가변 적응형 사전을 이용한 텍스트 압축방식의 병렬 처리를 위한 VLSI 구조

이 용 두[†] · 김 희 철[†] · 김 중 규[††]

요 약

컴퓨터 통신망의 발달로 다량의 텍스트(Text) 또는 영상(Image) 정보의 전달이 이루어 지고 있다. 텍스트 압축과정에서 주어진 어휘를 이전에 나타난 같은 어휘를 가리키는 포인터로 대치시키는 원리에 준하여 설계된 LZ77 알고리즘은 가변적응형 (adaptive) 사전을 이용한 텍스트 압축 방식으로 실제적으로 가장 많이 사용되는 알고리즘이다. 본 논문은 LZ77의 병렬 처리를 위해 LZ77의 Parallelism에 대한 분석 결과를 보여주며, 그 분석 결과를 적용한 병렬 LZ77 알고리즘의 설계, 그리고 그러한 병렬 LZ77 알고리즘을 처리하도록 고안된 VLSI 시스템 구조에 관한 연구 내용을 기술한다. 이전의 유사한 연구 내용과 비교하여, 본 논문에서 제안된 VLSI 시스템은 사전 윈도우(dictionary window)의 크기에 제한이 없으므로 확장성이 뛰어난 장점을 갖으며, 입력 텍스트의 길이가 (N)일때, 사전 윈도우의 크기에 관계없이 그 처리속도가 O(N)이며 VLSI 구현시 다른 유사한 시스템보다 향상된 집적도를 갖는다.

A Novel VLSI Architecture for Parallel Adaptive Dictionary-Based Text Compression

Yong-Doo Lee[†] · Hiecheol Kim[†] · Jung-Gyu Kim[††]

ABSTRACT

Among a number of approaches to text compression, adaptive dictionary schemes based on a sliding window have been very frequently used due to their high performance. The LZ77 algorithm is the most efficient algorithm which implements such adaptive schemes for the practical use of text compression. This paper presents a VLSI architecture designed for processing the LZ77 algorithm in parallel. Compared with the other VLSI architectures developed so far, the proposed architecure provides the more viable solution to high performance with regard to its throughput, efficient implementation of the VLSI systolic arrays, and hardware scalability. Indeed, without being affected by the size of the sliding window, our system has the complexity of O(N) for both the compression and decompression and also requires small wafer area, where N is the size of the input text.

※이 논문은 1995학년도 대구대학교 학술연구비 지원에 의한 논문임
† 정 회 원:대구대학교 정보통신공학부
†† 비 회 원:대구대학교 정보통신공학부
논문접수:1996년 12월 10일, 심사완료:1997년 5월 2일

1. 서 론

컴퓨터 통신망의 발달로 막대한 양의 텍스트(Text) 또는 영상(Image) 정보의 전달이 이루어지고 있다. 하지만, 통신회선의 처리용량은 여전히 제한되어 있으므로, 가급적이면 정보의 압축을 통한 전달이 바람직하며, 서적등 큰 용량의 텍스트를 실시간(Real-time)에 압축할 수 있는 고성능 압축 방식을 더욱 필요로 하게 된다.

텍스트 압축 알고리즘에 대한 많은 연구가 있었다 [1, 4, 6, 10, 11, 12, 13, 17, 18, 20]. 그 중에 압축될 어휘를 이전에 나타난 어휘를 가리키는 포인터로 대치시키는 원리를 이용하는 가변 적응형 텍스트 압축 알고리즘에 대한 많은 연구가 있었다[2, 14, 15, 20, 21]. 그러한 알고리즘은 가변적응식 사전을 사용함으로써, 고정된 사전식 압축 방식과 비교하여 특정한 텍스트의 내용에 제한 받지 않고 양질의 압축 코드를 생성하는 장점을 갖고 있다[21].

그러나 가변 적응형 알고리즘에서는 압축하고자 하는 텍스트의 길이가 N이고 최대 압축 어구의 크기가 L 일 때 O(N2L)의 글자 비교를 해야 하므로 소프트웨어만으로 구현할 경우 그 압축 처리 속도가 매우 느리게 된다[4, 6, 9, 15]. 보다 향상된 처리 속도를 얻기 위해 병렬처리 하드웨어 방식이 시도되었다[3, 16, 23, 24]. 특히, 병렬 처리를 위해서 고안된 모델의 일종인 VLSI를 이용한 시스톨릭 배열(Systolic arrays)은 균일한 상호 접속(Interconnection)을 통한 많은 프로세서들의 집적을 가능케 하므로 텍스트 압축 알고리즘의 병렬처리 시스템 구현에 매우 적합하다[8]. 실제로, 이러한 VLSI 시스톨릭 배열을 이용한 방식을 이용하여 빠른 속도의 텍스트 압축 처리를 위한 연구가 시도되었다[3, 16, 23]. 그러나 그러한 시스템들은 확장성, 처리속도, 그리고 웨이퍼 집적도 등의 측면에서 많은 개선이 필요하다[16].

본 논문은 이전의 다른 유사한 연구 내용과 비교하여, 사전 윈도우(Window)의 크기에 제한이 없고, 확장성이 뛰어나며, 입력 텍스트의 길이가 N일 때 압축 처리 구간의 크기에 상관없이 처리속도 O(N) 그리고, VLSI 구현시 웨이퍼 면적은 O(N)의 복잡도를 갖는 이상적인 성능의 VLSI 시스템 제안한다.

본 논문의 2절에서는 텍스트 압축과 관련된 용어 및 기존의 방식을 포함한 이론적 배경을 소개한다. 3절에서는 텍스트 압축의 병렬처리방식과 성능과의 상호 관계를 분석하기 위하여 수행한 LZ77 알고리즘의 Parallelism 분석 결과를 서술한다. 4절에서는 제안하는 시스템의 구조와 그 동작 원리가 자세히 설명되며, 5절에서는 다른 유사한 연구와의 성능비교를 보여준다. 마지막으로 6절에서는 본 논문의 결론이 서술된다.

2. 이론적 배경

본 절에서는 텍스트 압축에 관련한 이론적 배경을 소개한다.

2.1 텍스트 압축

텍스트 압축 (Text compression)이란 한개 이상의 글자(Character)로 이루진 입력 텍스트에 대하여, 보다 작은 크기의 압축 코우드(Code)를 산출해 내는 동작을 말하며, 출력된 압축 코우드는 반드시 원래의 입력 텍스트로 원상 회복(Decompression)될 수 있는 형태를 갖는다. 텍스트 압축 방식의 성능은 압축률, 즉 출력된 압축 코우드의 크기(Size)에 대한 입력 텍스트의 크기의 비율로 정의된다.

텍스트 압축 방식은 크게 통계(Statistics) 방식과 사전(Dictionary) 방식으로 구분된다[17]. 통계 방식은 입력 텍스트를 구성하는 각 글자가 입력 텍스트에 나타나는 확률을 계산한 다음 높은 확률을 갖는 글자일수록 작은 크기의 코우드를 부여함으로 생성되는 압축 코우드의 크기를 줄이는 원리를 갖는다. 한편, 사전 방식은 일련의 어휘(Phrase)들로 구성된 사전(Dictionary)을 참조하여 이루어진다. 입력 텍스트에 나타나는 어휘가 사전에 등록되어 있는 경우에는 그것을 그 사전에 등록된 어휘를 가리키는 포인터(Pointer)로 대치시킴으로써 출력 텍스트의 크기를 줄이게 된다.

통계 방식과 사전 방식은 서로 상이한 원리와 특성을 갖는다[17]. 통계 방식에서는 압축을 위한 코우딩(Coding)이 글자 단위로, 반면에 사전 방식은 어휘 단위로 이루어진다. 통계 방식은 각 글자에 대하여 계산된 확률 통계에 바탕을 두고 있으므로, 성능 면에 있어서 사전 방식보다 우수하다[17, 18]. 일반적으로 텍스트 압축이 적용되는 분야(context)는 매우 다양하

다. 그러나 확률 통계를 얻어내는 과정이 복잡하며 한 분야에 계산된 확률 통계가 다른 분야의 입력 텍스트에는 알맞게 적용이 되지 않는 경우가 많아 통계방식은 실용성 면에서 심각한 제한이 따른다[8].

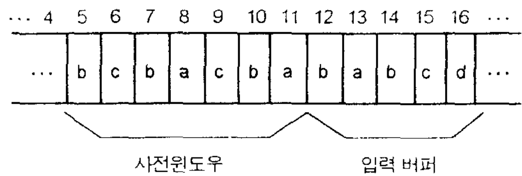
2.2 사전 방식 텍스트 압축 기법

사전방식은 통계 방식보다 이론적으로 성취할 수 있는 성능의 최고값은 낮으나, 실용면에 있어서 비교적 우수한 성능을 얻을 수 있는 압축 방법으로 그 구현이 매우 용이한 장점을 갖는다[21]. 사전방식은 다음과 같이 표현되는 사전(D)으로 정의된다: $D=(P, C)$. 여기서 (P)는 어휘(Phrase)들의 집합(Set)이며 (C)는 그 어휘들을 주어진 코드값에 일대일로 매핑(Mapping)시키는 함수이다. 예를 들어, P={a, ba, bc, ac} 그리고 $C(a)=0$, $C(ba)=00$, $C(bc)=01$, $C(ac)=10$, $C(abc)=11$인 사전이 있다고 하자. 이때, 입력 텍스트 "abc"는 "001" 또는 "11"로 압축 될 수 있다. 쉽게 추론해 볼 수 있듯이, 사전의 내용은 사전 방식 압축의 성능에 가장 큰 영향을 미친다. 그러므로 사전의 내용 선택은 사전 방식의 구현시 가장 중요한 요소중의 하나이다. 사전을 결정하는 방식에 따라 사전을 사용하는 텍스트 압축 방식은 크게 고정식(Static), 준적응식(Semi-adaptive), 그리고 적응식(Adaptive)으로 분류된다[18, 21]. 고정식 사전 방식은 기존의 코드체계에서 텍스트로 사용하지 않는 코드에 자주 사용하는 어휘들을 할당하여 입력 텍스트의 압축시 사용하는 방식이다. 준적응식 사전방식은 고정식 사전 방식과 그 기본 원리에 있어서는 동일하지만 주어진 텍스트에 적합하게 미리 정한 특별한 사전을 사용한다. 한편, 적응식 사전 방식은 주어진 입력 텍스트의 압축 과정에서 빈번히 사용되는 어휘들을 자동으로 추출하여 그 사전으로 이용한다.

고정식이나 준적응식 사전 방식은 사전의 크기가 작다. 그 결과로 그 압축 성능이 매우 낮기 때문에 실용적이지 못하다. 반면 적응식 사전 방식은 비교적 우수한 성능을 얻을 수 있다. 적응식 사전방식에 대해서는 많은 연구가 있었는데 그중 구현성과 성능 면에서 뛰어난 LZ77 알고리즘이 가장 많이 사용되어 왔다[3, 16, 21, 22].

2.3 LZ77 알고리즘

LZ77 알고리즘의 원리는 다음과 같다. 압축 과정에서 (N)개의 글자로 이루어진 입력 텍스트는 압축된 부분과 압축될 부분으로 나뉘어 지며, 압축된 부분의 마지막 (W)개의 글자는 사진 윈도우(Window)로 정의되고 압축될 처음 (L)개의 글자는 입력 버퍼(Buffer)로 정의된다[21]. 예를 들면, (그림 1)은 (W)와 (L)의 값을 각각 7과 4로 지정 사용하는 LZ77 알고리즘을 (N)개의 글자 열을 가진 입력 텍스트에 대해 적용할 경우, 처음 11개의 글자까지 압축을 끝냈을 시점의 사전 윈도우 및 입력 버퍼의 구성을 보여준다.



(그림 1) LZ77 알고리즘의 텍스트 구성
(Fig. 1) The LZ77 sliding window

LZ77의 알고리즘은 압축(Encoding)과 원상 회복(Decoding)의 두 부분으로 구성된다. (그림 2)는 압축과 회복 알고리즘을 보여주는데, (N), (W), 그리고 (L)은 각각 입력 택스트, 사전 윈도우, 그리고 입력 버퍼의 크기를 나타내며, Text와 LZ77_Code는 각각 입력 텍스트와 LZ77 압축 코우드를 저장하는 데이터 어레이(Array)를 나타낸다. 또한 window(A[i], A[j])와 buffer(A[i], A[j])는 어레이 (A)의 (i)번째부터 (j)번째까지의 글자들로 구성된 글자 열들을 각각 window와 buffer라는 이름으로 나타내기 위해 사용한 표현이다.

(그림 2)의 왼쪽에 보여지는 LZ77 압축 알고리즘은 기본적으로 입력 버퍼의 첫글자부터 시작하는 글자열(Character string)과 매칭(Matching)이 되는 사전 윈도우의 가장 긴 글자열을 찾아내어 그 시작 위치와 길이, 그리고 다음 입력 버퍼의 첫 글자로 이루어진 LZ77 코우드(Code)라고 부르는 압축 코우드 (즉, 알고리즘 상에 $\langle j_m, l_m, Text[i_m]\rangle$)를 입력 버퍼의 매칭이 된 글자열 대신 출력하게 된다. 그리고 매칭된 글자열의 크기만큼 사전 윈도우와 입력 버퍼를 오른쪽으로 이동시킨후 위의 과정을 다시 반복한다. 여기서, LZ 코우드에 다음 입력 버퍼의 첫 글자를 포함시키

는 이유는 사전 윈도우와 입력 버퍼간에 매칭이 일어나지 않는 경우를 (즉, 매칭된 글자열의 길이=0) 고려한 때문이다. 압축 코우드인 LZ77 코우드를 원래 텍스트로 회복시키는 알고리즘은 (그림 2)의 오른편에 보여진다. 각 LZ77 코우드는 해당하는 글자열을 지정하는 시작위치와 길이로 구성되어 있으므로, 회복 알고리즘은 그 코우드를 해당하는 글자열을 사전 윈도우에서 찾아 바꿔 주는 원리에 준해 구성된다.

3. LZ77의 Parallelism 분석

LZ77의 병렬 처리를 위해서는 그 알고리즘에 내재된 Parallelism에 대한 분명한 이해가 필요하다. 본 절에서는 LZ77 알고리즘의 Parallelism의 분석 결과를 서술하며 그 결과를 기존의 방식에 적용하여 설명한다.

3.1 LZ77의 Parallelism

LZ77의 인코딩(Encoding) 알고리즘간에 입력 버퍼의 내용과 사전 윈도우의 내용중 매칭이 되는 어휘들중 길이가 제일 큰 것을 찾아내는 부분(그림 2의 line 7-9)이 병렬 처리를 가능케 하는 Parallelism을 내포하고 있다. 보다 정확한 설명을 위하여 사전 윈도우 크기 (W)가 (n)이며, 입력 버퍼의 크기 (L)이 (n)인 경우를 생각해 보자. 여기서 사전 윈도우의 내용과 입력 버퍼의 내용을 각각 ⟨$w_1, w_2, w_3, \ldots, w_n$⟩와 ⟨$x_1, x_2, x_3, \ldots, x_n$⟩로 정의하면, (그림 2)의 line 7-9를 수행하기 위해서는 모든 ⟨x_i, w_j⟩ ($1 \le i, j \le n, i \le j$)에 대하여 x_i와 w_j간에 글자 비교가 필요하다. 각 글자 비교는 서로 의존성 (Dependency)를 갖지 않으므로 병렬로 이루어질 수 있으며, 본 논문의 나머지 부분에서는 그러한 병렬성을 매칭(Matching) Parallelism이라고 부른다.

매칭 Parallelism은 (n)개의 Processor가 사전 윈도우에 할당되어 있을 때, 다음의 두 가지 경우로 병렬 처리가 가능하다.

1. ⟨글자열 단위 병렬 처리⟩: 입력 버퍼의 글자열과 해당하는 사전윈도우의 글자열을 병렬로 글자비교를 수행한다.
2. ⟨글자 단위 병렬 처리⟩: 입력 버퍼의 각글자를 사전윈도우의 모든 글자에 병렬로 글자비교를 수행한다.

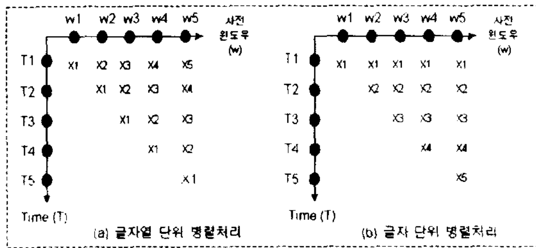
(그림 3)은 (n=5)의 경우에 ⟨글자열 단위⟩와 ⟨글자

```
Procedure LZ77-Encode (Text)
1: begin
2:   i ← W+1:
3:   while i ≤ N-L+1 do
4:     begin
5:       LM ← φ ;
6:       for j ← i-W until i-1 do
           begin
7:           Find longest match between
                 window(Text[j], Text[i-1]) and
                 buffer(Text[i], Text[i+L-1]):
8:           l ← the size of longest match:
9:             Add two tuples<j,l,Text[i+l-1]> to LM;
           end
10:      Find <jm,lm,Text[im]> which is the longest Match in
            LM;
11:      i ← i + jm:
12:      LZ77_Code[k].offset ← jm;
13:      LZ77_Code[k].length ← lm;
14:      LZ77_Code[k].char ← Text[i]:
15:      k ← k + 1;
16:    end
17:  retruln LZ77_Code:
18: end
```

```
Procedure Lz77-Decode (LZ77_Code)
 1: begin
 2:   i ← 1, j ← 1;
 4:   while C is not empty do
 5:     begin
          /* get starting location and length
             of a LZ77 code. */
 6:       offset ← LZ77_Code[i].offset;
 7:       count ← LZ77_Code[i].count;
 8:       I ← i+l
 9:       while (count>0) do
10:         begin
11:           Text[j] ←Text[j-offset];
12:           j ← j+l
13:         end
14:     i ← i+l
15:     end
16:     return Text:
17: end
```

(그림 2) LZ77 코딩과 회복 알고리즘
(Fig. 2) LZ77 encoding and decoding algorithm

단위〉 병렬 처리의 각각의 경우에 글자 비교가 어떻게 일어나는 지를 보여주고 있다. 여기서 가로축은 사전 윈도우의 내용을 세로축은 시간을 나타내 주며, (w_i, T_j)의 좌표값 x_k는 시간 T_j에 w_i와 x_k간에 글자 비교가 일어남을 의미한다. 그림에서 볼 수 있듯이, LZ77 알고리즘에서 주어진 글자 x_i는 〈x_{i-n} .. x_{i-n-1}〉의 (n)개의 글자들과 글자 비교가 수행되는데 〈글자열 단위 병렬처리〉 방식에서는 글자 비교에 걸리는 단위 시간이 T라고 정의할 때 xi에대한 글자 비교가 직렬로 n*T에 걸쳐 이루어지며, 〈글자 단위 병렬처리〉에서는 단지 T만 걸리는 것을 볼 수 있다. 그러한 차이는 시스템의 성능과 복잡도에 결정적인 요인으로 작용하며, 그 자세한 내용은 본 고의 나머지 부분에서 설명된다.



(그림 3) 매칭 Parallelism의 병렬 처리
(Fig. 3) Two cases of the exploitation of matching parallelism

3.2 분석결과의 기존 병렬처리 방식에의 적용

LZ77의 병렬처리를 위한 VLSI 구현 시스템은 인코더(Encoder)와 디코더(Decoder)로 구성된다. 일반적으로 인코더와 디코더는 사전 윈도우 (Window)와 입력 버퍼의 글자 크기에 따라 각 글자 해당하는 프로세서(Processor)들이 선형으로 연결된 어레이 (Array) 구조를 갖는다[3]. 이러한 구성은 입력 버퍼와 사전 윈도우에 대한 매칭(Matching)을 자연스럽게 지원할 수 장점을 갖는다. 그러나, 인코딩의 알고리즘에서 입력 버퍼의 내용과 사전 윈도우의 내용간에 매칭이 되는 어휘들중 길이가 제일 큰 것을 찾아내는 부분(그림 2의 line 7-9)의 구현을 위해서는 부가적인 회로가 필요하며, 앞절에서 분석한 매칭 Parallelism의 처리 방식이 그 구현의 성능과 하드웨어의 복잡도에 결정적인 영향을 미친다.

(그림 4)는 M. Gonzales와 J. Storer가 제안한 시스템, 즉 Match Tree Architecture(MTA)의 구조와 그 작동원리를 보여주고 있다. MTA는 사전 윈도우에 해당하는 선형 어레이 프로세서들은 단말 노드(Terminal node)에, 단말 노드를 제외한 비단말(Nonterminal) 노드에는 새로운 종류의 프로세서를 사용하는 바이너리 트리(Binary tree)의 형태로 구현된다. 바이너리 트리를 위해 새로이 도입된 프로세서들은 매 글자의 처리시 가장 긴 매칭 글자열을 찾아내는 역할을 수행한다. 이때 선형 어레이의 프로세서의 수가 (N)일 때 (N-1)개의 비단말용 프로세서가 필요하다.

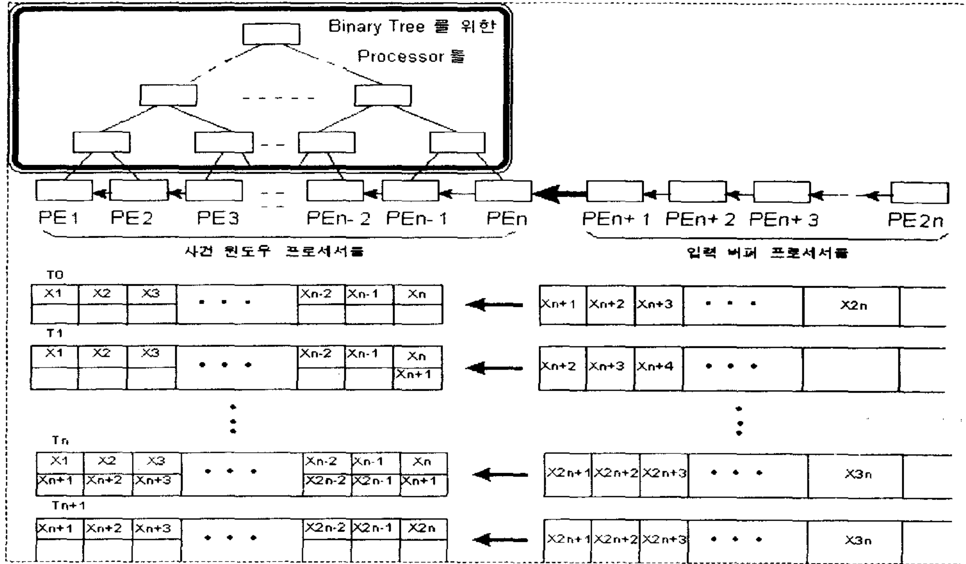
좀더 상세한 설명을 위한 간단한 예가 (그림 3)의 아래편에 보여지며, 처음 (n)개의 글자열 (즉, x_1 .. x_n)이 사전 윈도우에 저장이 되어 있을 때, 다음 (n)개의 글자열 (즉, x_{n+1} .. x_{2n})의 처리 과정이 아래 〈표 1〉에 요약된다.

<표 1> MTA의 병렬처리 동작요약
<Table 1> Operations of the MTA

시간	동작 요약
T_0	사전윈도우와 입력버퍼에 각각 <x_1 .. x_n>과 <x_{n+1} .. x_{2n}>가 있음.
T_1	x_{n+1}을 사전 윈도우의 PE_n에 Shift in 시키고, x_n과 비교함.
T_n	x_{2n} 을 사전 윈도우의 PE_n에 shift-in 시키고 <x_1 .. x_n>과 <x_{n+1} .. x_{2n}>을 병렬로 비교함.
T_{n+1}	<x_{n+1} .. x_{2n}>를 사전윈도우로 저장하여 다음 <x_{2n+1} .. x_{3n}>의 처리를 준비함.

〈표 1〉에서 나타나 있듯이, 사전윈도우의 크기가 (W)일 경우 각 글자의 처리시 필요한 동작과 복잡도는 다음과 같이 요약된다.

- 사전윈도우와 입력 버퍼 processors의 입력 글자열의 Shift-left : $O(1)$
- W개의 〈x_i, x_{n+i}〉 $(1 \le i \le N)$의 병렬 글자 비교 : $O(1)$
- Binary tree processors에 의한 최대 매칭 글자 열의 처리 : $O(log(W))$
- (n)개의 입력 글자 처리후, 사전 윈도우의 갱신 : $O(1)$



(그림 4) M. Gonzales와 J. Storer의 시스템
(Fig. 4) An overview of M. Gonzales and J. Storer's system

위의 내용들을 종합해 보면, M. Gonzales와 J. Storer가 제안한 시스템의 매칭 Parallelism의 처리 방식은 앞절에서 정의한 〈글자열 병렬 처리 방식〉에 해당한다. 그러나, 이러한 방식은 Binary tree의 비단말용 프로세서의 필요로 인해 하드웨어의 복잡도가 커지며, 사전 윈도우의 크기가 W일 때 각 글자를 처리하는데 $O(logW)$의 시간이 걸리게 되는 단점이 있다.

4. 제안하는 방식

본 절에서는 LZ77 알고리즘의 병렬 처리를 위해 본 논문에서 제안하는 병렬 알고리즘과 VLSI 시스템의 설계 내용을 기술한다.

4.1 제안하는 시스템의 구현 원리

본 논문에서 제안하는 시스템은 하드웨어의 간결성과 빠른 압축 처리 속도를 그 설계 목표로 한다. MTA와는 달리, 제안하는 시스템은 〈글자 단위 병렬 처리〉를 채택하고 또한 매글자 처리후 글자와 사전 윈도우의 글자열이 왼편으로 이동(Sliding)되는 방식을 채택하여 최대 매칭 글자열의 찾기를 효과적으로 지원한다. 제안하는 시스템의 구조하에서는 사전 윈도우의 입력 글자열과의 매칭이 되는 글자열이 사전 윈도우에 존재할 때, 그 글자열에 대한 매칭은

(원리 1) 항상 같은 프로세서에서
(원리 2) 연속적으로

일어난다. 좀더 자세히 설명을 하면 다음과 같다. 입력글자열과 사전윈도우의 글자열중 매칭이 되는 부분이 각각 〈x_1 .. x_m〉과 〈y_1 .. y_m〉이라고 하자. 만약 x_1과 y_1의 매칭이 프로세서 PE_i에서 일어났다고 가정하면, x_1의 처리후 사전 윈도우가 왼편으로 쉬프트(Shift)되므로 y_2가 PE_i에 놓여지게 된다. 따라서 x_2와 y_2의 매칭도 프로세서 PEi에서 일어나게 되며, 결국 같은 원리로, 그 외의 글자들의 매칭도 프로세서 PE_i에서 일어나게 된다. 위의 (원리 1)과 (원리 2)를 이용하면, 각 프로세서에서 일어나는 매칭의 갯수를 계산함으로 그 매칭(Matching) 글자열의 글자 수를 찾아낼 수 있다.

여기서 주목해야 할 점은 M. Gonzales와 J. Storer가 제안한 시스템에서와 마찬가지로 사전 윈도우의 프로세서의 개수가 N일 때, 최대 N개까지의 매칭글자열이 동시에 시스템에 존재할 수 있다는 점이다.

그러므로, 그들 중 최대 매칭글자열의 결정 방법의 효과적인 구현이 필요하게 된다. 제안하는 시스템에서는 그 구현을 위해 아래의 원리를 이용한다.

(원리 3): 어느 입력 글자열의 처리 과정 중, 어느 한 글자의 처리 시, 바로 그 이전 글자에 대하여 최대 매칭 글자열을 발생시킨 프로세서들만 이 최대 글자열을 발생시킬 가능성이 있다.

(원리 3)을 바탕으로한 최대 매칭 글자열의 구현은, 프로세서마다 최대 매칭 글자열을 처리하고 있는지 여부를 나타내는 상태(State) 플래그를 통하여 다음과 같이 이루어진다. 어느 시간 T에 주어진 프로세서의 상태플래그값을 S(T)라고 나타낼 때, 그 값은 다음과 같이 정의된다. 즉, 최대 매칭 스트링을 처리하고 있을 때, S(T)는 '1'의 값을 갖고, 그렇지 않으면 '0'을 갖는다. 매글자 처리후 글자 비교의 결과 (M)은 글자 매칭이 이루어지면 '1', 그렇지 않으면 '0'으로 정의된다. 그러므로 어느 시점 T_i의 상태값 $S(T_i)$은 바로 이전의 상태값 $S(T_{i-1})$와 M을 논리 앤드(AND)시킨 값 (즉, $S(T_i) = S(T_{i-1}) \wedge M$)으로 결정된다. 여기서, $S(T_{i-1})$을 사용함으로 이전에 매칭이 실패한 프로세서는 비록 글자 매칭이 이루어 졌다고 해도 최대 매칭 글자열을 처리하는 프로세서들에서 배제되도록 한다. 아래에 간단한 예를 보면, PE_i에 6개의 글자열 'abcbcd'의 매칭이 일어나며, 각각의 시간에 그 상태값 S(t)가 '1'로 된다. 하지만 PE_j에서는 T=3 일 때, 글자 매칭이 일어나지 않아 상태값이 '0'으로 되기 때문에, 그 이후 글자열'bcd'의 매칭이 일어난다 할지라도 상태값이 계속 '0'으로 남아 있는다. 그러므로 PE_j는 T=3 이후에는 최대 매칭 글자열의 처리에서 배제된다.

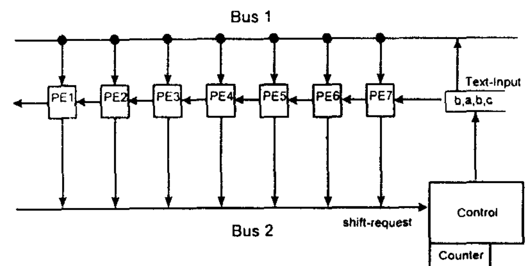
```
Time 1 2 3 4 5 6
----------------------------
PEi: 매칭 a b c b c d
     S(T) 1 1 1 1 1 1
PEj: 매칭 a b b c d
     S(T) 1 1 0 0 0 0
```

위에 설명한 (원리 3)을 이용한 최대 매칭 찾기의 하드웨어 구현은 아래와 같다. 먼저 각 프로세서는 글자 비교시 매칭이 일어났는가의 여부를 나타내 주는 신호를 발생하게 하며, 중앙의 컨트롤러에서는 그 신호들의 논리 OR 값을 산출한다. 컨트롤러는 그 값을 검사하여 입력 버퍼의 글자열에 대해 진행되는 사전 윈도우의 글자 매칭이 연속적으로 이루어지는 횟수를 계수기(Counter)를 이용하여 계산한다. 연속적인 글자 매칭이 이루어지고 난 후 모든 프로세서가 매칭에 실패하였으면 계수기의 값이 최대 매칭된 글자열의 크기가 된다. 구현되는 시스템의 구체적인 구조와 동작은 본 고의 나머지 부분에서 자세히 설명된다.

4.2 VLSI 시스템 구조

앞에서 설명한 대로, 가장 대표적인 LZ77 알고리즘의 병렬처리 연구인 M. Gonzales와 J. Storer가 제안한 시스템은 VLSI시스톨릭 어레이 구현을 위해 사전윈도우와 입력 버퍼를 구성하는 선형 프로세서 어레이(array)와 최대 매칭 글자열 찾기를 위한 트리 구조의 프로세서들로 구성된다[4, 24]. 본 연구에서는 제안되는 시스템은 사전윈도우를 구성하는 선형 프로세서 어레이와 Broadcasting 버스(Bus)로 구성되는 VLSI시스톨릭 어레이로 구현된다.

(그림 5)는 제안되는 VLSI 시스템의 구조를 보여준다. 그 시스템은 컨트롤러, Broadcasting Bus, 그리고 사전 윈도우의 크기 (W)와 같은 갯수의 동일한 프로세서들이 선형으로 배열된 형태를 갖는다. 각 프로세서들은 입력 글자를 저장되어 있는 사전 윈도우의 글자와 비교하여 그 결과를 출력하는 것을 그 주요 기능으로 갖는다. 모든 프로세서에는 같은 글자가 입력될 수 있도록 단일 라인의 입력 버스(Bus)로 연결되며, 그 출력 버스를 통하여 그 결과가 컨트롤러에 전달될 수 있도록 접합되어 있다. 또한 왼쪽에 인접한



(그림 5) 제안된 선형 배열 방식 VLSI구조
(Fig. 5) The overview fo the proposed system

프로세서에 데이터를 전송할 수 있도록 프로세서들이 접합되어 있다. 한편, 컨트롤러 안에는 계수기(Counter)가 있어 최대 매칭 글자열의 글자 수를 기록한다.

4.2 프로세서 구조

(그림 6)은 프로세서의 구조를 보여준다. 각 프로세서는 사전 레지스터, 비교기(Comparator), 상태(State) 레지스터, 이전상태(Pre-state) 레지스터, 그리고 버스 및 컨트롤(Control) 신호들로 구성된다. 각 프로세서의 사전 레지스터는 다른 프로세서의 사전 레지스터들과 함께 쉬프트 레지스터로 작동되며, 사전윈도우의 글자열은 각사전 레지스터들에 한 글자씩 저장된다. 비교기는 사전 레지스터에 저장되어 있는 글자와 Bus-1으로 들어오는 입력 글자를 비교한다. 그 외의 상태(State) 레지스터와 이전상태(Pre-state) 레지스터를 포함한 시스템의 다른 구성 요소는 시스템의 동작 원리와 함께 설명된다.

4.3 VLSI 시스템 작동 원리

앞에서 설명한 프로세서들과 그 선형 배열 접속을 갖는 VLSI 시스템은 가변적응식 사전방식 텍스트 압축을 최대 매칭 글자열의 단위로 처리한다. 각 최대 매칭 글자열의 처리시, 입력 버퍼의 첫 글자부터 시작하는 글자열들중 사전 윈도우와 최대로 매칭이 되는 (즉, 제일 긴) 글자열을 찾아 해당하는 LZ77 코우드로 변형 출력후, 입력버퍼의 첫 글자부터 시작하는 (즉, 바로 이전에 처리된 최대 매칭 글자열 다음의) 글자열에 대하여 다음은 스텝(Step)의 최대 매칭 글자열 처리를 수행한다. 각 최대 매칭 글자열의 자세한 처리 과정은 아래와 같다.

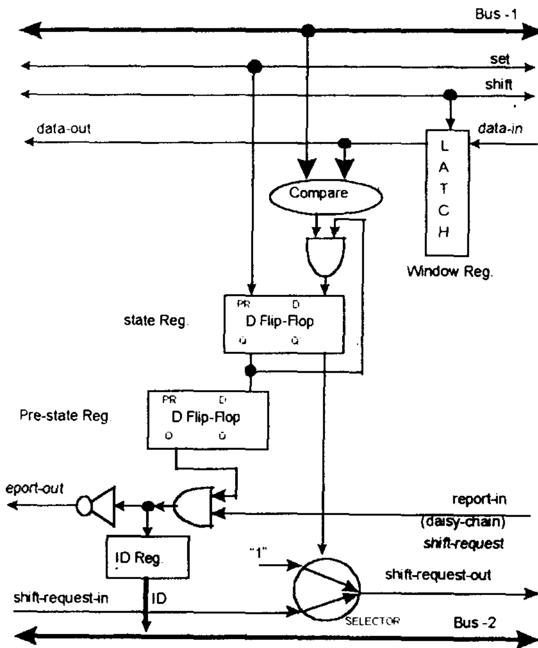
(단계 1): Bus-1을 통하여 입력되는 버퍼에 내장된 압축될 어구의 첫 번째 글자가 각 프로세서에 입력되어 그 프로세서의 사전 레지스터에 저장되어 있는 글자와의 글자 비교(Character comparison)가 이루어진다.

(단계 2): 글자 비교의 결과는 다음과 같은 절차로 처리된다. 먼저, 글자 비교의 결과로, 두 글자가 같은 경우 비교기의 출력단자에 '1'이, 그렇지 않은 경우에 '0'이 출력되어 상태값을 나타내는 상태(State) 레지스터의 이전 값과의 논리 앤드(AND)값이 다시 상태 레지스터에 저장된다. 이 새로운 값이 상태 레지스터에 저장되기 전, 현재의 값은 이전상태(Pre-state) 레지스터에 저장된다. 상태 레지스터의 초기값은 항상 '1'이 되도록 설계되어 있으며, 상태 어느 시점 't'의 레지스터의 값은 앞에서 설명한 상태값 S(t)를 나타내며 이전상태 레지스터의 값은 S(t − 1)의 값을 나타낸다. 맨 마지막까지 제일 긴 글자열을 발생한 프로세서의 경우 그 이전상태 레지스터의 값이 '1'이 되며, 중간에 매칭이 실패한 경우는 '0'으로 된다. 한편 최대 매칭 글자열 처리는 S(t)의 값이 '1'인 프로세서가 최소한 한 개 이상이면 계속되어야 하며, 데이지 체인(Daisy-chain) 구조로 연결된 shift-request-out 버스를 통하여 적정한 신호가 컨트롤러에 전달된다. 즉, 상태 레지스터의 값이 '1'인 프로세서의 경우에는 단일 비트 shift-request-out 버스를 통하여 '1'을 오른편 프로세서에 출력하여 컨트롤러로 하여금 최대 매칭 글자열 처리가 계속 되어야 함을 알려준다. 그와는 반대로 상태 레지스터의 값이 '0'인 프로세서 경우에는 왼편 프로세서로 부터 전달된 신호를 오른)P 프로세서에 바로 전달함으로 최대 매칭 글자열 처리의 결정에 영향을 미치지 않게 된다.

(단계 3): 컨트롤러는 shift-request-out신호를 점검하여 그 값이 '1'이면 모든 프로세서로 하여금 레지스터의 값을 왼편에 접속된 프로세서에 쉬프트(Shift)시키도록 컨트롤 신호를 보냄으로써 사전 윈도우를 한 칸 왼편으로 이동시킨다. 이때 입력 버퍼도 함께 왼편으로 이동되어 바로 이전에 처리가 된 입력 버퍼의 첫 글자는 오른편 마지막 프로세서에 입력된다. 또한, 컨트롤러는 계수기의 숫자를 '1' 증가 시켜 현재의 최대 매칭 글자열을 크기를 수정시킨 후, (단계 1)부터 다시 다음 글자를 처리하게 된다.

(단계 4): (단계 3)과는 반대로, 모든 프로세서의 S(t) 값이 '0'인 경우에는, 컨트롤러는 report 신호로 '1'을 출력시켜, S(t-1)를 나타내는 이전상태 레지스터에 '1'을 갖고 있는 프로세서는 프로세서명 (processor ID)을 Bus-2에 출력하도록 한다. 여기서, 매칭된 글자열이 서로 다른 프로세서에 여러 개 있을 수 있으며, 그러한 경우에 여러 프로세서로 부터 동시에 한 개 이상의 프로세서명이 Bus-2에 출력되는 것을 피하기 위하여, report신호는 데이지 체인(Daisy-chain)구조로 되어 있어 가장 먼저 발견되는 프로세서명만이 선택

된다. 한편, 컨트롤러는 프로세서명으로부터 매칭된 어구와 시작 위치를 계산하고, 또한 컨트롤러내의 계수기의 값으로 나타내지는 글자 수를 이용하여, 해당하는 LZ77압축 코우드를 생성시켜 출력한다 (단계 5 참조). 그후 각프로세서에 신호를 보내 다음 글자열에 필요한 부분의 초기화(initialization)를 시킨 후, 단계 1로 돌아가 다음 최대 매칭 글자를 처리한다.



(그림 6) 프로세서의 구조
(Fig. 6) The structure of a processor

(단계 5):(단계 4)에서 최대 매칭 글자열이 발견되었으며 그 프로세서의 ID가 PID_m이고 컨트롤러의 계수기 값이 Count일 때, 해당하는 LZ77 코우드는 다음과 같이 산출된다. 먼저 LZ77 코우드를 산출하기 위해 필요한 정보는 (1)매칭된 입력 글자열이 발견된 프로세서의 이름과 (2)그 글자열의 글자 갯수, (3)매칭된 입력 글자열과 사전 글자열의 상대적인 위치를 나타내 주는 Offset값, 그리고 (4)다음 입력글자열의 첫번째 글자이다. 제안된 시스템에서 최대 매칭 글자열이 발견되었을 때, 매칭된 입력 글자열은 항상 맨 오른편 프로세서부터 Count개의 프로세서에 저장된다. 그러므로, 사전 윈도우의 크기가 W인 시스템에서 사전윈도우 프로세서들의 ID를 왼편부터 1, 2,..., W로 정의 할때, 매칭된 글자열의 첫 번째 글자열을 저장하고 있는 프로세서의 ID인 PIDin은 아래와 같이 결정된다.

$$PID_{in} = W - Count + 1 \qquad (a)$$

그리고, 매칭된 사전 글자열은 항상 최대 매칭 글자열이 발견된 프로세서의 바로 왼편의 Count개의 프로세서에 저장되기 때문에 그 첫번째 프로세서의 ID인 PIDd 는 아래와 같이 결정된다.

$$PID_d = PID_m - Count \qquad (b)$$

한편, Offset값은 다음과 같이 결정된다.

$$\begin{aligned} Offset &= PID_{in} - PID_d \\ &= W - PID_m + 1 \end{aligned} \qquad (c)$$

결국, 제안하는 시스템은 최대 매칭 글자열의 발생시 Count와 PID_m을 발생시키고, 다음 입력 글자열의 첫 글자는 입력 버퍼의 첫 글자 위치에 놓여져 있으므로 〈Offset, Count, 다음 입력글자열의 첫 글자〉로 정의되는 LZ77 코우드를 생성시킬 수 있으며, 프로세서명이 PID인 프로세서에 저장, 그 매칭 입력 글자열을 대치시킴으로 LZ77 코우드의 생성 및 처리를 완벽하게 구현하고 있음을 볼 수 있다. 여기서 최대 매칭 글자열의 크기가 '1'인 경우나 전혀 매칭이 일어나지 않는 경우에는 LZ77 코우드를 생성시키지 않고 입력버퍼의 첫 글자를 그냥 왼편 프로세서에 입력시키기 때문에 입력 텍스트에 대해 생성되는 최종 압축코우드는 LZ77 코우드와 보통 글자들로 구성된다.

(그림 7)은 위에서 설명한 시스템의 동작을 좀더 분명히 하기 위한 간단한 예를 보여준다. 여기서 보여지는 시스템의 사전윈도우의 크기가 '7'이며, '7'개의 프로세서에 이미 압축 처리된 입력 텍스트 'bcbacba'가 각 프로세서의 사전 레지스터에 저장되어 있다. 그리고 압축될 입력 글자열 'babc'가 입력 버퍼에 저장되어 있으며, 그 처리과정은 아래와 같다.

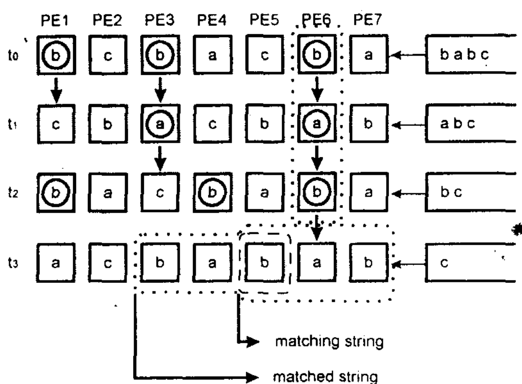
(t0) 입력 버퍼의 첫 글자 'b'에 대하여 글자 비교가 각

프로세서에서 일어나며 그 결과 사전 레지스터의 글자가 'b'일 경우 S(t1)의 값이 '1'이 되고 그렇지 않을 경우는 '0'이 된다. (그림 6)에서 S(t1)의 값이 '1'인 경우는 화살표로 표시되어 있으며, PE1, PE3, 그리고 PE6에서 S(t1)의 값이 '1'이 되는 것을 볼 수 있다.

(t1) 처리되는 입력 글자는 'a'이며 글자 비교 결과 PE3와 PE6에서 S(t1)의 값이 '1'이 된다.

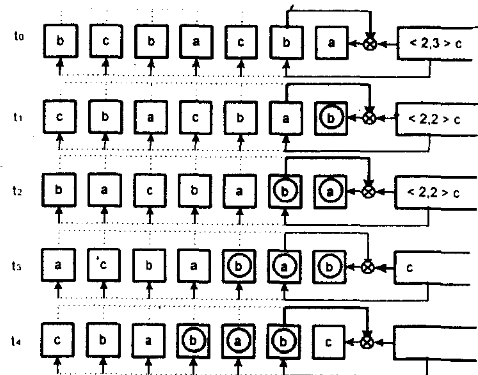
(t2) 처리되는 입력 글자는 'b'이며, 글자 비교 결과 PE1, PE4, 그리고 PE6에서 글자 매칭이 일어나지만 PE6와는 달리 PE1과 PE4의 경우에는 S(t1)의 값이 '0'이므로 그 S(t2)값은 '0'이 된다.

(t3) 처리되는 입력 글자는 'c'이며, 글자 매칭이 일어나는 프로세서가 없으므로 모든 프로세서의 S(t3)의 값이 '0'이 된다. 이때 앞에서 설명한 (단계 4)에 의하여, S(t2)의 값이 '1'인 PE6가 최대 매칭 글자열을 발생시킨 프로세서가 되어, 그 프로세서명 '6'을 컨트롤러에 보낸다. 여기서 W=2, PIDm=6, Count=3 이므로 이 값들을 (단계 5)의 식 (c)에 대입하여, Offset =2 (즉, 7−6+1)를 얻어 LZ77 코우드〈2, 3〉를 생성하게 된다. (여기서 LZ77 코우드의 3번째 성분인 다음 글자열의 처음 글자인 'c'는 본 시스템의 압축 과정에서 입력버퍼의 첫 글자에서 항상 발견할 수 있으므로 생성시키지 않는다.)



(그림 7) 인코딩 설명
(Fig. 7) Sample encoding operations

LZ77 코우드와 텍스트의 글자들로 구성된 LZ77 압축 코우드의 원상 회복(Decompression)에 관한 설명은 본 논문의 주된 발표 범위가 아니므로 앞절에서 생략했으나, 아래에 간단히 LZ77 코우드로 구성된 압축 코우드를 원래의 텍스트로 회복시키는 과정을 설명한다. 압축 과정과는 반대로 입력 버퍼는 압축 코우드를 갖으며, 시스템의 출력은 원래의 텍스트가 된다. 회복 과정은 기본적으로 입력 버퍼의 LZ77 코우드에 대하여 해당하는 글자열을 이미 처리된 사전 윈도우에서 찾아 바꿔서 한 글자씩 사전 맨 오른편 프로세서에 입력시켜주는 원리에 준하여 구현된다. (그림 8)은 위의 과정을 LZ77 코우드 〈2, 3〉에 대하여 각 글자 처리시의 사전윈도우 프로세서의 내용을 보여준다. 즉 LZ77 코우드 〈2, 3〉을 만났을 때, 먼저 Offset값인 '2'를 이용하여 사전 윈도우의 매칭 글자열의 첫 글자를 저장하고 있는 참조(Reference) 프로세서인 PE6을 (즉, PID=W−Offset+1) 찾은 다음, Count값의 시간 동안은 프로세서의 모드(mode)를 대치 모드(Replacement mode)로 변환시켜 동작시킨다. 대치 모드 중에는 LZ77 코우드의 Count값을 각 글자 처리시마다 '1'씩 감소시키며 '0'이 될때까지 참조 프로세서인 PE_6의 사전 윈도우의 글자를 사전 윈도우의 제일 오른편 프로세서에 입력시킨다. Count값이 '0'이 되면, 입력 버퍼에서 처리된 LZ77 코우드를 제거하고, 그 다음 코우드의 처리를 시작한다.



(그림 8) 디코딩 설명
(Fig. 8) Sample decoding operations

5. 제안된 방식의 VLSI구현시 처리 속도와 집적 효율

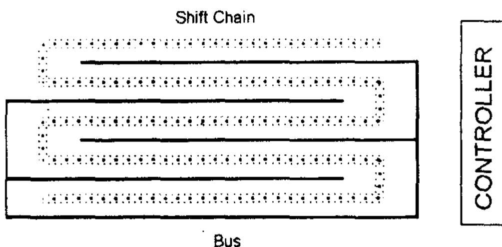
앞에서 설명한 제안된 방식의 구조 및 동작 원리를

갖는 시스톨릭 어레이의 VLSI구현에 있어서 그 속도와 집적도에 대한 분석은 실제 적용 측면에서 매우 중요하다. 본 절에서는 제안하는 방식의 성능의 정량적 분석 결과를 기술한다. 성능 분석은 VLSI 시스템의 성능 평가에서 가장 중요한 두 요소인 (i) 시스템의 속도와 (2) VLSI 구현에 있어서의 복잡도 및 효율성를 중심으로 이루어 졌으며, 제안하는 방식과 기존의 대표적인 방식인 M. Gonzales와 J. Storer의 Match Tree Architecutre(MTA)[3]와의 성능비교가 이루어졌다.

5.1 제안 VLSI 시스템의 속도

압축 속도면에서의 텍스트 압축 시스템(System)의 성능은 어프리케이션(Application)의 성격에 따라 상이한 결과를 나타내게 되므로 그 결정요소를 추출해 내기가 어렵다. 그러나, 길이가 (w)이고 최대 매칭어구의 길이가 (m)인 텍스트를 압축하는 시스템을 구현할 경우, 일반적으로 다음의 세가지 요소가 주요한 성능 결정 요소가 된다[17].

1. $Time(w, m)$: 인코더(Encoder)가 입력 텍스트를 처리하거나, 디코더(Decoder)가 출력 코드를 생성시키는 최대 시간(Worst-case time)
2. $Lag(w, m)$: 입력 글자가 시스템 pipe에 들어오는 시점부터 나갈 때까지 걸리는 최대시간(Worst-case time)
3. $Restart(w, m)$: 새로운 입력 데이터의 매칭을 시작하기 위하여 사전을 초기화시키는데 걸리는 최대 시간(Worst-case time)



(그림 9) 제안된 방식의 웨이퍼 구현의 예
(Fig. 9) An Example layout of the proposed system

Match Tree Architecture는 트리 방식을 사용하여 매칭을 수행하므로, log(w)만큼의 시간이 최대로 긴 매칭 어구를 찾는데 필요하게 된다. 결국, 위의 요소들에 대하여 M. Gonzales와 J. Storer가 제안한 Match Tree Architecture(MTA)는 다음과 같은 값을 갖는다[17].

- 인코더 : $Time(w, m) = O(log(w))$
- 디코더 : $Time(w, m) = O(1)$
- $Lag(w, m) = O(w)$
- $Restart(w, m) = O(log(w))$

본 고에서 제안된 시스톨릭 어레이는 사전윈도우의 크기에 상관없이 항상 일정한 처리 속도를 갖으며, 주어진 글자가 시스템 Pipe에 입력된 후 나갈때까지의 시간은 사전크기에 비례하며, 새로운 매칭을 수행하기 위해 MTA에서와 같은 Matching Tree의 초기화가 필요하지 않다. 그러므로 위의 성능 결정 요소들의 값은 다음과 같다.

- 인코더 : $Time(w, m) = O(1)$
- 디코더 : $Time(w, m) = O(1)$
- $Lag(w, m) = O(w)$
- $Restart(w, m) = O(1)$

본고에서 제안하는 시스템은 그 동작 원리가 기본적으로 쉬프트 동작과 간단한 신호 전송으로 구성되므로 향상된 속도를 얻을 수 있는 장점이 있다. 실제로 위의 결과에서 볼수 있듯이, 본 고에서 제안하는 방식은 압축(Encoder의 성능)이나 회복(Decoder의 성능)면에서 우수한 성능을 보이고 있으며, 기본적으로 텍스트 압축 시스템에서 구현할 수 있는 이상적인 성능을 보유하고 있음을 알수 있다.

5.2 VLSI 시스템 구현성

한편, 압축 시스템의 VLSI 구현성을 평가함에 있어서 많은 요소들을 고려해야 하지만, 아래와 같은 요소들을 가장 중요한 요소라 할 수 있다.

1. 시스템 Pipe의 각 셀(Cell)의 구현에 필요한 Hardware의 양
2. 시스템의 웨이퍼 집적도

3. Multi-Chip을 사용한 확장성

위의 세요소들에 대하여 본고에서 제안하는 시스템과 Match Tree Architecture(MTA)의 성능을 비교하면, 먼저 각셀의 구현에 필요한 Hardware양은 MTA와 제안하는 시스템은 비슷하다고 할 수 있다. 그 이유로는 MTA의 경우 Matching Tree를 위한 부가적인 Hardware가 필요한 반면, 제안하는 방식은 버스와 각 셀내에 상태 레지스터등의 부가적인 Hardware가 필요하기 때문이다. 웨이퍼 집적도 면에서 보면, 사전 윈도우 크기가 W일 때, MTA의 경우에는 사전 윈도우 처리를 위한 선형 Pipe와 (W-1)개의 노드로 구성되는 바이너리 트리 구조의 Matching Tree를 집적해야 하나, 제안하는 시스템은 Matching Tree가 필요없는 선형 Pipe로 구성되므로 효율적이다. 실제로 제안하는 시스템의 선형 구성은 아래 (그림9)에서 볼 수 있는 것과 같이 버스와 프로세서 어레이를 선형으로 배치시켜 웨이퍼에 집적할 수 있게 된다. 마지막으로 확장성을 살펴보면, MTA의 경우 Matching Tree로 인하여, 서로 다른 웨이퍼로 구성된 압축 시스템의 연결이 가능한 웨이퍼의 구성이 매우 복잡하다. 그러나, 제안하는 시스템은 선형 Array의 형태를 갖으므모 기존의 웨이퍼 배치를 바꾸지 않고, 다른 웨이퍼를 서로 연결, 보다 확장된 윈도우 크기를 갖는 시스템을 용이하게 구성 할 수 있는 장점이 있다.

6. 결 론

본 논문은 LZ77의 병렬처리를 위한 병렬 LZ77알고리즘과 그 병렬 LZ77 알고리즘을 처리하도록 고안된 VLSI 시스템 구조에 관한 연구 내용을 다루고 있다. 이전의 다른 연구 내용과 비교하여, 본 논문에서 제안된 VLSI시스템은 기능면에서나 실제 웨이퍼 구현 면에서 사전 윈도우의 크기에 대한 확장성이 뛰어나 사전 윈도우나 압축 부피의 크기에 제한을 갖지 않는 장점이 있으며, 성능 면에서도 입력 텍스트의 길이가 N일 때, 사전 윈도우의 크기에 상관없이 항상 처리속도가 O(N)이며 시스톨릭 어레이 구현시 향상된 웨이퍼 집적도를 갖는 이상적인 성능을 보유하고 있다.

참 고 문 헌

[1] S. Agostino and J. Storer, Parallel Algorithm for Optimal Compression using Dictionaries with the Prefix Property, *Proceedings IEEE Data Compression Conference*, Snowbird, Utah, 1992.

[2] E. Fiala and D. Greene, Data Compression with Finite Windows, *Communications of the ACM 32 :4*, pp. 490-505, 1989.

[3] M. Gonzalez and J. Storer, Parallel Algorithms for Data Compression. *Journal of the ACM.32:2* pp. 344-373, Apr. 1985.

[4] A. Hartman and M. Rodeh, Optimal Parsing of Strings, Cobimatorial Algorithms on Words, *Springer-Verlag*, pp. 155-167.

[5] H. Kucera and W. Francis, Computational Analysis of Present-Day American English. *Brown Univ. Press*, Providence, R. I., 1967.

[6] G. Landon and J. Rissanen, A Simple General Binary Source Code, *IEEE Transaction on Information Theory 28:5*, pp. 800-803, 1982.

[7] D. Lelewer, Data Compression on Machine with Limited Memory, *Ph.d Dissertation*, Dept of Information and Computer Science, University of California, Irvine, 1991.

[8] C. Mead and I. Conway, Introduction to VLSI System. Addison-Wesley, 1982.

[9] M. Patterson, M. Ruzzo, L. Snyder, Bounds on minimax edge length for complete binary trees. In Preceedings of the 13th Annual ACM Symposium on Theory of Computing. New York, pp. 293-299.

[10] J. Rief and J. Storer, Adaptive Lossless Data Compression over a Noisy Channel, *Proceedings Communication, Security, and Sequences Conference*, Positano, Italy

[11] M. Rodeh, V. Pratt, and S. Even Linear, Linear algorithm for data compression via string matching. J. ACM 28, 1 (Jan. 1981), pp. 16-24

[12] J. Seery and J. Ziv, A Universal Data Compression Algorithm: Description and Preliminary

Results, *Technical Memorandum 77-1212-6*, Bell Laboratories, Murray Hill, N.J.

[13] J. Seery and J. Ziv, Futher Results on Universal Data Compression, *Technical Memorandum 78-1212-8*, Bell Laboratories, Murray Hill, N.J.

[14] J. Storer and T. Szymanski, Data Compression via Textual Substitution. Journal of ACM. 16:3, pp. 928-951. Oct. 1982.

[15] J. Storer, NP-completeness results concerning data compression. Tech. Rep. 234, Dept. EE and Computer Sci, Princeton Univ, N. J.

[16] J. Storer, J. Rief, and T. Markas, A Massively Parallel VLSI Design for Data Compression using a Compact Dynamic Dictionary, *Proceedings IEEE VLSI Signal Processing Conference*, San Diego, CA, 1990.

[17] J. Storer, Image and Text Compression, Kluwer Academic Publishers, pp. 159-178, 1992.

[18] C. Timothy, B. John, G. Cleary, and H. Written, Text Compression. Prentice Hall. 1990.

[19] R. Wagner, Common Phrases and Minimum Text Storage, Communications of the ACM 16, pp. 148-152, 1973.

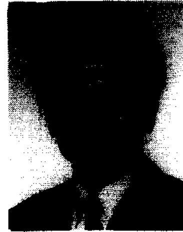
[20] T. Welch, A technique for High-Performance Data Compression, *IEEE Computer 17:6*, pp. 8-19, 1984.

[21] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression. *IEEE Transaction or Information Theory*, 23:3, pp 337-343, 1977.

[22] J. Ziv and A. Lempel, "Compression fo Individual Sequences Via Variable-Rate Coding", *IEEE Transactions on Information Theory 24:5*, pp. 530-536, 1978.

[23] R. Zito-Wolf. "Broadcast/Reduce Architecture for High Speed Data Compression, Proceedings Second IEEE Symposium on Parallel and Distributed Processing", Dallas, TX, pp. 174-181, 1990.

[24] 이용두, 채수환, "LZ77 텍스트 압축방식의 병렬 처리를 위한 VLSI 구조", 1994년 한국정보처리 응용학회 추계 학술 발표 논문집 제 1권 22호 pp. 231-234.

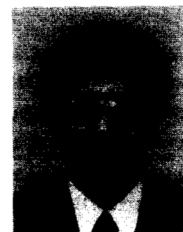


이 용 두

1975년 한국항공대학교 통신공학과 졸업(공학사)
1982년 영남대학교 대학원 전자공학과(공학석사)
1995년 한국항공대학교 대학원 전자공학과(공학박사)
1982년~현재 대구대학교 정보통신공학부 교수
1981년~1982년 (일)동경대학 전자공학과 객원교수
1991년~1993년 Univ. of Southern California 교환교수
관심분야: 컴퓨터구조, 컴퓨터통신, Internet 응용기술

김 희 철

983년 연세대학교 전자공학과 졸업(공학사)
991년 Univ. of Southern Californi (Computer Eng. M.S.)
996년 Univ. of Southern Californi (Computer Eng. Ph.D.)
983년~1988년 (주)삼성전자 주임연구원
1996년~1997년 (주)삼성SDS 수석연구원
1997년~현재 대구대학교 정보통신학부 전임강사
관심분야: 병렬처리, 컴퓨터구조, 컴파일러



김 중 규

1984년 연세대학교 전자공학과 졸업(공학사)
1986년 연세대학교 대학원 전자공학과(공학석사)
1992년 연세대학교 대학원 전자공학과(공학박사)
1992년~현재 대구대학교 정보통신공학부 조교수
관심분야: 컴퓨터 네트워크, 통신망 성능분석, 통신프로토콜