Hardware Implementation of Multiple Vector Quantization Decoder

Noritaka Shigei†, Hiromi Miyajima†, Shingo Hashiguchi†, Michiharu Maeda††, Lixin Ma†††

†Kagoshima University, Kagoshima, Japan
††Fukuoka Institute of Technology, Fukuoka, Japan
†††University of Shanghai for Science and Technology, Shanghai, China

Summary
Vector Quantization (VQ) is successfully applied to data compression. In image compression, Multiple-VQ with *Index Inference* (MVQII) can provide a much better restored image quality than conventional VQ methods. However, the index inference increases the computational complexity of the decoding process in proportion to the number of weights. In this paper, in order to accelerate the decoding process of MVQII, we present three types of digital circuit design for the decoder. The presented designs are implemented on FPGAs and are evaluated in terms of operating speed, processing time. The results show that two versions achieve good improvement in processing speed at reasonable expenses of circuit size and that the fastest version is nearly ten times faster than a software implementation on a modern standard computer.

Key words:
Vector quantization, decoder, multiple vector quantization, digital circuit, FPGAs.

1. Introduction

Vector Quantization (VQ) is a process for approximating a large set of input vectors by a smaller set of weight vectors. VQ is a useful technique in many applications such as data compression, data mining and pattern recognition [1,2,4]. VQ is successfully applied to data compression [8]. However, the quality of the restored data greatly depends on the number of weights (codebook size) and the dimension of weights (block size). One of the methods relaxing the trade-off is Multistage Residual VQ (MRVQ), in which multiple quantizers are concatenated in series [2]. In MRVQ, the first stage quantizer operates on the input vectors, and the second stage operates on the errors between the input vector and the first stage output. MRVQ is a serial approach. On the other hand, we have proposed multiple-VQ (MVQ) as a parallel approach [3]. The MVQ method performs independently multiple VQ processes, and combines the independent low quality results into a high quality one. We have shown that, with the useful technique *index inference*, MVQ with Index Inference (MVQII) is effective in data compression [5,6]. However, the index inference increases the computational complexity of decoder in proportion to the codebook size. For example, when decoders with κ codebook vectors are implemented on a CPU, an MVQII decoder has κ times lower throughput than conventional VQ decoders.

Digital circuit implementation is effective in terms of processing speed and power consumption when compared with software implementation on a general purpose PC. Some systems based on VQ have been implemented in digital circuits [9,10,11]. In [9], a speaker identification system based on VQ has been implemented in FPGAs. In [10], a processor for self-organizing map (SOM), which is one of VQ methods[4], has been implemented in FPGAs. In [11], a systolic architecture of SOM has been proposed and also implemented in FPGAs.

In this paper, in order to accelerate the decoding process of MVQII, we study hardware implementation of MVQII decoders. Specifically, we present three types of digital circuit design for the decoder. The first one is a naive implementation, and the second and third ones are improved versions, respectively, with “pipeline” and “parallelization”. The presented designs are implemented in FPGAs and are evaluated in terms of operating speed, processing time and circuit size. The results show that the last two versions achieve good improvement in processing speed at reasonable expenses of circuit size and that the parallel version is nearly ten times faster than a software implementation on a modern standard computer.

The rest of this paper is organized as follows. Section 2 describes VQ, MVQ and MVQII, and shows some simulation results for performance comparison. Section 3 presents our proposed digital circuit designs for MVQII decoder. Section 4 shows the performance evaluation results. Finally, section 5 concludes the paper and presents future works.

2. Vector Quantization and Multiple Vector Quantization

2.1 Vector Quantization

Vector Quantization (VQ) is to approximate a large set of input vectors $X=\{\boldsymbol{x}_1,\cdots,\boldsymbol{x}_\nu\}$ by a smaller set of weight vectors $W=\{\boldsymbol{w}_1,\cdots,\boldsymbol{w}_\kappa\}$, where $\boldsymbol{x}_i,\boldsymbol{w}_j\in\Re^n$ are n-dimensional Euclidean vectors and X is a random sample from a probability density function (PDF) $p(\boldsymbol{x})$. W and

Manuscript received November 5, 2008
Manuscript revised November 20, 2008

$\boldsymbol{w}_j$ are also referred to, respectively, as codebook and codebook vector. In VQ, an input vector $\boldsymbol{x} \in X$ is replaced with a weight vector $\boldsymbol{w}_{k_{win}} \in W$ such that $d(\boldsymbol{x}, \boldsymbol{w}_{k_{win}}) = \min_{k \in \{1,\cdots,\kappa\}} d(\boldsymbol{x}, \boldsymbol{w}_k)$, where $d(\boldsymbol{x}, \boldsymbol{w}) = \| \boldsymbol{x} - \boldsymbol{w} \|$. In other word, the VQ process divides the input space $\Re^n$ into κ sub-spaces $S_1, \cdots, S_\kappa$ such that $S_k = \{\boldsymbol{x} \in \Re^n \mid d(\boldsymbol{x}, \boldsymbol{w}_k) \le d(\boldsymbol{x}, \boldsymbol{w}_l), k \ne l\}$. The approximation accuracy of VQ is evaluated in terms of the average distortion error $E = \frac{1}{\nu} \sum_{k \in \{1,\cdots,\kappa\}} \sum_{x \in S_k} d(\boldsymbol{x}, \boldsymbol{w}_k)$.

A fundamental method minimizing the error E is Competitive Learning (CL), which is based on gradient descent [4]. The CL procedure iterates a simple adaptation step. At t-th iteration, the CL procedure calculates the closest weight (called *winner*) $\boldsymbol{w}_{k_{win}}$ to a given input vector $\boldsymbol{x} \in X$, and then updates $\boldsymbol{w}_{k_{win}}$ as follows:

$$\boldsymbol{w}_{k_{win}} \leftarrow \boldsymbol{w}_{k_{win}} + \varepsilon(t)(\boldsymbol{x} - \boldsymbol{w}_{i_{win}}), \qquad (1)$$

where $d(\boldsymbol{x}, \boldsymbol{w}_{k_{win}}) = \min_{k \in \{1,\cdots,\kappa\}} d(\boldsymbol{x}, \boldsymbol{w}_k)$ and $\varepsilon(t)$ is a learning rate decreasing with t.

2.2 VQ based Image Compression

This section describes single-VQ based data compression (SVQ). We assume G-bit gray images of $M \times N$ pixels. Let $p_{i,j} \in \{0, \cdots, 2^G - 1\}$ ($i \in \{1, \cdots, M\}$, $j \in \{1, \cdots, N\}$) be the gray level of the pixel at coordinates (i, j). Then an image is represented as an $M \times N$ matrix $P = (p_{i,j})$. The algorithm SVQ is as follows.

Algorithm SVQ
Step 1 (Input Preparation)

Given an input image P. The $M \times N$ pixels in P are divided into $(M \times N)/(J \times K)$ blocks of size $J \times K$. The blocks are represented as JK-dimensional vectors $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{MN/(JK)}$.

Step 2 (Vector Quantization)

The set of weight vectors $W = \{\boldsymbol{w}_1, \cdots, \boldsymbol{w}_\kappa\}$, called codebook, is trained by using the set of vectors $X = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{MN/(JK)}\}$ as input data. The training is performed by Eq.(1).

Step 3 (Index Calculation)

For each $i \in \{1, \cdots, MN/(JK)\}$, the index number l_i is calculated, where $d(\boldsymbol{x}_i, \boldsymbol{w}_{l_i}) = \min_{j \in \{1,\cdots,\kappa\}} d(\boldsymbol{x}_i, \boldsymbol{w}_j)$.

End of SVQ.

The compressed data is composed of the codebook W and the index sequence $L = (l_1, \cdots, l_{MN/(JK)})$. From W and L, an image is restored. In the restored image, each block $\boldsymbol{x}_i \in X$ in the original image is replaced with $\boldsymbol{w}_{l_i}$. The quality of the restored image is evaluated in terms of mean square error (MSE) as follows:

$$MSE(X, W, L) = \frac{1}{MN} \sum_{i \in \{1,\cdots,MN/(JK)\}} d(\boldsymbol{x}_i, \boldsymbol{w}_{l_i})$$

2.3 Multiple-VQ based Image Compression

In this subsection, we describe multiple-VQ (MVQ) and *index inference* for image compression. The fundamental idea of MVQ is that a high quality image can be created by averaging multiple low quality images that independently restored from different pairs of codebook and index sequence. The MVQ algorithm for compression phase is as follows.

Algorithm MVQ
Step 1 (Input Preparation)

From the input image P, two input data sets $X^{(0)}$ and $X^{(1)}$ are generated as follows:

$$\begin{aligned} X^{(c)} &= \{\boldsymbol{x}_i^{(c)} \mid i \in \{1, \cdots, \frac{MN}{J_c K_c}\}\} \text{ for } c \in \{0,1\}, \\ \boldsymbol{x}_i^{(c)} &= (p_{\phi(i,c,1),\psi(i,c,1)}, \cdots, p_{\phi(i,c,J_c),\psi(i,c,1)}, \\ &\quad p_{\phi(i,c,1),\psi(i,c,2)}, \cdots, p_{\phi(i,c,J_c),\psi(i,c,2)}, \\ &\quad \cdots \\ &\quad p_{\phi(i,c,1),\psi(i,c,K_c)}, \cdots, p_{\phi(i,c,J_c),\psi(i,c,K_c)}), \end{aligned} \qquad (2)$$

where

$\phi(i, c, j) = (((i-1) \bmod \frac{M}{J_c}) J_c + c + j - 1) \bmod M + 1$ and

$\psi(i, c, j) = (\lfloor (i-1) / \frac{M}{J_c} \rfloor K_c + c + j - 1) \bmod N + 1$. **Fig. 1** is a schematic explanation of $\boldsymbol{x}_i^{(c)}$.

Step 2 (Vector Quantization)

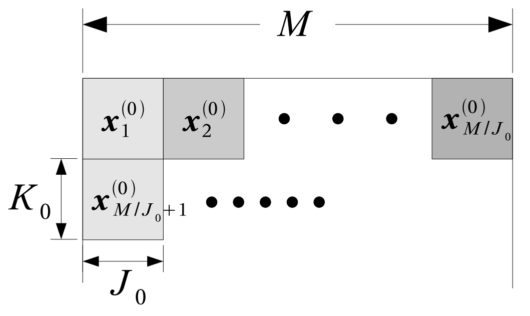
Perform VQ twice to generate two codebooks $W^{(0)}$ by using data sets $X^{(0)}$ and $X^{(1)}$, respectively. The training is performed by Eq. (1).

Step 3 (Index Calculation)

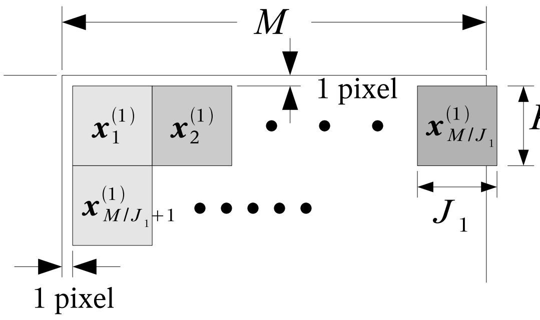
The index sequence $L^{(0)} = (l_1^{(0)}, \cdots, l_{MN/(J_0 K_0)}^{(0)})$ is calculated from $W^{(0)}$.

End of MVQ.

In Step 2, two codebooks $W^{(0)}$ and $W^{(1)}$ are generated from different input data sets $X^{(0)}$ and $X^{(1)}$, respectively. The use of different data sets is an essential requirement for obtaining a good quality [4].



(a) For $X^{(0)}$.



(b) For $X^{(1)}$.

Fig. 1 Segmentation of P for generating $X^{(c)}$

In Step 3, only $L^{(0)}$ is calculated, and the compressed data are $W^{(0)}$, $W^{(1)}$ and $L^{(0)}$. If, in addition to the compressed data, $L^{(1)}$ is used, a better quality image can be restored. However, in this case, the compressed data size significantly increases, and as a result, the restored image quality is too low for a fixed data size. Instead, the lost data $L^{(1)}$ is restored by a decoder side technique *index inference* [11]. The algorithm is as follows.

Algorithm MVQ with Index Inference

Step 1 (Image Restoration by SVQ)

Restore an image $\tilde{P}^{(0)} = (\tilde{p}_{i,j}^{(0)})$ from $W^{(0)}$ and $L^{(0)}$.

Step 2 (Input Preparation for Inference)

From the restored image $\tilde{P}^{(0)}$, an input data set $\tilde{X}^{(1)} = \{\tilde{\boldsymbol{x}}_1^{(1)}, \cdots, \tilde{\boldsymbol{x}}_{MN/(J_1K_1)}^{(1)}\}$ is generated as in Eq. (2).

Step 3 (Index Calculation)

For each $i \in \{1, \cdots, MN/(J_1K_1)\}$, the index number $l_i^{(1)}$ is calculated, where $d(\tilde{\boldsymbol{x}}_i^{(1)}, \tilde{\boldsymbol{w}}_{l_i^{(1)}}^{(1)}) = \min_{j\in\{1,\cdots,\kappa\}} d(\tilde{\boldsymbol{x}}_i^{(1)}, \boldsymbol{w}_j^{(1)})$.

End of Index Inference.

Now, after index inference, we have $W^{(0)}$, $W^{(1)}$, $L^{(0)}$ and $L^{(1)} = (l_1^{(1)}, \cdots, l_{MN/(J_1K_1)}^{(1)})$. Next, two images $\tilde{P}^{(0)}$ and $\tilde{P}^{(1)}$ are independently restored from two pairs $(W^{(0)}, L^{(0)})$ and $(W^{(1)}, L^{(1)})$, respectively. And then, a restored image $\tilde{P} = (\tilde{p}_{j,j})$ is generated by combining the two restored images as follows.

$$\tilde{p}_{i,j} = \frac{1}{2}(\tilde{p}_{i,j}^{(0)} + \tilde{p}_{i,j}^{(1)}), \tag{3}$$

where $\tilde{P}^{(c)} = (\tilde{p}_{i,j}^{(c)})$ for $c \in \{0,1\}$.

In the following, the method described in this section is referred as MVQ with Index Inference (MVQII).

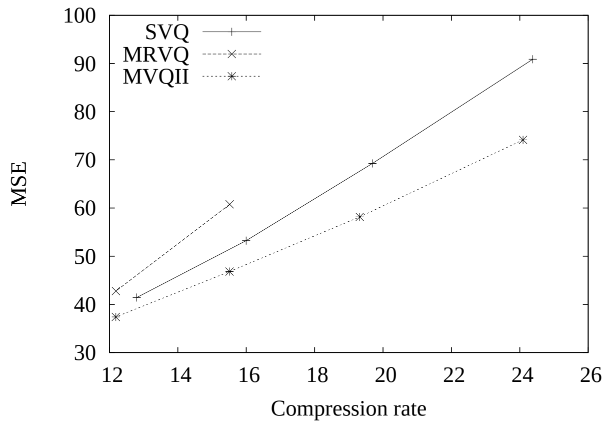
(a) Lena (b) Goldhill

Fig. 2 Test images.

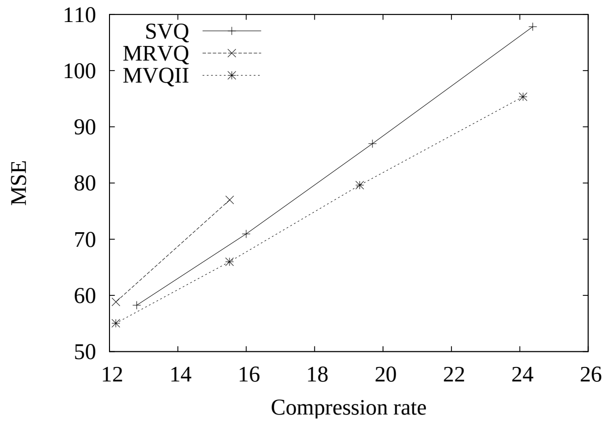
2.4 Performance Comparison

In order to demonstrate the effectiveness of MVQII, we show some simulation results on SVQ, MRVQ (Multistage Residual VQ) and MVQII. SVQ is the algorithm described in Sect. 2.2. MRVQ is performed by a software package QccPack[7]. For MRVQ, the number of stages is two and each stage uses the same codebook size. MVQII is performed with $J_0 \times K_0 = 4 \times 4$, $J_1 \times K_1 = 2 \times 2$ and $\kappa_0 = \kappa_1$. The used test images are shown in **Fig. 2**. The two images consists of 512×512 pixels of 256 gray level, that is $M = N = 512$.

The simulation results are shown in **Fig. 3**, where the compression rate is the ratio of the original data size ($512 \times 512 \times 8$ bits) to the compressed data size (the total size of codebooks and index sequence). For all images, MVQII achieves the best performance among all the methods when the compression rate is high.



(a) For Lena.



(b) For Goldhill.

Fig. 3 MSE versus compression rate for SVQ, MRVQ and MVQII.

3. Hardware Implementation of MVQII Decoder

MVQII described in the previous section requires $O(\kappa_1)$ times as more time as SVQ method, because unlike SVQ, MVQII requires finding nearest weights for $W^{(1)}$ as in Step 3. In this section, in order to accelerate the decoding process of MVQII, we present its hardware implementation.

3.1 MVQII Algorithm for Hardware Implementation

In this subsection, we rewrite the MVQII algorithm into a form suitable for hardware implementation. As a preliminary step, we give some definitions.

$$\begin{aligned}\tilde{\boldsymbol{x}}_i^{(c)} = (&\tilde{p}_{\phi(i,c,1),\psi(i,c,1)},\cdots,\tilde{p}_{\phi(i,c,J_c),\psi(i,c,1)},\\ &\tilde{p}_{\phi(i,c,1),\psi(i,c,2)},\cdots,\tilde{p}_{\phi(i,c,J_c),\psi(i,c,2)},\\ &\cdots\\ &\tilde{p}_{\phi(i,c,1),\psi(i,c,K_c)},\cdots,\tilde{p}_{\phi(i,c,J_c),\psi(i,c,K_c)}),\end{aligned}$$

where $\tilde{P} = (\tilde{p}_{j,j})$ is the matrix that will store the final restored image.

When $J = J_0 = J_1$ and $K = K_0 = K_1$, the rewritten version is given as follows.

Algorithm MVQII for Hardware Implementation
Input:

$W^{(0)} = \{\boldsymbol{w}_1^{(0)},\cdots,\boldsymbol{w}_{\kappa_0}^{(0)}\}$, $W^{(1)} = \{\boldsymbol{w}_1^{(1)},\cdots,\boldsymbol{w}_{\kappa_1}^{(1)}\}$,

$L^{(0)} = (l_1^{(0)},\cdots,l_{MN/(JK)}^{(0)})$

Output: $\tilde{P} = (\tilde{p}_{j,j})$

```
 1: For $i := 1$ to $M / J$ do
 2:     $\tilde{\boldsymbol{x}}_i^{(0)} \leftarrow \boldsymbol{w}_{l_i}^{(0)}$
 3: End for
 4: For $j := 1$ to $N / K$ do
 5:     $\tilde{\boldsymbol{x}}_{jM/J+1}^{(0)} \leftarrow \boldsymbol{w}_{l_{jM/J+1}}^{(0)}$
 6:     For $i = 2$ to $M / J$ do
 7:         $\tilde{\boldsymbol{x}}_{jM/J+i}^{(0)} \leftarrow \boldsymbol{w}_{l_{jM/J+i}}^{(0)}$
 8:         Find $\boldsymbol{w}^* \in W^{(1)}$ such that
 9:             $d(\tilde{\boldsymbol{x}}_{jM/J+i-1}^{(1)},\boldsymbol{w}^*) = \min_{\boldsymbol{w}\in W^{(1)}} d(\tilde{\boldsymbol{x}}_{jM/J+i-1}^{(1)},\boldsymbol{w})$
10:         $\tilde{\boldsymbol{x}}_{jM/J+i-1}^{(1)} \leftarrow \frac{1}{2}(\tilde{\boldsymbol{x}}_{jM/J+i-1}^{(1)} + \boldsymbol{w}^*)$
11:     End for
12: End for
```

The difference between two versions is that the above version does not explicitly calculate the index sequence $L^{(1)}$. Although we assume $J_0 = J_1$ and $K_0 = K_1$, the algorithm can be generalized by modifying lines 5 and 7 to "if" statements. The statements from lines 7 to 10 can be processed in pipeline fashion.

3.2 MVQII Hardware Design

We implement the proposed MVQII algorithm on FPGA. We present three different circuit design: (I), (II) and (III).

Circuit Design (I)

The top view of design (I) is shown in **Fig. 4**. In the figure, a box with ">" means a synchronous circuit block and the other boxes are combinatorial circuit blocks. Note that a synchronous circuit block must be connected with a clock line and some control signal lines, which are omitted for simplicity in the figure. The design consists of a counter, four RAMs, combinatorial circuit units Average and Find Nearest. Four RAMs store $L^{(0)}$, $W^{(0)}$, $\tilde{P}$ and $W^{(1)}$, respectively. The Counter sequentially outputs memory addresses of $l_i^{(0)} \in L^{(0)}$, $\boldsymbol{w}_1^{(0)} \in W^{(1)}$, $\tilde{x}_i^{(0)}$ and $\tilde{x}_i^{(1)}$. The unit Find Nearest outputs the closest weight $\boldsymbol{w}^* \in W^{(0)}$ to $\tilde{\boldsymbol{x}}^{(1)}$, after $\boldsymbol{w}_1^{(1)},\cdots,\boldsymbol{w}_{\kappa_1}^{(1)}$ are inputted. That is, for a given $\tilde{\boldsymbol{x}}^{(1)}$, the calculation requires κ_1 clocks. This unit, which is a key module in our design, will be explained in detail later. And then, the average of $\boldsymbol{w}^*$ and $\tilde{\boldsymbol{x}}^{(1)}$ is calculated and written to RAM $\tilde{P}$. **Fig. 5** shows the inside of the unit Find Nearest for design (I). The unit DIST calculates $d = \| \tilde{\boldsymbol{x}}^{(1)} - \boldsymbol{w}^{(0)} \|^2$ by performing subtraction and squaring. The unit CMP performs the comparison between d and d_{min} . If $d < d_{min}$ then CMP sets $f = 1$. Otherwise it sets $f = 0$. The signal f is used as the enable signal of two registers. That is, if $f = 1$ then the two registers latch d_{min} and $\boldsymbol{w}^{(0)}$. The operating speed of the design (I) is dominated by the unit DIST. Because the unit performs subtraction and squaring and involves the longest combinatorial logic path among all logic blocks in the design.

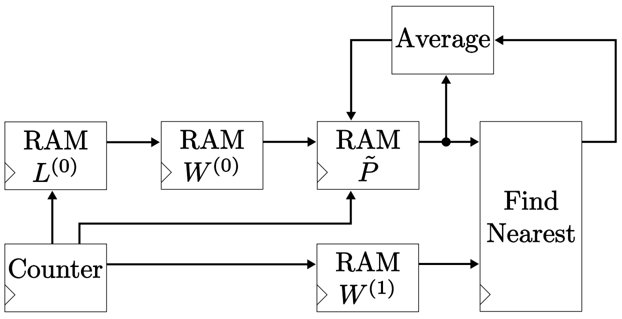


Fig. 4 The top view of design (I).

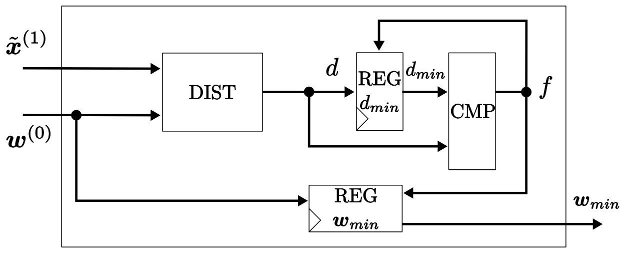


Fig. 5 The unit Find Nearest for design (I).

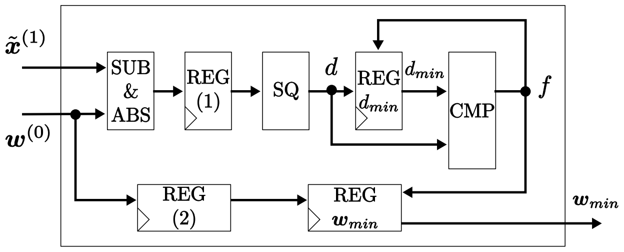


Fig. 6 The unit Find Nearest for design (II).

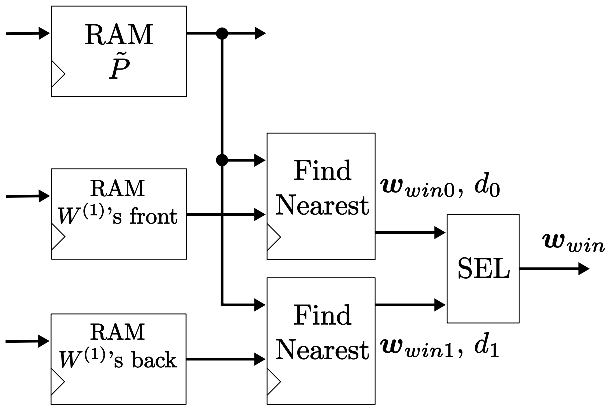


Fig. 7 The parallelized Find Nearest in design (III).

Circuit Design (II)

In order to improve the processing speed, we consider dividing the most complex logic, DIST, in the design (I) into two parts. The unit Find Nearest for the improved version, the design (II), is shown in **Fig. 6**. In this design, the unit DIST in design (I) is divided into two logic blocks SUB & ABS and SQ. The unit SUB & ABS calculates $\boldsymbol{d} = (|x_1 - w_1|,\cdots,|x_\kappa - w_\kappa|)$, where $\tilde{\boldsymbol{x}}^{(1)} = (x_1,\cdots,x_\kappa)$ and $\boldsymbol{w}^{(1)} = (w_1,\cdots,w_\kappa)$. The unit SQ calculates $\boldsymbol{d}\cdot\boldsymbol{d}$. Although this modification can improve the critical path delay in the circuit, two clocks are needed to calculate $d = \| \tilde{\boldsymbol{x}}^{(1)} - \boldsymbol{w}^{(0)} \|^2$. In order to hide the increased step, an additional register REG (2) is added. With REG (2), the unit Find Nearest operates in pipeline fashion. Its pipeline operation calculates m closest weights in $m\cdot\kappa_1 + 1$ clocks.

Circuit Design (III)

In order to reduce the number of clocks for calculating the closest weight, we consider the parallelization of the unit Find Nearest. The parallelized circuit is shown in **Fig. 7**. In the circuit, the weight set $W^{(1)}$ is divided into two sub-sets $W_0^{(1)} = \{\boldsymbol{w}_1^{(1)},\cdots,\boldsymbol{w}_{\kappa_1/2}^{(1)}\}$ and $W_1^{(1)} = \{\boldsymbol{w}_{\kappa_1/2+1}^{(1)},\cdots,\boldsymbol{w}_{\kappa_1}^{(1)}\}$, which are stored in RAM $W_0^{(1)}$ and RAM $W_1^{(1)}$, respectively. Two Find Nearest units calculate the closest weights

$\boldsymbol{w}_{win0} \in W_0^{(1)}$ and $\boldsymbol{w}_{win1} \in W_1^{(1)}$ to $\tilde{x}^{(1)}$ in parallel. This calculation completes in $\kappa_1/2+1$ clocks. Note that each Find Nearest unit is same as in **Fig. 6**. The unit SEL outputs $\boldsymbol{w}_{win} = \boldsymbol{w}_{win0}$ for $d_0 < d_1$ and otherwise $\boldsymbol{w}_{win} = \boldsymbol{w}_{win1}$. That is, $\boldsymbol{w}_{win}$ is the closest weight in $W^{(1)}$.

The number of clocks required for processing Q vectors are summarized in **Table 1**, where Q corresponds to $MN/(J_1K_1)$. The difference of required clocks between designs (I) and (II) is just one clock. Design (III) needs only about half the number of clocks required for design (I).

Table 1 The number of clocks required for processing Q vectors.

Design	Number of clocks
(I)	$Q(\kappa_1+1)+2$
(II)	$Q(\kappa_1+1)+3$
(III)	$Q(\kappa_1/2+1)+4$

4. Implementation Results

We implement the proposed designs on FPGA and evaluate their operating frequency, processing time and circuit size. The design environment is Xilinx ISE Design Suite 10.1, the targeted device is Spartan 2E XCS300E, and the design is coded in VHDL. Major specifications of the implemented circuits are summarized in **Table 2**.

Table 2 Specifications of the implemented MVQII.

Item	Spec.
Numeric coding	8 bits integer
Dimension of vectors	$J_2=2, K_2=1$ or $J_2=2, K_2=2$
Number of weight vectors	$\kappa_2=16$

The evaluation results on operating frequency are shown in **Table 3**. For any vector dimension, each design operates at almost the same frequency, and the design (II) achieves the highest operating frequency among all the designs. **Fig. 8** compares our three designs and a software implementation in terms of the processing time required for processing Q vectors. The processing time for each design is calculated by dividing the number of clocks in Table 1 by the operating frequency in **Table 3**. "Soft" indicates the processing time (CPU time) for software implementation, where the program compiled with the GNU Compiler GCC 4.2 runs on a GNU/Linux PC with 2.6.24 kernel, Intel Core2 Quad CPU Q6600 (2.40GHz) and 4GB of RAM. This result shows that the design (III) is the fastest among all the methods. The design (III) is about five times, twice and ten times faster than (I), (II) and Soft, respectively.

The evaluation results on circuit size are summarized in **Table 4**. The number of Slice Flip Flops (SFFs) corresponds to the circuit size of registers and counters, and the number of 4 input LUTs (4LUTs) corresponds to the circuit size of combinatorial logic circuits. For each design type, the extension from $J_2=2, K_2=1$ to $J_2=2, K_2=2$ increases SSFs and LUTs by 1.67~1.95 times, because the number of bits for representing the weight vectors is doubled. For each dimension of vectors, the designs (II) and (III) use approximately twice and 4 times as many SFFs as the as the design (I), respectively. On the other hand, the design (II) uses fewer 4LUTs than the design (I), and the design (III) uses 1.69~1.78 times as many 4LUTs as the design (I).

Finally, we mention the difference between the proposed designs and a naive implementation of algorithm MVQII described in subsection 2.3. The algorithm MVQII initially restore a whole image $\tilde{P}^{(0)} = (\tilde{p}_{i,j}^{(0)})$ from $W^{(0)}$ and $L^{(0)}$, and then calculates the index sequence for $W^{(1)}$. Therefore, the naive implementation requires about twice as much clocks as the designs (I) and (II).

Table 3 Maximum operating frequency of implemented circuits.

Design	$J_2=2, K_2=1$	$J_2=2, K_2=2$
(I)	28.91 MHz	25.38 MHz
(II)	75.22 MHz	72.86 MHz
(III)	71.52 MHz	71.38 MHz

Table 4 Numbers of slice Flip Flops and 4 input LUTs for implemented circuits.

Dim.of vectors	Design	# of Slice Flip Flops	# of 4 input LUTs
$J_2=2$ $K_2=1$	(I)	72	204
	(II)	143	172
	(III)	256	346
$J_2=2$ $K_2=2$	(I)	120	369
	(II)	273	326
	(III)	500	656

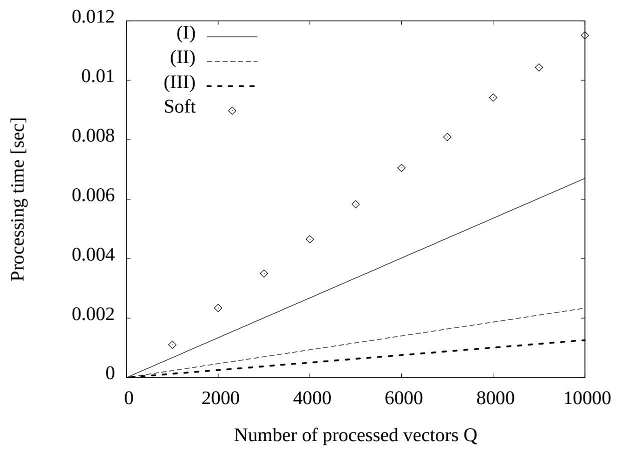


Fig. 8 Processing time for $J_2 = 2, K_2 = 2$ **.**

5. Conclusions

In this paper, we have presented three types of digital circuit designs for Multiple-VQ with Index Inference (MVQII) decoders. The first one, which is a naive implementation, is of the smallest circuit size, but its operating speed is slow due to its long combinatorial path. The second and third ones improve the processing time by introducing “pipeline” and “parallelization”, respectively. The latter two versions achieve the good improvement in processing speed at a reasonable expense of circuit size. In particular, the design (II) and (III) are about 2.87 times and 5.32 times faster than the design (I), respectively, while the designs (II) and (III) require about twice and four times larger circuits than the design (I), respectively. Further, the processing speed of the circuits (I), (II) and (III) is about 1.72 ~ 9.15 times faster than a software implementation on a modern standard computer.

Another problem related to MVQII is the training process of codebooks, which is performed in the compression phase. The computational complexity for MVQII is twice larger than for SVQ. One of our future works is to develop effective digital circuit design for the training process.

Acknowledgments

This research is partly supported by the Grant-in-Aid for Scientific Research (C) (No. 19500195) of the Ministry of Education, Culture, Sports, Science and Technology of Japan.

References

[1] R.M. Gray, “Vector Quantization," *IEEE ASSP Magazine*, pp.4-29, 1984.

[2] A. Gersho and R.M. Gray, Vector Quantization and Signal Compression, Kluwer Academic Publishers, 1992.

[3] N. Shigei, H. Miyajima and M. Maeda, “A Multiple Vector Quantization Approach to Image Compression,” *Proc. of International Conference on Natural Computation, Lecture Notes in Computer Science*, Vol.3611, pp.361-370, 2005.

[4] T. Kohonen, Self-Organizing Maps, Springer-Verlag, Berlin Heidelberg, New York, 1997.

[5] N. Shigei, H. Miyajima, M. Maeda and L. Ma, “Compression Rate Improvement of Multiple Vector Quantization Based Image Compression,” *Proc. of International Conference on Intelligent Technologies*, pp.133-140, 2005.

[6] N. Shigei, H. Miyajima and M. Maeda, “Learning Algorithm for Multiple Vector Quantization Based on Image Compression,” *Proc. of International Symposium on Nonlinear Theory and its Applications*, pp.836-839, 2006.

[7] J. E. Fowler, “QccPack: An Open-Source software Library for Quantization, Compression, and Coding,” in *Applications of Digital Image Processing XXII (Proc. SPIE 4115)*, A. G. Tescher, ed., pp.294-301, 2000.

[8] C. Amerijckx, J.-D. Legat, M. Verleysen, “Image Compression Using Self-Organizing Maps," *Systems Analysis Modelling Simulation*, 43, pp.1529-1543, 2003

[9] F.A. Elmisery, A.H. Khaleil, A.E. Salama, F. El-Geldawi, "An FPGA Based VQ for Speaker Identification," *The 17th International Conference on Microelectronics*, pp.130-132, 2005.

[10] J.J. Raygoza-Panduro, S. Orega-Cisneros, E. Boemo, "FPGA implementation of a synchronous and self-timed neuroprocessor," *Proc. of the 2005 International Conference on Reconfigurable Computing and FPGAs*, 2005.

[11] I. Manolakos, E. Logaras, "High Throughput Systolic SOM IP Core for FPGAs," *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 2, pp.61-64, 2007.

Noritaka Shigei received the B.E., M.E., D.E. degrees from Kagoshima University, Japan, espectively, in 1992, 1994, and 1997. He is an Assistant Professor in the Dept. of Electrical & Electronic Eng. at Kagoshima University. His research interests include neural networks, digital circuit design, mobile agent system, and energy efficient communication in sensor networks. He is a member of the IEEE, the ACM and the IEICE.

Hiromi Miyajima received the B.E. degree in electrical engineering from Yamanashi University, Japan, in 1974, and the M.E. and D.E. degrees in electrical and communication engineering from Tohoku University, in 1976 and 1979, respectively. He is currently a Professor in the Department of Electrical & Electronics Engineering at Kagoshima University. His current research interests include fuzzy modeling, neural networks, quantum computing, and parallel computing.

Shingo Hashiguchi received the B.E. and M.S. degrees in electrical and electronics engineering from Kagoshima University in 2006 and 2008, respectively. His research interests have been digital circuit design and digital wireless communication system. Currently, he is working at Fujitsu Microelectronics Limited, Japan.

Michiharu Maeda received the B.E. and M.S. degrees in theoretical physics in 1990 and 1992, respectively, and the Ph.D. degree in information and computer science in 1997, from Kagoshima University, Japan. He is currently an Associate Professor in the Department of Computer Science & Engineering at Fukuoka Institute of Technology. His research interests include computational intelligence, mathematical & physical computation, and signal processing.

Lixin Ma received the D.E. degree in System Information Engineering from Kagoshima University, Japan in 1999. He is currently a Professor at College of Computer and Electrical Engineering in University of Shanghai for Science and Technology. His research interests include neural networks, fuzzy theory and its industrial applications.