가 변하는 것으로 Homogeneous 표현에서 단지 위치벡터의 성분만 변하게 된다. 이동변환을 수식으로 표현하면 다음과 같다.

$$F_{new} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} n_x & o_x & a_x & P_x + d_x \\ n_y & o_y & a_y & P_y + d_y \\ n_z & o_z & a_z & P_z + d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(식 2.4)}$$

변환 전의 프레임과 변환 후의 프레임을 비교해보면 단지 원점의 위치만 변화함을 확인할 수 있다.

2.2.2 회전변환 (Rotation Transformation)

$$Rot(x, \theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(식 2.5)}$$

$$Rot(y, \theta) = \begin{bmatrix} \cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(식 2.6)}$$

$$Rot(z, \theta) = \begin{bmatrix} \cos\theta & -sin\theta & 0 & 0 \\ sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(식 2.7)}$$

회전변환은 프레임을 x, y, z축에 대하여 각 Θ만큼 회전시킨다. 따라서 Homogeneous 표현에서 전 벡터들의 성분이 영향을 받게 되는데, x축에 대한 회전을 수식으로 나타내면 다음과 같다.

$$F_{new} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} n_x & o_x & a_x & P_x \\ cos\theta n_y - sin\theta n_z & cos\theta o_y - sin\theta o_z & cos\theta a_y - sin\theta a_z & cos\theta P_y - sin\theta P_z \\ sin\theta n_y + cos\theta n_z & sin\theta o_y + cos\theta o_z & sin\theta a_y + cos\theta a_z & sin\theta P_y + cos\theta P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(식 2.8)}$$

2.2.3 이동변환과 회전변환의 조합

실제 변환은 이동변환과 회전변환의 연속적인 조합으로 이루어진다. 즉, 각 변환행렬의 순차적인 곱셈을 통해 프레임 혹은 벡터의 변환된 최종의 위치를 얻게 된다. 이 때 곱해지는 순서에 따라 다른 변환결과를 보이게 되는데, 변환행렬을 대상행렬의 앞에 곱하는 경우(pre-multiply)는 기준 프레임에 대하여 절대적인 변환을 수행하며, 뒤에 곱하는 경우(post-multiply)는 현재의 프레임에 대하여 상대적인 변환을 수행하게 된다. 따라서 적절한 변환순서를 정해야 하는데, 예를 들어, 일반적인 직각좌표계에서 먼저 기준 프레임에 대한 이동변환 후, 변환된 프레임의 각 축에 대한 회전변환으로 이루어지는 최종의 변환은 다음과 같은 알고리듬을 갖는다.

$$F_{new} = T \times F_{old} = (T_1 \cdot T_2 \cdot T_3 \cdot T_4) \times F_{old} \quad \text{(식 2.9)}$$

$$T_1 = Trans(d_x, d_y, d_z)$$
$$T_2 = Rot(a, \theta_a)$$
$$T_3 = Rot(o, \theta_o)$$
$$T_4 = Rot(n, \theta_n)$$

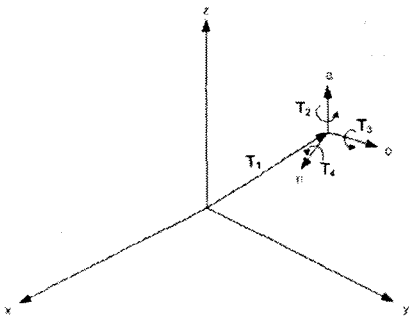

그림 2.3 직각좌표계에서의 좌표변환
Fig 2.3 Transformation in cartesian coordinate

2.3 행렬곱셈의 병렬성

임의의 두 N차 행렬의 곱셈을 하나의 곱셈기와 덧셈기를 가지고 순차적으로 처리할 경우, 곱셈은 N^3회, 덧셈은 $N^2(N-1)$회 만큼의 계산시간을 가진다. MAC 유닛을 이용할 경우, 각 원소에 대한 누적덧셈을 곱셈과 동시에 처리하므로 덧셈에 대한 시간을 줄일 수 있으나 여전히 많은 계산시간을 필요로 한다. 따라서 이를 병렬적으로 처리하는 알고리듬이 요구된다.

기본적으로 같은 차원을 가지는 행렬의 곱셈에서 결과 행렬의 각 열벡터는 첫 번째 행렬과 두 번째 행렬의 각 열벡터의 행렬/벡터 간 곱셈의 결과이다. 즉, 두 번째 행렬의 각 열벡터는 결과 행렬의 대응되는 열벡터에만 영향을 미친다. 따라서 행렬의 곱셈을 첫 번째 행렬과 두 번째 행렬의 각 열벡터 간 곱셈으로 나누어 처리할 수 있으며, 여기서 연산

의 첫 번째 병렬적 특성을 찾을 수 있다.

다음으로 행렬과 열벡터의 곱셈에서 열벡터의 각 원소는 행렬의 대응되는 열에만 곱해지게 된다. 이것이 두 번째 병렬적 특성으로 첫 번째 행렬의 각 열을 독립적인 기본 연산 단위로 하는 병렬연산이 가능하게 된다. 이를 Homogeneous Transformation Matrix 곱셈에 적용한 결과 수행되는 병렬 연산 알고리듬은 다음과 같다.

표 2.1 Homogeneous Transformation Matrix 곱셈의 병렬연산 알고리듬

Table 2.1 Parallel algorithm of the multiplication of homogeneous transformation matrices

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & p_1 \\ a_{21} & a_{22} & a_{23} & p_2 \\ a_{31} & a_{32} & a_{33} & p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} & q_1 \\ b_{21} & b_{22} & b_{23} & q_2 \\ b_{31} & b_{32} & b_{33} & q_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} z_{11} & z_{12} & z_{13} & z_{14} \\ z_{21} & z_{22} & z_{23} & z_{24} \\ z_{31} & z_{32} & z_{33} & z_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

제1회	$\begin{pmatrix} z_{11} \\ z_{21} \\ z_{31} \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} \times b_{11} + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} \times b_{21} + \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} \times b_{31}$
제2회	$\begin{pmatrix} z_{12} \\ z_{22} \\ z_{32} \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} \times b_{12} + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} \times b_{22} + \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} \times b_{32}$
제3회	$\begin{pmatrix} z_{13} \\ z_{23} \\ z_{33} \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} \times b_{13} + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} \times b_{23} + \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} \times b_{33}$
제4회	$\begin{pmatrix} z_{14} \\ z_{24} \\ z_{34} \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} \times q_1 + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} \times q_2 + \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} \times q_3 + \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}$

따라서 이 알고리듬을 4×4 Homogeneous Matrix의 곱셈에 적용할 경우, 표 2.1에서 나타난 것과 같이 총 4회의 계산만으로 행렬의 곱셈을 수행할 수 있으며, 순차적인 처리에 비해 매우 빠른 계산이 가능하다. 첫 번째 행렬의 각 열벡터는 행렬곱셈이 수행되는 동안 고정적으로 곱해지며, 두 번째 행렬의 각 열벡터가 매회 순차적으로 입력되는 방식으로 전체 연산을 수행하게 된다. 마찬가지로 결과 또한 매회 열벡터의 순서대로 출력되므로 각 행렬의 열벡터를 기본 연산 단위로 하는 벡터구조가 적용될 수 있다. 이를 위해 하드웨어적인 관점에서 벡터의 각 원소 간 연산을 수행하는 벡터 곱셈기와 벡터 덧셈기가 기본을 이루며, 행렬단위의 처리를 위해 벡터 레지스터 파일의 수정이 요구된다. 또한 마지막 열벡터의 계산 시 추가적인 덧셈이 필요하므로 이에 대한 약간의 제어도 필요하다.

3. 설 계

3.1 프로세서의 구조

설계한 프로세서는 2장에서 설명한 알고리듬을 위해 행렬의 열벡터를 연산단위로 하는 벡터구조를 기본으로 하고 있으며, 벡터 처리유닛의 배열을 통해 병렬성을 최대화하였다. 행렬과 열벡터의 곱셈은 벡터와 스칼라의 곱셈 후 벡터 간 덧셈의 2단계로 이루어지므로 처리율의 향상을 위해 각 처리유닛은 파이프라인구조를 갖는다. 프로세서의 전체 구조는 그림 3.1과 같다.

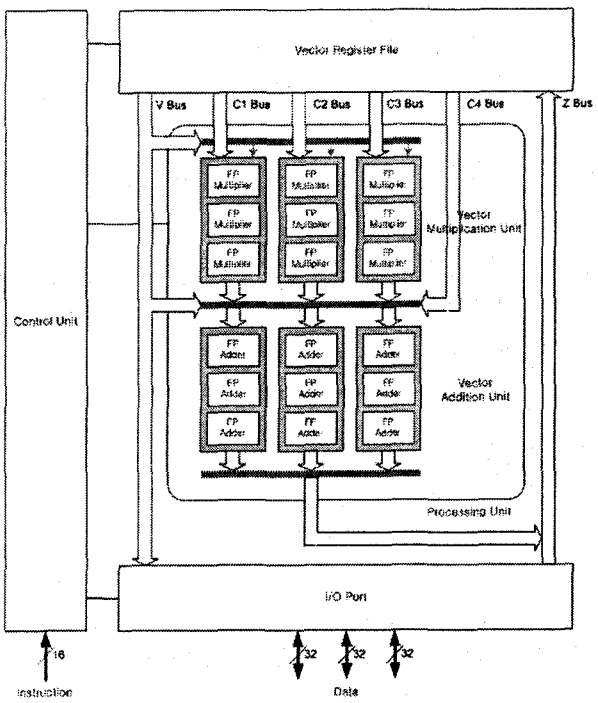

그림 3.1 프로세서의 블록 다이어그램

Fig 3.1 Block diagram of processor

프로세서의 구조는 행렬과 벡터의 연산을 수행하는 Processing Unit(PU), 데이터를 임시적으로 저장하는 Vector Register File(VRF), 외부와의 데이터 송수신을 위한 I/O port와 데이터의 내부전송을 위한 여러 개의 버스, 그리고 전체 프로세서를 제어하는 Control Unit(CU)으로 구성되어 있다.

내부버스는 행렬단위의 처리를 위해서 다소 복잡한 구조를 가진다. V 버스는 행렬과 벡터의 곱셈 혹은 행렬과 행렬의 곱셈 시 벡터 데이터를 전송한다. C1부터 C4까지의 버스는 행렬 데이터를 열벡터의 형태로 동시에 전송하며, Z 버스는 결과 벡터를 담당한다. 독립적인 벡터덧셈 시 V 버스와 C4 버스를 공유하도록 설계하였다. 모든 연산이 32-bit 부동 소수점 수로 표현되는 3×1 벡터 단위로 이루어지므로 내부버스는 96-bit의 폭을 갖는다.

I/O port는 실시간적인 응답을 위해 외부와 데이터를 주고받는 역할을 한다. 3개의 32-bit 포트를 통해 벡터단위로 입출력을 수행하며, 입력을 위해서 Z 버스, 출력을 위해서 V 버스를 내부적으로 공유하고 있다.

Control Unit는 외부로부터 16-bit의 명령어를 받아 프로세서의 전체적인 제어를 수행한다. 설계한 프로세서는 상당히 복잡한 제어가 요구되며, 향후 기능의 확장을 위해 마이크로프로그래밍(micro-programming) 방식의 제어구조를 채택하였다.[6]

3.2 프로세서의 명령어

프로세서의 명령어는 외부로부터 데이터를 입력받는 LD.V와 LD.M, 외부로 데이터를 출력하는 OUT.V와 OUT.M, 행렬과 벡터의 곱셈을 수행하는 FPM.V, 행렬과 행렬의 곱셈을 수행하는 FPM.M, 두 벡터의 덧셈을 수행하는 FPA.V의 7개로 구성되어있다. 이 중 벡터덧셈 명령어는 이동변환은 실제로 곱셈이 아닌 덧셈으로 수행할 수 있으므로 연산의 효율성을 위해 추가하였다.

명령어의 형식은 16-bit의 고정길이를 갖는다. 상위부터 4-bit씩, 4개의 필드로 구성되는데, 순서대로 OP code, 목적지 레지스터, 소스 레지스터 1, 소스 레지스터 2를 의미한다. 벡터 레지스터는 96-bit의 크기를 갖는 V0부터 V7까지 8개이며, 행렬로 접근 시 4개의 벡터 레지스터가 하나의 행렬을 구성하므로 M0부터 M1까지 2개의 행렬을 처리할 수 있다. 명령어의 형식과 코드는 표 3.1에 정리하였다.

표 3.1 프로세서의 명령어 형식
Table 3.1 Instruction format of processor

	15 12	11 8	7 4	3 0
	OP Code	Dest. Reg	Src. Reg1	Src. Reg2
LD.V	0 0 0 0	0 x x x	0 0 0 0	0 0 0 0
LD.M	0 0 0 1	0 x 0 0	0 0 0 0	0 0 0 0
OUT.V	0 0 1 0	0 0 0 0	0 0 0 0	0 x x x
OUT.M	0 0 1 1	0 0 0 0	0 0 0 0	0 x 0 0
FPM.V	0 1 0 0	0 x x x	0 x 1 1	0 x x x
FPM.M	0 1 0 1	0 x 0 0	0 x 1 1	0 x 0 0
FPA.V	0 1 1 0	0 x x x	0 x x x	0 x x x

①LD.V
■ 형식 : LD.V Vd
■ 동작 : 외부로부터 입력되는 벡터를 레지스터 Vd에 저장한다. I/O Port로부터 3개의 32-bit 데이터를 받아 벡터형식으로 변환 후, Z 버스를 통해 레지스터 파일로 전송한다.

②LD.M
■ 형식 : LD.M Md
■ 동작 : 외부로부터 입력되는 행렬을 레지스터 Md에 저장한다. 기본적인 동작은 LD.V와 동일하며, 입력은 4 cycle 동안 열벡터의 형태로 이루어진다.

③OUT.V
■ 형식 : OUT.V Vs
■ 동작 : 레지스터 Vs의 벡터를 외부로 출력한다. V 버스를 이용하여 I/O Port로 전송한다.

④OUT.M
■ 형식 : OUT.M Ms
■ 동작 : 레지스터 Ms의 행렬을 외부로 출력한다. 기본적인 동작은 OUT.V와 동일하며, 출력은 4 cycle동안 열벡터의 형태로 이루어진다.

⑤FPM.V
■ 형식 : FPM.V Vd, Ms, Vs
■ 동작 : 행렬 Ms와 벡터 Vs를 곱한 결과 벡터를 레지스터 Vd에 저장한다. 이 명령어는 Processing Unit이 수행하는 가장 기본적인 연산이다. 부동 소수점 수 곱셈기는 2단계, 부동 소수점 수 덧셈기는 3단계의 파이프라인 구조를 가지므로 명령어 해석과 레지스터 파일에 저장하는 과정을 포함하여 11 cycle의 수행시간을 갖는다. 실제 응용 시 추가적인 변환을 위해서 소스 벡터와 결과 벡터는 다른 행렬 레지스터의 벡터만 허용하도록 설계하였다.

⑥FPM.M
■ 형식 : FPM.M Md, Ms1, Ms2
■ 동작 : 행렬 Ms1과 Ms2를 곱한 결과행렬을 레지스터 Md에 저장한다. 이 명령어의 경우 내부적으로 행렬/벡터 곱셈의 반복이다. 처리유닛이 파이프라인 구조이므로 처음 4 cycle동안 두 번째 행렬의 각 열벡터가 순서대로 입력되어 중첩된 곱셈을 수행한다. 총 수행시간은 14 cycle이다.

⑦FPA.V
■ 형식 : FPA.V Vd, Vs1, Vs2
■ 동작 : 벡터 Vs1과 Vs2의 원소 간 덧셈을 수행한 후, 결과를 레지스터 Vd에 저장한다. 첫 번째 피 연산벡터를 접근하기 위해 곱셈명령과 C4 버스를 공유하고 있다. 총 수행시간은 6 cycle이다.

3.3 레지스터 파일

실제 변환행렬은 순차적인 곱셈을 통해 완성된다. 또한 하나의 변환행렬에 대해 많은 위치의 변환이 요구되므로 내부적으로 이를 저장하기 위한 레지스터를 설계하였다. 모든 연산은 32-bit의 부동 소수점 수를 원소로 하는 3×1 열벡터를 기본으로 하므로 벡터 레지스터의 형식을 갖는다.

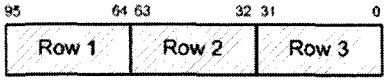

그림 3.2 벡터 레지스터의 형식
Fig 3.2 Format of vector register

그림 3.2에서 보듯이 벡터 레지스터는 96-bit이며, 상위부터 열벡터의 각 행원소를 나타낸다. 이런 형식의 벡터 레지스터를 가지고 V0부터 V7까지 8개로 구성되는 벡터 레지스터 파일(vector register file)을 설계하였다.

여기서 한 가지 문제가 발생하는 데, 설계한 프로세서는 벡터를 기본 연산단위로 하지만 행렬단위의 접근 또한 요구된다. 따라서 이를 위해 일종의 레지스터 윈도우(register window)의 개념을 적용하여 벡터와 행렬에 대해 각각 독립적인 접근이 가능하도록 하였다. 즉, 하나의 행렬은 4개의 열벡터로 구성되므로 V0(000)부터 V3(011)까지를 행렬 M0,

V4(100)부터 V7(111)까지를 행렬 M1로 mapping하여, 행렬 연산 시 명령어 레지스터 필드의 하위 세 번째 비트로 피연산행렬을 접근할 수 있다. 그림 3.3은 이러한 구조를 보여준다.

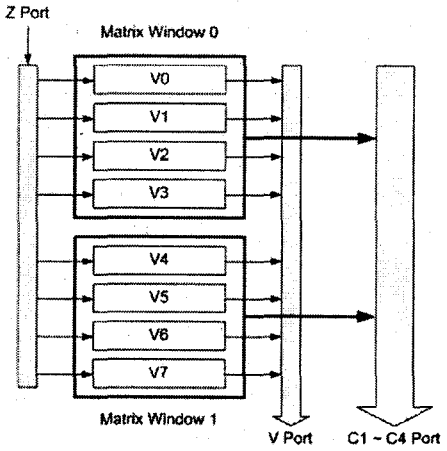

그림 3.3 벡터 레지스터 파일의 구조
Fig 3.3 Structure of vector register file

레지스터 파일은 5개의 읽기 포트와 1개의 쓰기 포트를 갖는다. 각각의 레지스터들은 벡터연산 시 V 포트를 통해 개별적으로 출력되며, 행렬연산 시 C1 ~ C4 포트를 통해 행렬단위로 출력된다. 모든 연산에 있어 결과는 벡터단위이므로 입력은 Z 포트로 하나로 충분하다.

3.4 부동 소수점 수 곱셈기

기본적으로 부동 소수점 수의 연산은 수의 형식상 부호(sign), 지수(exponent), 가수(fraction)의 각 필드별 처리를 통해 수행된다. 따라서 보통의 정수연산에 비해 복잡하며, 몇 개의 단계를 거치는 것이 일반적이다. 본 논문에서는 빠른 연산을 위해 그 동안 제안된 많은 알고리듬을 조합하여 설계하였다.

부동 소수점 수의 곱셈은 다른 연산에 비해 비교적 간단하며, 가수의 경우 보통 다음과 같은 4단계의 과정을 거친다.[7]

①가수의 곱셈
②정규화(normalization)
③라운딩(rounding)
④정규화(normalization)

첫 번째 정규화는 곱셈 시 발생할 수 있는 1 bit의 오버플로우를 위한 것이다. 곱셈의 결과는 항상 두 배의 bit수를 가지므로 라운딩을 통해 이를 처리하며, 이 경우 발생할 수 있는 추가적인 오버플로우를 위해 마지막 정규화 과정이 필요하다.

이 과정을 병렬적으로 처리하기 위한 많은 알고리듬이 제안되었으며 여기서는 'round to nearest up'방식으로 라운딩을 수행한 후, 마지막에 한 비트의 수정만으로 'round to nearest even'방식의 결과를 얻을 수 있는 알고리듬을 적용하였다.[8] 이 알고리듬은 곱셈의 마지막에 필요한 carry propagation addition 단계와 rounding 단계를 하나로 합치고, 간단히 멀티플렉서를 이용하여 정규화를 수행한다. 따라서 하드웨어를 줄이는 동시에 전체 단계를 줄이므로 보다 빠른 시간 내에 부동 소수점 수 곱셈을 수행할 수 있다. 그림 3.4는 곱셈기의 전체 구조를 보여주고 있으며, 2단계의 파이프라인 구조로 설계하였다.

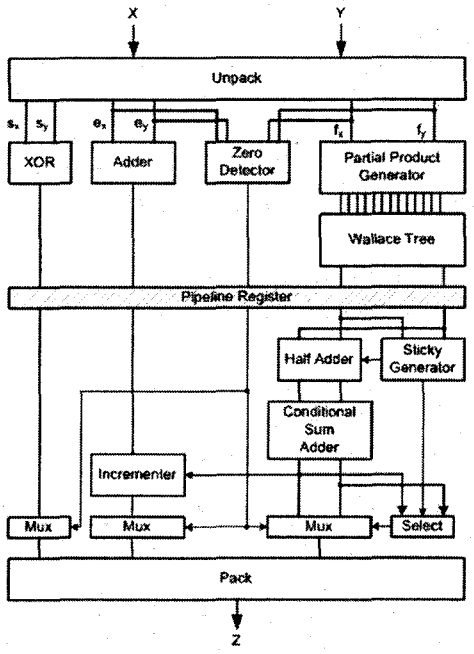

그림 3.4 부동 소수점 수 곱셈기의 블록 다이어그램
Fig 3.4 Block diagram of floating-point multiplier

▶1 단계 : 가수의 곱셈

먼저 입력되는 두 수를 각 필드별 처리를 위해 분리한다. 곱셈 결과의 부호는 XOR 연산으로 간단히 해결하며, 지수는 입력되는 두 지수의 합으로 처리한다. 이 때 각 지수가 127만큼 초과된 상태이므로 더한 후, 127을 빼주어야 한다. 아울러 0의 처리를 위해 입력되는 수가 0인지를 판단하여 마지막 단계에서 이를 처리하도록 한다.

가수의 곱셈은 24-bit의 unsigned integer multiplier를 이용한다. 일반적으로 곱셈의 성능은 부분 곱(partial product)의 개수를 줄임과 동시에 생성된 부분 곱의 덧셈을 빨리 처리함으로써 향상된다. 이를 위해 'Radix-4 Modified Booth's Algorithm'을 적용하여 부분 곱의 개수를 줄였으며, Wallace Tree를 이용해 그 덧셈을 병렬적으로 처리하였다. Wallace Tree는 보통 3:2 counter로 구성되나 본 논문에서는 4:2 counter를 이용하여 단계를 줄였다.[9]

▶2 단계 : 라운딩 및 정규화

Wallace Tree를 통한 결과는 carry save 형태의 두 수이다. 따라서 이를 최종적으로 처리하기 위한 carry

propagation 덧셈이 필요하다. 앞서 설명한 바와 같이 이를 round to nearest up 방식의 라운딩과 같이 처리한다. 오버플로우가 발생하는 경우 1 bit의 오차로 인해 추가적인 라운딩이 필요하므로 conditional sum adder로 두 가지 경우에 대한 값을 구한 후, 마지막의 멀티플렉서를 통해 정규화를 수행한다. 아울러 오버플로우가 발생한 경우 지수에 1을 더하여 조정한다. 본 논문에서는 round to nearest even 방식을 채택하였으므로 최종적으로 이에 대한 보정 또한 요구된다.

3.5 부동 소수점 수 덧셈기

부동 소수점 수의 덧셈은 가장 복잡한 과정을 필요로 하며, 따라서 다양한 알고리듬과 구조가 제안되었다. 일반적으로 다음과 같이 6 단계의 과정을 거친다.[7]

①정렬(alignment)
②가수의 덧셈/뺄셈
③변환(conversion)
④정규화(normalization)
⑤라운딩(rounding)
⑥정규화(normalization)

덧셈 시 지수가 같아야 하므로 지수의 차이를 구하고, 작은 지수를 가지는 가수를 차이만큼 정렬한다. 부동 소수점 수는 signed magnitude 방식의 표현이므로 음의 결과는 양으로 변환하는 과정이 필요하다. 정규화 후 오차를 줄이기 위해 라운딩을 수행하고, 추가적으로 발생하는 오버플로우를 위해 마지막 정규화 과정을 거친다.

본 논문에서는 이 단계를 줄이기 위해 두 가지 알고리듬을 적용하였다. 우선 가수 간 자리교환(swap)을 통해 뺄셈 후 양수로의 변환과정을 없앴다. 즉, 정렬되는 가수를 항상 두 번째 피연산자로 함으로써 결과는 항상 양수가 되도록 하였다. 문제는 두 수의 지수가 같은 경우인데, 이는 가수 간 직접비교를 통해 실제 뺄셈 시 선택적으로 complement를 수행하도록 하였다. 이러한 방식은 실제로 많은 프로세서에서 채택하고 있다. 다음으로 연산의 결과 요구되는 정규화와 라운딩은 동시에 일어나지 않는다는 특성을 이용하여 두 단계를 동시에 처리한 후, 최종적으로 선택하도록 하였다. 라운딩 후 필요한 정규화는 1 bit의 right shift이므로 역시 최종 선택 시 처리할 수 있다.[10][11][12] 그림 3.5는 덧셈기의 전체 구조를 보여주고 있으며, 3단계의 파이프라인 구조로 설계하였다.

▶1단계 : 정렬

우선 두 수의 지수의 차이를 구한다. 큰 지수가 연산 결과의 지수이므로 선택하여 다음 단계로 보내고, 작은 지수의 가수를 자리교환을 통해 지수의 차이만큼 정렬한다. 또한 0의 처리를 위해 입력되는 두 수가 0인지를 판단하여 다음 단계의 연산 시 처리하도록 한다.

두 수의 부호와 수행연산, 지수의 차이와 가수의 비교결과를 이용해 연산결과의 부호를 결정하고, 다음 단계에서 필요한 각종 제어신호를 생성한다.

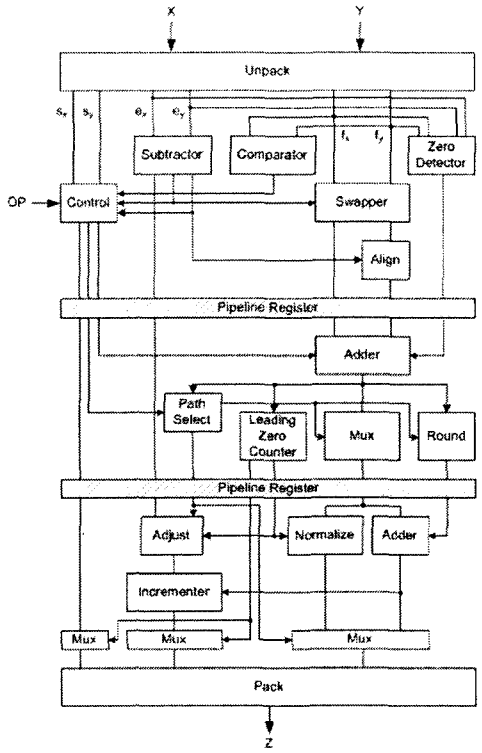

그림 3.5 부동 소수점 수 덧셈기의 블록 다이어그램
Fig 3.5 Block diagram of floating-point adder

▶2 단계 : 덧셈 / 뺄셈

앞 단계에서 결정된 제어신호를 이용해 실제 덧셈 혹은 뺄셈을 수행한다. 연산의 결과는 부호 확장 비트와 G, R, S 비트를 포함하여 28-bit이다.

다음 단계의 정규화를 위해 선행 0의 개수를 결정하고, 라운딩을 위해 필요한 1 bit의 추가 shift를 수행하며 라운딩 결정신호를 생성한다. 정규화와 라운딩이 동시에 처리되므로 연산의 결과가 어느 경우에 해당되는지를 결정하여 적절한 경로 선택(path selection)을 한다.

▶3 단계 : 정규화 / 라운딩

정규화와 라운딩을 수행한 후, 앞 단계에서 결정된 경로 선택 신호를 이용하여 최종의 가수를 선택한다. 지수도 마찬가지로 각 경우에 해당하는 조정을 거치고, 라운딩 후 오버플로우가 발생하였다면 추가적으로 1을 더한다. 연산 결과가 0인 경우 부호와 지수는 멀티플렉서를 이용하여 처리한다.

4. 검증 및 성능평가

설계한 프로세서는 Altera사의 MAX+plusⅡ를 이용, FPGA로 구현하여 동작을 검증하였다. FLEX10KE 계열의 칩에서 최대 50ns의 지연시간을 보였고, 실제 시뮬레이션은 100ns의 clock 주기를 바탕으로 진행하였다. 실제 좌표변환 알고리듬의 구현을 통해 프로세서의 전체 동작을 검증하였다. 구하고자 하는 변환 알고리듬은 다음과 같다.

$$T = Trans(4, -3, 7) \cdot Rot(y, 90^\circ) \cdot Rot(z, 90^\circ) \quad \text{(식 4.1)}$$

먼저 기준 프레임에 대하여 이동변환을 수행한 후, 대상 프레임의 y축과 z축에 대하여 각각 90도의 회전변환을 수행한다. 이 변환행렬에 의해 벡터 $[7\ 3\ 2]^T$가 실제 변환되는 과정을 수식으로 나타내면 다음과 같다.

$$\begin{pmatrix}6\\4\\10\\1\end{pmatrix} = \begin{pmatrix}1&0&0&4\\0&1&0&-3\\0&0&1&7\\0&0&0&1\end{pmatrix} \cdot \begin{pmatrix}0&0&1&0\\0&1&0&0\\-1&0&0&0\\0&0&0&1\end{pmatrix} \cdot \begin{pmatrix}0&-1&0&0\\1&0&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} \cdot \begin{pmatrix}7\\3\\2\\1\end{pmatrix} \quad \text{(식 4.2)}$$

즉, 벡터 $[7\ 3\ 2]^T$는 좌표변환에 의해 $[6\ 4\ 10]^T$의 위치로 이동하는 것이다. 이를 설계한 프로세서로 구현하기 위해서 다음과 같은 명령어 sequence를 구성하였다.

표 4.1 좌표변환 알고리듬의 적용

Table 4.1 Application of coordinate transformation algorithm

	명령어		설명
1	LD.M	M0	기준 프레임 입력
2	LD.V	V7	이동변환 행렬 입력
3	FPA.V	V3, V3, V7	이동변환
4	LD.M	M1	회전변환 행렬 입력
5	FPM.M	M0, M0, M1	회전변환
6	LD.M	M1	회전변환 행렬 입력
7	FPM.M	M0, M0, M1	회전변환
8	LD.V	V7	대상 벡터 입력
9	FPM.V	V7, M0, V7	최종변환
10	OUT.V	V7	결과 출력

위의 알고리듬을 실제 수행한 결과가 그림 4.1이다. 총 10개의 명령어로 순차적으로 이루어지는 좌표변환을 수행하고, 그 결과를 확인할 수 있다. 부동 소수점 수로 6은 40C00000H, 4는 40800000H, 10은 41200000H이다.

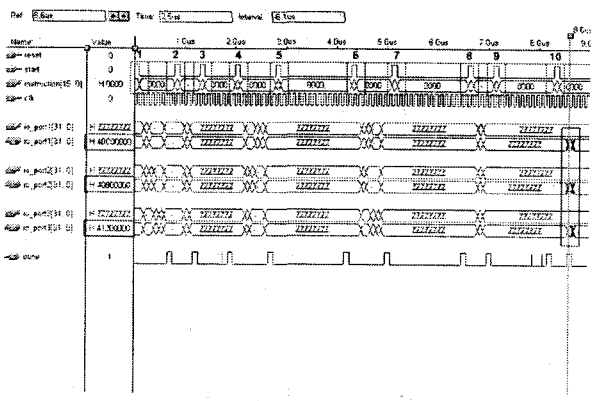

그림 4.1 좌표변환 알고리듬의 시뮬레이션 결과

Fig 4.1 Simulation result of coordinate transformation algorithm

프로세서의 성능평가는 독립적인 하나의 부동 소수점 수 곱셈기와 덧셈기를 탑재한 범용의 유니프로세서(uni-processor)를 기준으로 하였다. 부동 소수점 수 곱셈기와 덧셈기는 설계한 프로세서와 같은 구조 및 지연시간을 갖고 있으며, 그 외 성능의 향상을 위한 다른 방법들은 적용되지 않았다고 가정한다. 이상의 조건을 바탕으로 하여 Homogeneous Transformation Matrix의 곱셈을 각각 수행한 결과는 다음과 같다.

표 4.2 프로세서의 성능비교

Table 4.2 Performance comparison of processor

	설계한 병렬구조 프로세서	범용 유니프로세서
명령어 Issue	1회	곱셈 : 36회 덧셈 : 27회
Homogeneous Transformation Matrix 곱셈의 수행 cycle	11 clock cycles	143 clock cycles

수행 cycle은 명령어 해석과 레지스터 저장 단계를 제외하고 실제 곱셈이 수행되는 단계만 고려하였다. 위 표에서 보듯이 병렬구조의 하드웨어를 이용하여 처리할 경우, 범용의 유니프로세서를 기반으로 하는 순차적 처리에 비해 수행 cycle을 기준으로 약 13배의 성능의 향상을 기대할 수 있다. 게다가 소프트웨어적 처리는 메모리 접근이 요구되므로 이를 포함한다면, 전체 수행시간에 있어 엄청난 향상을 보일 것이다. 또한 실제 변환은 연속적인 곱셈으로 이루어지므로 이 구조를 이용할 경우, 훨씬 유연한 처리가 가능하다고 할 수 있다.

5. 결 론

Homogeneous Transformation Matrix는 3차원 공간에서 물체의 위치를 표현하는 방법으로 연속적인 곱셈을 통해 좌표변환을 수행한다. 따라서 실제 응용에 있어서는 매우 빠른 행렬곱셈이 요구된다. 이를 위해 행렬곱셈의 병렬연산 알고리듬을 도출한 후, 전용의 하드웨어로 구현하였다.

프로세서의 구조적인 특징으로 우선 벡터 프로세서의 개념을 확장하여 병렬성을 최대한 이용하였다. 따라서 하드웨어가 많이 소요되지만 매우 빠른 연산결과를 얻을 수 있었다. 행렬곱셈의 경우, 단 14 clock cycle만으로 처리되므로 소프트웨어는 물론이고 기존의 다른 구조에 비해 엄청난 성능의 향상을 기대할 수 있다. 또한 독립적인 명령어와 내부의 레지스터 파일을 가짐으로 인해 순차적으로 이어지는 다양한 변환 알고리듬의 적용이 가능하다.

설계한 프로세서는 마이크로프로그래밍(micro-programming) 방식의 제어구조를 채택함으로써 향후 기능의 확장을 통해 범용 프로세서에 부가되는 보조 프로세서나 다양한 독립적인 응용 프로세서로서의 역할을 기대할 수 있겠다.

참 고 문 헌

[1] K. Diefendorff and P. K. Dubey, "How multimedia workloads will change processor design," IEEE Computer Magazine, vol. 30, no. 9, pp. 43-45, September 1997.

[2] C. E. Kozyrakis and D. A. Patterson, "A new direction for computer architecture research," IEEE Computer Magazine, vol. 31, no. 11, pp. 24-32, November 1998.

[3] Ponnuswamy Sadayappan, et al, "A Restructurable VLSI Robotics Vector Processor Architecture for Real-Time Control," IEEE Transactions on Robotics and Automation, vol. 5, no. 5, October 1989.

[4] IEEE Standards Board, "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Std 754-1985.

[5] Saeed B. Niku, Introduction to Robotics: Analysis, Systems, Applications, Prentice Hall, p. 38, 2001.

[6] G. J. Myeres, Digital System Design with LSI Bit-Slice Logic, John Wiley & Sons, New York, 1980.

[7] B. Parhami, Computer Arithmetic: Algorithms and hardware Designs, Oxford University Press, New York, 2000.

[8] Mark R. Santoro, et al, "Rounding Algorithm for IEEE Multipliers," Proceedings of the 9th Symposium on Computer Arithmetic, 1989.

[9] Gary W. Bewick, Fast Multiplication: Algorithms and Implementation, Ph.D. dissertation, Stanford University, February 1994.

[10] Nobuhiro Ide, et al, "A 320-MFLOPS CMOS Floating-Point Processing Unit for Superscalar Processors," IEEE Journal of Solid-State Circuits, vol. 28, no. 3, March 1993.

[11] Nhon T. Quach and Michael J. Flynn, "An Improved Algorithm for High-Speed Floating-Point Addition," Technical Report: CSL-TR-90-442, Stanford University, August 1990.

[12] Nhon T. Quach and Michael J. Flynn, "Design and Implementation of the SNAP Floating-Point Adder," Technical Report: CSL-TR-91-501, Stanford University, December 1991.

저 자 소 개

권 두 울 (權 斗 兀)

2003년 중앙대 전자전기공학부 (학사)
2005년 중앙대 전자전기공학부 (석사)
Tel : 02) 825-1644
E-mail : twoducks@hanmail.net

정 태 상 (鄭 台 相)

1978년 서울대 전기공학과 (학사)
1982년 미국 Ohio 주립대 (석사)
1985년 미국 Ohio 주립대 (박사)
1986~1992년 미국 Kentucky 대 조교수
1992~현재 중앙대 전자전기공학부 교수
Tel : 02) 820-5321
Fax : 02) 823-2492
E-mail : tschung@digital.cau.ac.kr