정형 허프만 코드의 효율적인 부호화기의 설계

박 화 식[†] · 조 경 연[††]

요 약

본 논문에서는 정형 트리 형태의 허프만 코드를 연속된 '1'의 수를 나타내는 필드와 나머지 비트열을 나타내는 필드로 구성된 허프만 테이블에 의한 새로운 부호화 알고리즘을 제안한다. 제안한 알고리즘은 테이블의 구성이 단순하고 부호화 과정이 Run-length 복호와 시프트 연산 만을 수행함으로 하드웨어 구현에 적합하다. 제안한 부호화기는 COMPASS 상에서 VHDL로 설계하고 시뮬레이션하여 동작을 확인하였다. 설계한 칩은 0.8 미크론의 반도체 공정에서 약 5,000 개의 게이트가 소요되었으며, 25 MHz의 클럭에서 정상적으로 시뮬레이션 되었다. 본 논문에서 제안한 허프만 부호화기는 하드웨어 구현에 적합하므로 실시간 처리를 필요로 하는 멀티미디어와 데이타 통신 분야 등에 폭넓게 적용될 것이 기대된다.

Design of the Efficient Encoder for the Canonical Huffman Code

Hwa-Sik Park[†] · Gyung-Yun Cho[††]

ABSTRACT

In this paper, a new Huffman encoding algorithm with memory based Huffman table is proposed. The memory based table on the Huffman code of the canonical tree form consists of the Run-length code of the consecutive '1's and the rest of the code bit stream. The proposed algorithm is suitable for the hardware implementation since the construction of the table is simple and the encoding process is operated by only performing Run-length decoding and shift operation. The proposed encoder is implemented by using VHDL, simulated and verified the operation on the COMPASS. The designed chip occupies about 5,000 gates using 0.8 micron process. It is normally simulated at the 25 MHz clock. The proposed Huffman encoder is expected that it will be widely applied to the real-time processing such as the field of Multimedia, digital data communications, since it is suitable for the hardware implementation.

1. 서 론

데이터 압축 기법은 데이터를 처리하기 위해 필요한 기억 장소와 통신 채널의 대역폭을 감소시킨다. 이러한 기법은 컴퓨터 시스템에서 파일 백업과 복구 비용의 감소, 보안성의 증가, 그리고 파일의 효율적인 검색과 같은 추가의 이점을 제공한다. 대부분의 전통적인 압축 기법은 추가의 압축율을 얻기 위해 부호화기의 마지막 출력 데이터에 다시 가역 부호화 방법을 사용한다. 이러한 방법은 데이터베이스, 문서 전송 시스템, 통신망, 그리고 고성능 슈퍼 컴퓨터의 설계에 유용하다. 1952년에 허프만에 의해 제안된 평균 부호 길이를 최소로 하는 허프만 부호화 방법[1]은 추가의 압축율을 얻기 위한 가역 부호화 방법이고 고정된 길이의 입력에 대한 최적 부호화를 제공하는 최적의 가

† 정 회 원:아남 에스엔티 연구소 연구원
†† 정 회 원:부경대학교 공과대학 컴퓨터공학과
논문접수:1996년 12월 2일, 심사완료:1997년 10월 17일

변 길이 부호화방법으로도 유명하다. 이러한 허프만 부호화 방법은 현재 많은 분야에서 이용되고 있으며, 다른 부호화 기법과 결합되어서도 사용되고 있다. 특히, Run-length 부호화 방법과 결합된 허프만 부호화 방법은 JPEG 또는 MPEG 등 DCT 기반의 영상 압축 및 복원에 많이 사용되고 있다.

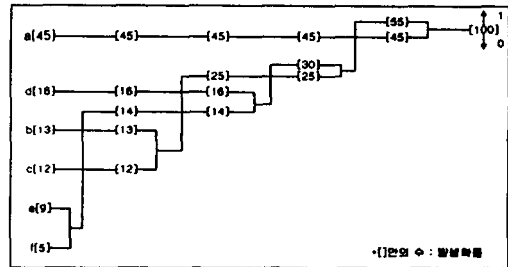
멀티미디어와 디지탈 데이타 통신이 보급되고 실시간적인 데이터의 압축과 복원이 요구되면서 하드웨어에 의한 허프만 부호화기와 복호화기의 구성 방법이 많은 연구자에 의해 제안되어 왔다[2]-[9], [11], [12]. 허프만 트리의 역트리 표현을 일대일로 하드웨어에 대응시킨 [2]의 방법은 허프만 테이블을 저장하기 위한 메모리가 필요없지만, 역 허프만 테이블을 하드웨어에 일대일로 대응시키므로 하드웨어가 너무 복잡해진다. 허프만 트리를 사용하여 각 입력 기호를 부호화할 때의, 트리 검색 과정을 메모리에 사상시켜 허프만 부호화기와 복호화기를 구성한 [5]의 방법은 부호화가 1-bit 씩 수행됨으로 메모리 읽기 동작은 평균적으로 허프만 부호의 평균 부호 길이만큼 발생한다. [5]의 방법과 유사하게 입력 기호에 대한 부호화 과정을 2-bit 또는 1-bit씩 수행하는 [4]의 방법은 512 ×12-bit 크기의 아주 큰 메모리를 요구한다. 또한 이 방법은 부호화 알고리즘이 매우 복잡하고 허프만 테이블의 구성 방법이 매우 복잡하며, 8-bit ASCII 문자를 부호화하는데 7 클럭 사이클이 필요하다. 정형(Canonical) 트리의 특징을 이용한 군집화 기법을 사용하여 허프만 테이블을 구성하고, 이 테이블을 패턴 매칭 기법에 의해 검색함으로써 허프만 복호화를 수행하는 [7],[11], 그리고 [12]의 논문들은 JPEG이나, MPEG 등의 DCT 기반의 영상 복원에 적합하지만, 256개의 잎 노드를 가지는 8-bit ASCII 문자에 대한 테이블 구성은 대단히 복잡해진다.

본 논문에서는 기존의 연구에서 나타난 문제점들을 해결하기 위하여 정형 트리의 특징인 부호어에 연속된 '1'의 수가 많다는 것을 이용하여 종래의 허프만 테이블을 수정한 수정 허프만 테이블의 구성 방법을 제안한다. 또한 제안한 수정 허프만 테이블을 메모리에 사상시켜 입력 기호의 부호화 과정을 수행할 때, 메모리 읽기 동작이 한 번만 발생하도록 한다. 그리고 실시간 부호화를 위해서, 수정 허프만 테이블에 기초하여 하드웨어 구현에 적합한 부호화 알고리즘을 제안하며, VHDL을 사용하여 제안한 알고리즘을 하드웨어로 구현한다. 본 논문에서 제안한 허프만 부호화기의 성능을 평가하기 위해, 8-bit ASCII 문자를 예로 들어, 표준 CTCC에서 제공된 파일에 대한 허프만 테이블을 본 논문에서 제안한 수정 허프만 테이블로 구성하고, 이것을 메모리에 사상시켜서 하드웨어를 설계한다.

2. 허프만 부호화와 정형 트리

허프만 부호화 알고리즘은 최적의 부호를 생성할 수 있는 방법으로 각 입력 기호의 발생 확률을 이용하여 부호어를 생성한다[1][10]. 예로써, 6개의 기호들을 가지고 있는 기호 집합, S={a, b, c, d, e, f}의 발생 확률이 F={45, 13, 12, 16, 9, 5}일 때(여기서의 발생 확률은 계산의 편의를 위해 ×100을 한 값이다), 허프만 부호를 구성한 것이 (그림 1)이다.

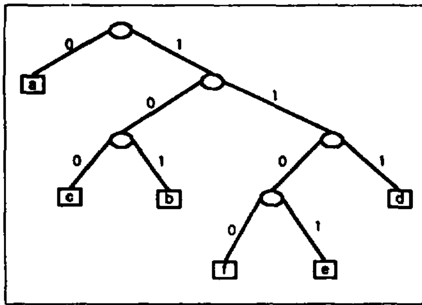
(그림 1)에서 구해진 각 기호들의 부호어를 〈표 1〉에 나타내고 허프만 트리로 표현한 것이 (그림 2)이다.



(그림 1) 허프만 부호의 구성 방법
(Fig. 1) Construction method of the huffman code

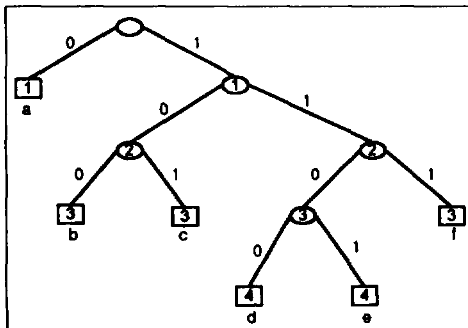
〈표 1〉 (그림 1)에 대한 허프만 테이블
〈Table 1〉 Huffman Table for the (Fig. 1)

기호	부호어	부호어 길이
a	0	1
b	101	3
c	100	3
d	111	3
e	1101	4
f	1100	4



(그림 2) 표 1에 대한 허프만 트리
(Fig. 2) Huffman tree for the <Table 1>

허프만 부호의 특징을 유지하면서 트리의 형태가 한 쪽으로 치우친 정형 트리는 부호어 길이를 오름차순으로 정렬하여 구성할 수 있다. 만약 두 개 이상의 기호들이 같은 길이를 가지고 있다면, 이 기호들의 기호 값으로 다시 오름차순 정렬을 행한다. 이렇게 정렬된 각 기호의 길이가 트리에서 잎 노드에 해당한다. 다음으로 가장 큰 두 개의 길이를 자식 노드로 가지는 부모 노드를 만들고 그 부모 노드의 부호어 길이를 [자식 노드의 부호어 길이 −1]로 설정한다. 만약, 가장 큰 부호어 길이가 2개 이상 존재한다면, 정렬된 순서에서 가장 오른쪽에 있는 2개의 잎 노드를 선택한다. 이 과정을 모든 잎 노드에 대해서 반복한다. (그림 3)은 <표 1>에 대한 정형 트리를 보여준다.



(그림 3) 정형 트리
(Fig. 3) Canonical tree

3. 수정 허프만 테이블과 부호화 알고리즘

3.1 수정 허프만 테이블

정형 트리는 "각 기호의 부호어 비트열에서 비트 1의 연속이 많다"는 특징을 가지고 있으며, (그림 3)을 허프만 테이블로 나타낸 <표 2>에 이러한 특징이 잘 나타나 있다. <표 2>로부터 연속된 비트 '1'을 Run-length로 표현한 수정 허프만 테이블을 <표 3>에 보인다. <표 3>의 '연속된 '1'의 수' 항목은 <표 2>의 각 기호의 부호어에서 연속된 '1'의 수를 나타내고, 그리고 '나머지 비트열'과 '길이' 항목은 각각 <표 2>에서의 연속된 '1'을 제외한 나머지와 그 길이가 된다.

<표 2> (그림 3)에 대한 허프만 테이블
<Table 2> Huffman Table of the (Fig. 3)

기호	부호어	부호어 길이
a	0	1
b	100	3
c	101	3
d	110	3
e	1110	4
f	1111	4

<표 3> 수정 허프만 테이블
<Table 3> Modified Huffman Table

기호	연속된 1의 수	나머지 비트열	길이
a	0	0	1
b	1	00	2
c	1	01	2
d	2	0	1
e	3	0	1
f	4	−	0

수정 허프만 테이블을 메모리에 사상시키기 위하여 (그림 4)와 같이 메모리의 한 워드를 2개의 필드로 나눈다.

주소 | 필드 1 | 필드 2 |

주소 : 각 기호 값
필드 1 : 연속된 '1'의 수
필드 2 : 연속된 '1'을 제외한 나머지 비트열과 길이를 추출하기 위한 추가 비트

(그림 4) 메모리 워드의 구성
(Fig. 4) Construction of the memory word

(그림 4)의 메모리 워드 구성 방법을 사용하여 〈표 3〉을 메모리에 사상하면 (그림 5)와 같이 된다.

	필드 1	필드 2
a	000	010
b	001	100
c	001	101
d	010	010
e	011	010
f	100	001

(그림 5) 〈표 3〉의 메모리 사상
(Fig. 5) Memory mapping of the 〈Table 3〉

〈표 3〉의 나머지 비트열은 필드 2의 최상위 비트부터 시작하여 최초의 '1' 다음의 비트열이 되고, 그 길이는 최초의 '1'이 나타난 비트 번호가 된다. 예를 들어, (그림 5)에서 기호 'a'는 비트 번호 1에 최초로 '1'이 나타나기 때문에 나머지 비트열의 길이는 1이 된다. 또한, 기호 'f'는 비트 번호 0에 '1'이 있으므로 그 길이는 0이 됨을 알 수 있다.

3.2 부호화 알고리즘

(그림 5)와 같이 구성한 메모리 상에서의 허프만 부호화 알고리즘은 (그림 6)과 같다. (그림 6)에서 입력 기호를 주소로 하여 메모리를 읽은 후, (그림 4)의 필드 2를 우선순위 인코더에 통과시켜 구해진 나머지 비트열의 길이와 필드 1의 값에 따라 Run-length 복호 또는 오른쪽 시프트 연산을 취함으로써 부호화를 수행한다.

```
repeat
   Field-1 ← Mem[8-bit input symbol] /*See
   Field-2 ← Mem[8-bit input symbol] /*<Fig. 4>
   Length ← Priority_Encoder(Field-2) /*<Table 3>
   if (Field-1 = 0) then /*CW : CodeWord
     CW ← Shr(Field-2, Length) /*Shift Right
   else if (Length = 0) then
       CW ← RLD(Field-1) /*Run-length decode
     else
       CW ← RLD(Field-1) ⊕ Shr(Field-2, Length)
     end if
   end if
until end-of-input-symbol
```

(그림 6) 부화화 알고리즘
(Fig. 6) Encoding algorithm

4. 하드웨어 설계 및 구현

4.1 알고리즘 구현

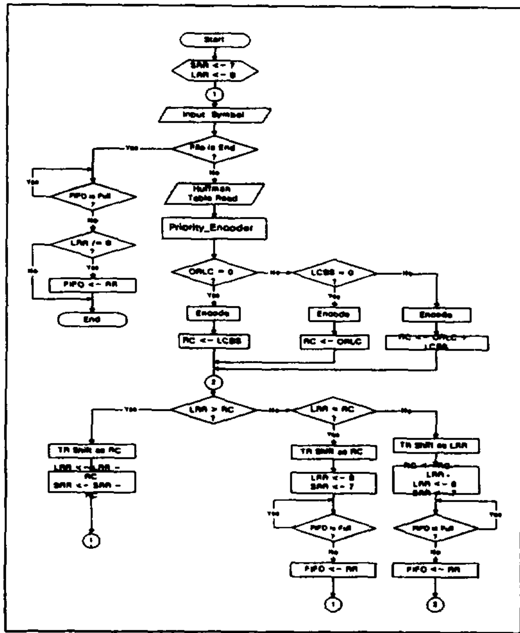
본 논문에서 제안한 허프만 부호화 알고리즘은 종래의 허프만 테이블을 정형 트리의 특징인 '각 부호어에 연속된 '1'의 수가 많다'는 것에 기초하여 허프만 테이블을 수정하였다. 이렇게 수정된 테이블을 사용하는 부호화 알고리즘을 구현하기 위한 레지스터들이 〈표 4〉에 나타나 있다.

〈표 4〉 레지스터 정의
〈Table 4〉 Register Definition

ORLC	One Run-Length Code
CBS	Code Bit Stream
LCBS	Length of CBS
TR	Temporary Register
RR	Result Register(8-bit)
RC	Result Counter
LRR	Length of Result Register(4-bit) Initial value is 8.
SRR	Start of Result Register(3-bit) Initial value is 7.

〈표 4〉에서 ORLC는 (그림 4)의 필드 1로 연속된 '1'의 수를 저장하는 레지스터이고 CBS는 (그림 4)의 필드 2를 저장하는 레지스터이다. 그리고 LCBS는 CBS를 우선순위 인코더에 통과시켜서 구해진 나머지 비트열의 길이를 저장한다. 각 입력 기호에 대한 부호어를 저장하기 위해 TR이 사용된다. TR에 저장된 부호어를 8-bit 크기의 16-byte FIFO에 출력하기 위해 RR을 사용한다. TR에 저장된 부호어를 8-bit 단위로 RR에 저장하기 위해서 RC, LRR, 그리고 SRR을 정의한다. RC는 입력 기호의 부호어 길이를 저장하며, LRR은 RR에 저장될 수 있는 부호어의 최대 길이를 지시한다. 그리고, SRR은 RR의 8-bit 중에서 부호어가 저장되기 시작하는 비트 번호를 지시하는 포인터이다.

<표 4>에 나열된 레지스터들을 사용하여 본 논문에서 제안한 알고리즘을 구현한 것이 (그림 7)에 보인 순서도이다.



(그림 7) 허프만 부호화기의 순서도
(Fig. 7) Flowchart of the huffman encoder

순서도에서 초기화로 부호어를 RR의 비트 7부터 저장하기 위하여 SRR에 7을 설정하고, RR에 8 비트를 저장할 수 있도록 LRR에 8을 설정한다. ①에서 원시 심볼을 입력한다. 만약 파일의 끝이면, FIFO로 전송되지 않은 부호어가 RR에 남아 있는지를 판단하기 위하여 LRR의 내용을 조사한다. LRR이 8이 아니면, 마지막 입력 기호에 대한 부호어가 RR에 남아있으므로 이를 FIFO로 전송하고 부호화를 종료한다.

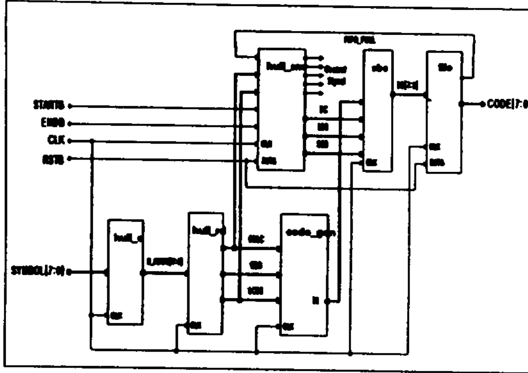
파일의 끝이 아니면, 원시 심볼을 어드레스로 하여서 허프만 테이블을 읽어 ORLC와 CBS에 저장하고 우선순위 인코더로 LCBS를 구한다. ORLC에 저장된 내용이 0이면, 연속된 '1'이 없으므로 CBS를 LCBS만큼 오른쪽으로 시프트하여 TR의 MSB(비트 번호 15)부터 저장하는 부호화를 수행하고 RC에 LCBS의 값을 저장한다. ORLC가 0이 아니고 LCBS가 0이면, 부호어가 모두 '1'로 구성된 것으로 ORLC의 값만큼 TR의 MSB부터 '1'로 세트시키는 부호화를 수행한다. 그리고 ORLC를 RC에 로드한다. ORLC와 LCBS가 모두 0이 아니면, 위에서 논의된 부호화 과정을 모두 수행하기 위하여 ORLC의 값으로 Run-length 복호화를 먼저 수행하고 CBS를 오른쪽으로 LCBS만큼 시프트하고 RC에 ORLC + LCBS를 로드한다. 이러한 부호화 과정으로 원시 심볼에 대한 부호어가 TR에 저장된다. LRR과 RC의 값을 비교하여 LRR이 RC보다 크면, RR에 저장될 수 있는 비트 길이가 부호어 길이보다 크기 때문에, TR에 저장된 원시 심볼에 대한 부호어를 RR로 전송하고 LRR과 SRR의 값을 변경한다. 그리고 LRR과 RC가 같으면 RR에 저장될 수 있는 비트 길이와 원시 심볼의 부호어 길이가 같기 때문에, TR에 저장된 부호어를 RR로 왼쪽 시프트 연산을 수행하여 전송하고, RR에 8 비트를 저장할 수 있도록 LRR과 SRR을 각각 8과 7로 설정한다. RR에 저장된 8비트 부호어들을 FIFO에 전송하기 전에 FIFO의 상태를 검사한다. LRR이 RC보다 크거나 같을 경우는 원시 심볼에 대한 부호어 모두가 TR에서 RR로 전송되었기 때문에 ①로 분기하여 새로운 원시 심볼을 입력받는다. TR에 저장된 원시 심볼에 대한 부호어 길이가 RR에 저장될 수 있는 비트 길이보다 크면, TR에 저장된 부호어를 LRR 만큼 시프트하여 RR에 전송하고 RC에는 LRR 만큼 감하여 저장하며 LRR과 SRR은 TR에 남아 있는 부호어를 RR에 저장하도록 각각 8과 7로 설정된다. 이 경우에는 RR의 부호어들을 FIFO로 전송하고 ②로 분기한다.

4.2 하드웨어 설계

본 논문에서 제안한 수정 허프만 테이블과 이것을 사용하는 부호화 알고리즘을 (그림 8)에 보인 하드웨어 블록도로 설계한다.

(그림 8)에서 STARTB 신호는 부호화의 시작을 나타내는 신호이다. 이 신호가 '0'일 때, huff_sm의 제어 신호에 의해 외부의 8-bit 버스를 통해 부호화될 기호가 huff_e에 래치된다. Huff_rd 블록은 huff_e에 래치된 기호를 주소로 하여 메모리 읽기 동작을 수행하고 내부에 있는 우선순위 인코더에 의해 구해진 LCBS와 ORLC, CBS를 huff_sm과 code_gen 블록으로 출력한다.

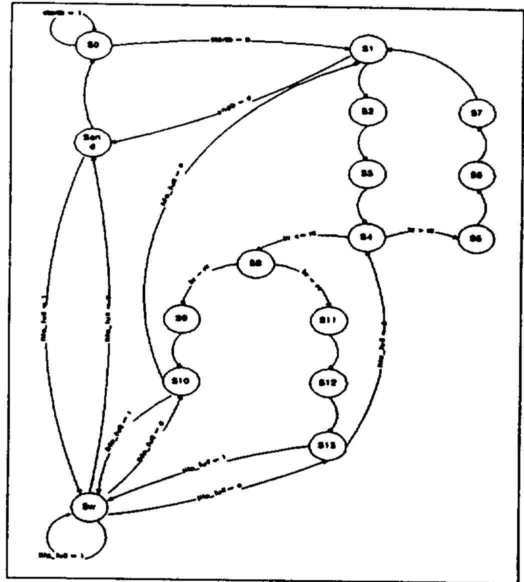
ORLC와 LCBS는 RC를 계산하기 위해 huff_sm 블



(그림 8) 하드웨어 블록도
(Fig. 8) Hardware block diagram

록에서 사용되고 동시에 code_gen 블록은 입력된 기호에 대한 실제 부호화를 수행하여, 그 부호어를 TR에 저장한다. TR에 저장된 부호어를 8-bit의 RR에 저장하기 위해, sbs 블록은 huff_sm에서 계산된 RC, LRR, 그리고 SRR을 사용한다. 이 블록에서는 내부 멀티플렉서에 의해 (그림 7)의 TR Shift에서 RC와 LRR 중 하나를 선택한다. 또한 (그림 7)의 순서도에서 ②로 분기하기 하는 경우에 TR에 저장된 부호어 중에서 RR로 전송되지 않고 남아있는 부호어를 다시 RR로 전송하는 기능을 수행한다. Fifo 블록은 RR에 저장된 8-bit 부호어를 외부로 출력하는 기능을 수행한다. 만약, FIFO_FULL 신호가 '1'이면, RR의 내용을 fifo 블록으로 전송하기 않고 이 신호가 '0'이 될 때까지 대기한다. 외부 신호 ENDB는 더 이상의 입력 기호가 없을 때, '0'으로 되고 이 신호에 의해 모든 부호화 동작이 정지되며, fifo 블록으로 전송되지 않고 RR에 남아있는 부호어를 전송한다.

(그림 8)의 각 기능 블록들을 위한 제어 신호를 생성하는 huff_sm 블록은 내부에 뺄셈기를 가지고 있어서 RC, LRR, SRR의 값을 변경시키는 상태 기계이다. 이 블록에 대한 상태도가 (그림 9)에 나타나 있다. (그림 8)의 RSTB 신호가 '0'일 때, LRR과 SRR을 각각 8과 7로 설정하고 (그림 9)의 S0 상태가 된다. 이 상태에서는 STARTB 신호가 '0'이 될 때까지 대기한다. 상태 S1에서는 ENDB 신호를 검사해서, 만약 이 신호가 '0'이면, Send 상태로 제어를 옮긴다. 상태 Send에서 FIFO_FULL 신호가 '1'이면 Sw 상태로 옮겨서 이 신호가 '0'이 될 때까지 대기한다. 그리고 Send 상태에서는 LRR이 8이 아니면 RR에 저장된 부호어를 fifo 블록에 전송한다. ENNB 신호가 '0'이 아니면, 외부 버스로부터 원시 심볼을 입력받고 S2로 간다. S2 상태에서는 메모리 읽기를 수행한다. 그리고 S3 상태에서는 ORLC와 LCBS의 값에 따라 RC를 계산하고 부호화를 수행한다.



(그림 9) 허프만 부호화기의 상태도
(Fig. 9) State diagram of the huffman encoder

상태 S4에서는 LRR과 RC를 비교하여, LRR이 더 크면 S5 상태로 가며, 그렇지 않으면, S8 상태로 간다. 상태 S5는 TR에 저장된 부호어를 RR로 전송하고 S6 상태로 간다. 상태 S6과 S7은 각각 LRR과 SRR을 변경시키고 S1 상태로 제어를 옮겨서 새로운 원시 심볼을 입력받는다. 상태 S8에서 LRR과 RC가 같으면, S9 상태로 제어를 옮기고 만약에, LRR과 RC가 같지 않으면, S11 상태로 제어를 옮긴다. 상태 S9는 S5 상태와 같은 동작을 수해하며 S10 상태로 제어를 옮긴다. 상태 S10은 LRR과 SRR을 각각 8과 7로 설정하고 FIFO_FULL 신호를 검사해서 '0'이면, RR에 저장된 부호어를 fifo 블록에 출력하고 S1 상태로 제어를 옮겨서 새로운 원시 심볼을 입력받는다. 상태 S11도 S5 상태와 같은 동작을 수행하며 S12 상태로 제어를 옮긴다. 상태 S12에서는 TR에 저장된 부호어가

RR에 모두 전송되지 않고 남아 있기 때문에, RC를 변경시키고 S13 상태로 제어를 옮긴다. 상태 S13은 LRR과 SRR을 각각 8과 7로 설정하고 FIFO_FULL 신호가 '0'이면, RR에 저장된 부호어를 fifo 블록에 출력하고 상태 S4(순서도에서는 ②로 분기하는 경우)로 제어를 옮긴다. 상태 Sw는 FIFO_FULL 신호가 '1'일 때, 이 신호가 '0'이 될 때까지 RR를 fifo 블록에 출력하지 않고 기다리는 대기 상태이다.

4.3 하드웨어 구현

본 논문의 하드웨어 구현을 위해서 256개의 심볼을 가지고 있는 8-bit ASCII 문자에 대한 허프만 테이블과 수정 허프만 테이블, 그리고 메모리 사상을 <표 5, 6, 7>에 각각 보인다.

<표 5>에 보인 허프만 테이블은 압축 프로그램들을 비교하기 위해 표준 Calgary Text Compression Corpus (CTCC)에서 제공한 파일들에 대해 작성하였다. 이러한 파일들은 ascii, binary, image 등의 몇 가지 종류를 포함한다. 그리고 <표 5>을 본 논문에서 제안한 테이블 구성 방법을 사용하여 작성한 테이블이 <표 6>이고 이것을 메모리에 사상시킨 것이 <표 7>이다. 필요한 메모리 크기는 연속된 '1'의 수를 표시하는데 4 bit, 나머지 비트열과 길이를 추출하기위해 필요한 추가 비트를 저장하는데 8 bit, 그러므로 총 256×(4+8) =3,072 bits가 된다.

<표 5> 표준 CTCC의 파일에 대한 허프만 테이블
<Table 5> Huffman Table for the file of the Standard CTCC

기호	부호어	길이
0	000	3
1	110111100	9
2	110111101	9
3	1110111100	10
⋮	⋮	⋮
245	111111111111111	15
⋮	⋮	⋮
252	111011100	9
253	1111100000	10
254	111011101	9
255	1100100	7

<표 6> (표 5)의 수정 허프만 테이블
<Table 6> Modified Huffman Table of the <Table 5>

연속된 1의 수	나머지 비트열	길이
0	000	3
2	0111100	7
2	0111101	7
3	0111100	7
⋮	⋮	⋮
15	–	0
⋮	⋮	⋮
3	011100	6
5	00000	5
3	011101	6
2	00100	5

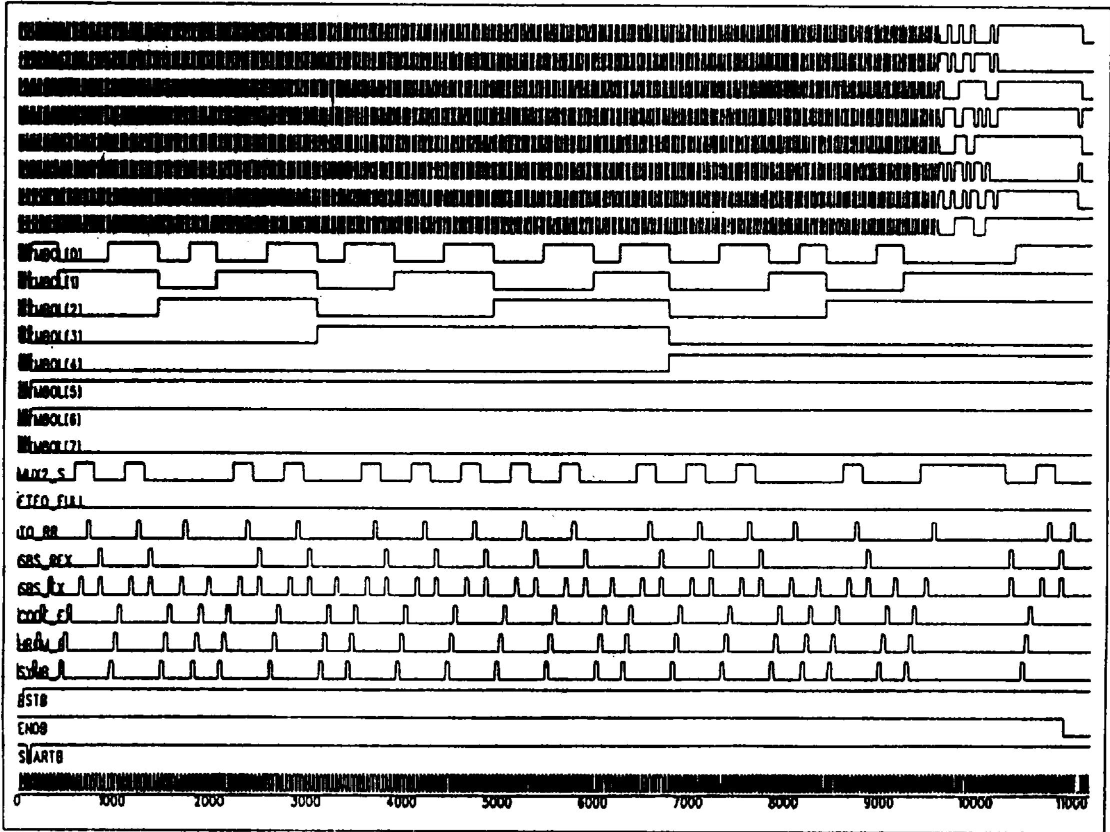
0 1 2 3 ⋮ 245 ⋮ 252 253 254 255

<표 7> <표 6>의 메모리 사상
<Table 7> Memory Mapping of the <Table6>

0000	00001000
0010	10111100
0010	10111101
0011	10111101
⋮	⋮
1111	00000001
⋮	⋮
0011	01011100
0101	00100000
0011	01011101
0010	00100100

0 1 2 3 ⋮ 245 ⋮ 252 253 254 255

<표 7>의 메모리 사상에 기초하여, 본 논문에서 제안한 부호화 알고리즘을 0.8 미크론의 반도체 공정에서 VHSIC(Very High Speed Integrated Circuit) HDL (Hardware Description Language)을 사용하여 구현하였으며 SUN 워크스테이션에서 COMPASS 툴을 사



(그림 10) 시뮬레이션 결과
(Fig. 10) Result of simulation

용하여 컴파일한 후의 회로 합성(Synthesis) 결과, 약 5,000개의 게이트가 소요되었다.

본 논문의 하드웨어는 25 MHz의 클럭에서 시뮬레이션되었으며, 3 클럭 사이클의 부호화 시간은 25 MHz 3 클럭/심볼 8 bits/심볼 67 Mbits/s의 압축률을 보였다.

(그림 10)에 시뮬레이션 결과를 보인다.

4.4 성능 평가

본 논문에서 제안한 하드웨어는 수정 허프만 테이블을 메모리에 사상시킴으로써 구현되었다. 이 하드웨어의 메모리 요구는 8-bit ASCII 문자에 대해 256 ×12-bit = 3,072 bits 크기이다. 이러한 메모리 요구는 [5]의 256×9-bit +64×18-bit=3,456 bits보다 더 적은 메모리가 필요하며, [4]의 512×12-bit=6,144 bits의 1/2에 불과하다는 것을 알 수 있다. 또한 부호화 시간은 본 논문에서 제안한 하드웨어에서는 3 클럭 사이클이 소요되지만, [4]와 [5]에서 제안된 하드웨어에서는 각각 7 클럭 사이클과 8 클럭 사이클이 소요되었다. 각각의 입력 기호에 대한 메모리 접근 수는 본 논문의 경우에 한 번만 발생하지 만, [4]와 [5]에서는 각각 평균 3, 4번 발생한다. 이를 〈표 8〉에 보인다.

〈표 8〉 성능 비교
〈Table 8〉 Performance Comparison(CC : Clock cycle per 8-bit symbol)

	Memory Simple	Algorithm	Memory Access	Encoding Time
This Paper	3,584 bit	Simple	1	3 CC
MALVLE [4]	6,144 bit	Complex	3	7 CC
HC Park[5]	3,456 bit	Medium	4	8 CC

5. 결 론

본 논문에서는 기존의 허프만 테이블을 정형 트리의 특징을 이용하여 수정하였으며, 이렇게 수정된 허프만 테이블에 기초하여 허프만 부호화 알고리즘을 제안하였다. 또한 수정 허프만 테이블을 메모리에 사상시키는 방법을 제안하였다. 그리고 8-bit ASCII 문자에 대해서, 허프만 테이블을 메모리에 사상시켰을 때, 그 크기는 하나의 ASCII 문자를 위해 12-bits의 메모리 워드가 필요하므로 256×12 bits=3,072 bits가 된다. 이것은 메모리 사상 기법을 사용하는 기존의 방법인 [5]와 비교했을 때, 거의 유사한 크기의 메모리가 필요하며, [4]의 방법의 1/2배 작은 메모리가 필요하다는 것을 알 수 있다. 그리고 본 논문의 알고리즘에서는 부호화에 3 클럭 사이클이 소요되므로 기존 연구의 7~8 클럭 사이클에 비하여 고속의 부호화가 가능하다.

그리고 제안한 허프만 부호화기를 구현하는 데에 0.8 미크론의 반도체 공정에서 약 5,000개의 게이트가 사용되었으며, 각각의 입력 기호를 부호화하기 위해 3 클럭 사이클이 소요되므로 25 MHz의 클럭에서 67 Mbps의 압축율을 나타내었다. 또한 허프만 테이블의 구성이 기존의 연구보다는 매우 간단하며, 입력 기호 수에 비례하여 하드웨어의 복잡도가 증가하는 종래의 방법과는 달리 하드웨어 복잡도는 입력 기호의 수에 비례하지 않는 특성을 가지고 있다.

본 논문에서 제안한 허프만 테이블의 재구성과 알고리즘은 문서 데이터 압축과 영상 데이터 압축, 모두에 적용될 수 있는 범용성을 가지고 있다.

참 고 문 헌

[1] D. Huffman, "A method for the construction of minimum redundancy codes", Proc. IRE, vol. 40, pp. 1098-1101, 1952.

[2] A. Mukherjee, N. Ranganathan, and M. Bassiouni, "Efficient VLSI designs for data transformations of tree-based codes", IEEE Trans. Circuits and Syst., vol. 38, pp. 306-314, 1991.

[3] K. K. Parhi, "High-Speed VLSI Architectures for Huffman and Viterbi Decoders", IEEE Trans. Circuits and Syst.-Ⅱ : Analogy and Digital Signal Processing, vol. 1, pp. 385-391, 1992.

[4] A. Mukherjee, J. W. Flieder, and N. Ranganathan, "MARVLE: A VLSI chip for data compression using tree-based codes", IEEE Trans. VLSI Syst. vol. 1, pp. 203-214, 1993.

[5] Heonchul Park, Viktor K. Prasanna, "Area Efficient VLSI Architectures for Huffamn Coding", IEEE Trans. Circuits and Syst., Vol. 40, pp. 568-575, 1993.

[6] Liang-Ying Lie et al, "CAM-Based VLSI Architecture For Dynamic Huffman Coding", IEEE Trans. Consumer Electronics, Vol. 40, No. 3, pp. 282-289, 1994.

[7] R. Hashemian, "Design and Hardware Implementation of A Memory efficient Huffman Decoding", IEEE Consumer Elec. Vol. 40, pp. 345-352, 1994.

[8] Tomoyuki Kawamura, Yoshikazu Eguchi, and Tetsuji Shigemura, "Compression of Huffman Code Table and Applications", 情報處理學會論文誌, Vol. 35, pp. 267-271, 1994 (in Japanese)

[9] Kwangsoo Seo, Seunghyun Nam, and Moonkey Lee, "VLSI Architecture for Programmable Variable Length Decoder", KITE Journal of Elect. Engineering, Vol. 6, pp. 21-25, 1995.

[10] 박지환 역, *문서 데이터 압축 알고리즘 입문*, 1st Ed. 성안당, pp. 41-64, 1995.

[11] R. Hashemian, "Memory Efficient and High-Speed Search Huffman Coding", IEEE Trans. Comm. vol. 43, pp. 2576-2581, 1995.

[12] Seungbae Choi, Moonho Lee, "High Speed Pattern Matching for A Fast Huffman Decoder", IEEE Trans. Consumer Elec. vol. 41, pp. 97-103, 1995.

박 화 식

1995년 2월 부산수산대학교 전자계산학과 졸업

1995년 3월~1997년 2월 부경대학교 전자계산학과 석사

1997년 3월~현재 아남 에스엔티 연구소 연구원

관심분야 : 전자계산기구조, ASIC 회로 설계

조 경 연

1990년 2월 인하대학교 공과대학 전자공학과 정보공학전공(공학박사)

1983년 3월~1991년 2월 삼보컴퓨터 기술연구소 책임연구원

1991년 3월~1995년 3월 부산수산대학교 자연과학대학 전자계산학과 조교수

1995년 4월~현재 부경대학교 공과대학 컴퓨터공학과 부교수

1991년 3월~현재 삼보컴퓨터 기술연구소 비상임 기술고문

1993년 6월~현재 아남 에스엔티 비상임 기술고문

1995년 1월~현재 대흥전자(주) 비상임 기술고문

관심분야 : 전자계산기구조, ASIC 회로 설계, ASIC memory