工學碩士 學位論文

DVB-S2 기반 멀티미디어 방송을 위한 고속 LDPC 부복호 알고리즘에 관한 연구

A Study on High Speed LDPC Encoder and Decoder Scheme for Multimedia Broadcasting based on DVB-S2 standard

指導敎授 鄭 智 元

2013年 2月

韓國海洋大學校 大學院

電 波 工 學 科

任 炳 洙

本 論文을 任炳洙의 工學碩士 學位論文으로 認准함.

委員長 : 工學博士 金 基 萬 (인)

委 員 : 工學博士 尹 榮 (인)

委 員 : 工學博士 鄭 智 元 (인)

2013年 02月

韓國海洋大學校 大學院

電 波 工 學 科

任 炳 洙

목 차

그 림 목 차

표 목 차

기호

dc : LDPC 복호에서 각 체크 노드와 연결된 엣지 수(row weight)

dv : LDPC 복호에서 각 비트 노드와 연결된 엣지 수(column weight)

u : 체크 노드에서 비트 노드로 연결된 엣지

v : 비트 노드에서 체크 노드로 연결된 엣지

N : LDPC 부호기의 코드워드 크기

K : LDPC 정보 비트의 크기

q : 체크 노드의 그룹 수

ABSTRACT

LDPC codes, proposed by Gallager in 1962, are adopted with a high-order modulation scheme as a powerful Forward Error Correcting(FEC) structure of DVB-S2. They have a remarkable Bit Error Rate(BER) performance that is close to the Shannon limit. An LDPC code is defined by a sparse parity check matrix H, where N bit nodes and (N-K) check nodes are connected by edges

LDPC codes can be decoded by implementing/using a message passing algorithm such as belief propagation, whereby messages transmitted iteratively between bit nodes and check nodes have probabilities corresponding to node values of 0 and 1. A message expressed in a LLR value contains both probabilities. The use of LLR values reduces the number of overall computations and minimizes the memory required for storing messages.

First, this thesis proposed high speed LDPC encoder architecture for DVB-S2 standard. The proposed LDPC encoding architecture is based on a parallel 360 bits-wise operations. The key issues for realizing high speed are using the two kinds of index addresses and make use of memories efficiently.

Second, horizontal shuffle scheduling(HSS) was studied in term of high speed decoding algorithm when the check node update at the same time to update the bit node. HSS can be accelerated to the decoding speed because of do not need to separate calculation of the bit nodes.

Finally, this thesis proposed efficient decoder architecture for high-speed decoding using the HSS method. In order to utilize memories efficiently, edge memories and bit-node memories are co-located in same memory.

제 1 장 서 론

서비스의 연속성이 보장되고 채널을 효율적으로 사용할 수 있는 초고화질 다채널 실감 방송 서비스를 전국 단위로 제공하기 위한 100 Mbps급 이상의 초고속 위성 방송 전송 기술 및 차세대 위성 방송 확보를 위해 기반 기술의 개발이 시급한 실정이다. 따라서 초고속 위성 방송 전송을 위해 고속 변복조 및 채널 부복호화의 고속화 등 전송기술 고도화를 위한 연구가 유럽에서 진행되고 있으며, 국내에서도 고속화에 대한 관심이 집중되고 있다. 일본 NHK는 디지털 위성방송의 전송 방식으로 강우에 따라 계층적 전송이 가능한 ISDB-S라는 위성방송 전송방식을 개발하여 사용하고 있으며, SHV전송을 위해 70 Mbps 이상의 전송속도를 갖는 LDPC 기반의 advanced ISDB-S2를 개발 중에 있다. 유럽의 경우 DVB-S2로 UHDTV 위성전송을 추진하고 있으나, 고속 대용량 전송의 어려움으로 최근 ASTRA를 중심으로 광대역 위성 방송 전송기술 표준화 움직임을 보이고 있어 기술적 대응이 시급하다. 특히 미국의 HNS(Hughes Network Systems)에서는 구현 시 문제점 및 큰 블록 크기에서의 LDPC 부호의 성능, 다양한 부호화율, 그리고 다양한 변조 방식에서의 성능을 검증하고 있으며 약 100 Mbps급의 FPGA 보드를 개발 중에 있다[1].

광대역 위성 방송에 적용될 수 있는 오류 정정 부호는 고속 데이터 전송에 효율적이고 성능이 우수한 복호기의 적용이 필수 불가결하며, 이는 DVB-S2에서 제시된 LDPC 부호화 방식이 초고화질 다채널 실감 방송 서비스를 전국 단위로 제공하기 위한 100 Mbps급 이상의 초고속 위성 방송 전송 기술로 적합하다. 유럽식 위성 방송 표준안인 DVB-S2에 적용되는 샤논의 채널 용량 한계에 근접한 LDPC 부호는 터보 부호에 비해 복호화의 복잡도가 낮을 뿐 아니라 좋은 거리 특성으로 오류마루 현상이 나타나지 않고, 완전 병렬 처리로 고속 처리가 가능한 장

점이 있다[2][3].

100 Mbps급 이상의 LDPC 부호를 설계하기 위해서는 부호화 관점에서는 높은 복잡도가 LDPC 부호의 중요한 문제점이었으나 최근에 삼각 행렬 분해법, Linear-congruence 방법을 사용하여 부호화기를 간단하게 하였다. 특히 DVB-S2기반 부호화 알고리즘은 높은 부호화기의 복잡도를 간단하게 구현하기 위해 검사행렬에서 “1”의 위치를 주소로 생성하여 주소를 이용하여 간단하게 구현되고 있다[3]. 그러나 부호화 설계 시 이전의 패리티 값을 알아야 다음의 패리티 값을 구할 수 있어 고속화를 위해 병렬처리가 가능한 알고리즘이 필요하다. 구현의 문제점 대부분은 복호부에 있으며, 이를 어떻게 100 Mbps급 이상의 전송률을 가지게 하는가에 있다. DVB-S2에서 제안한 알고리즘을 표준안으로 채택하고 있으며, 큰 블록 사이즈(N= 64800) 및 많은 반복 횟수를 요구하고 있다. 따라서 복호시에 병렬 처리, 병렬 처리에 따른 메모리 설계가 중요하다.

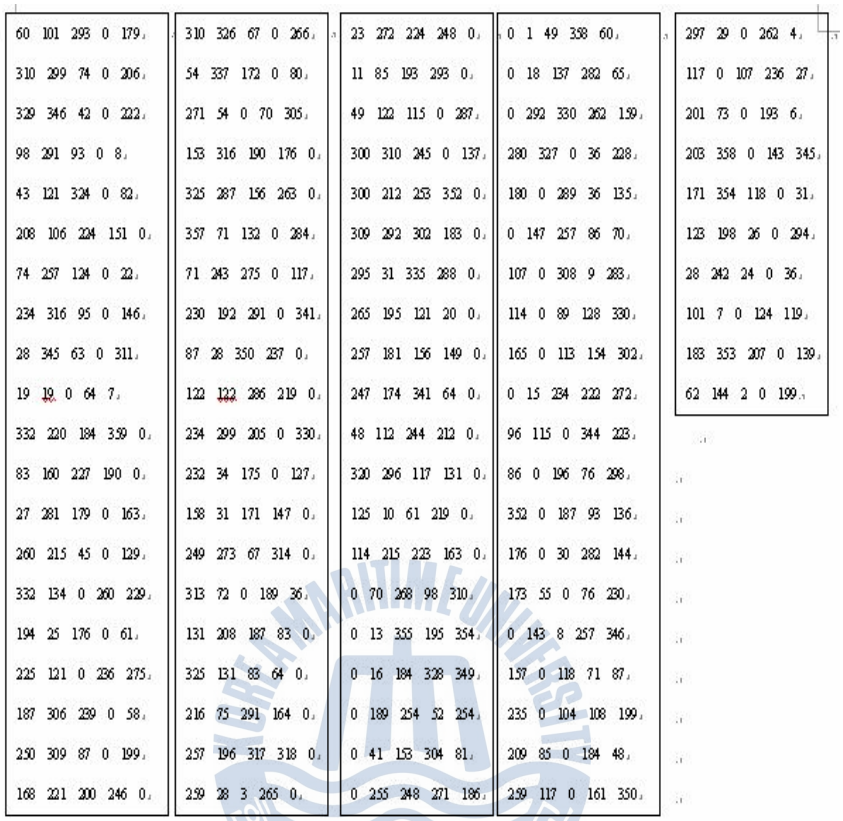
따라서 본 논문에서는 첫째로 고속 부호화 방식을 연구하였다. 기존의 직렬구조에서 360 개의 부분 병렬을 이용하여, 그리고 부호화 구조에서 메모리를 효율적으로 적용하여, 기본 주파수 클럭 100 MHz에서 약 10 Gbps급의 부호기 구조를 설계하였다. 둘째로 비트 노드 계산을 체크 노드 계산 후에 하지 않고 체크 노드 계산 중에 수행하는 HSS(Horizontal Shuffle Scheduling) 방식[4]을 기반으로 하는 복호기 구조에서 기존의 방식 보다 고속화 할 수 있는 방안을 연구 하였으며, 기존의 dc개의 직렬 구조에서 dc개의 병렬 구조로 메모리를 효율적으로 설계함으로써 고속화가 가능하게끔 하였다.

제 2 장 고속 LDPC 부호기 설계

LDPC 부호기는 코드워드 크기 $N=[k_0,k_1,\ldots k_k,p_o,p_1,\ldots,p_{n-k}]$에 정보 블록크기 $K=[k_0,k_1,\ldots k_k]$를 부호화하여 패리티를 생성한다. DVB-S2 규격 LDPC 부호화의 고속화를 위한 패리티를 생성하기 위해서는 R_index와 P_index가 필요하다. 그림 2.1은 부호화율이 1/2인 경우의 R_index와 P_index를 나타낸다.

26 27 30 36 74	11 19 31 56 74	15 20 70 75 76	6 12 21 38 57	2 13 26 64 80
10 10 20 37 41	0 8 11 57 67	17 31 57 58 77	7 12 31 61 70	16 27 34 72 76
4 16 27 38 51	19 31 58 71 85	16 20 27 78 86	8 12 12 35 65	0 9 28 34 55
2 6 16 39 68	3 9 19 45 59	14 21 59 79 89	1 1 9 29 34	7 27 29 33 62
9 24 33 40 82	4 17 35 52 60	8 19 25 39 80	3 10 18 23 33	11 18 28 30 69
3 13 24 30 41	19 24 27 61 78	1 5 18 22 81	11 13 17 47 72	9 10 29 31 33
14 21 32 42 76	2 29 52 62 83	7 21 33 71 82	7 12 15 24 81	8 23 31 32 32
4 28 34 43 89	7 13 34 63 66	0 26 50 65 83	9 13 30 34 49	11 25 33 42 84
16 18 30 44 87	18 25 32 45 64	0 10 14 35 84	6 14 31 32 43	5 7 15 34 56
21 24 45 58 79	5 6 14 35 65	0 5 29 33 85	15 22 26 79 84	0 19 32 35 43
1 4 5 37 46	20 22 47 66 86	1 11 12 60 86	1 13 16 20 32	
8 19 20 32 47	3 4 46 67 80	1 3 21 85 87	2 17 17 24 28	
15 26 33 48 78	6 17 25 64 68	5 17 73 87 88	2 18 34 39 53	
8 16 46 49 66	15 16 35 40 69	10 23 40 68 89	18 19 27 35 62	
0 18 50 63 67	11 12 70 77 88	0 3 6 8 83	9 9 20 22 75	
13 23 30 51 63	14 22 28 36 71	1 25 30 53 88	21 22 23 69 77	
6 11 52 59 73	5 21 25 51 72	2 6 28 30 42	14 22 26 29 54	
8 27 49 53 56	4 10 31 61 73	3 3 12 26 60	7 23 24 28 36	
14 23 25 54 81	7 10 37 50 74	4 4 23 29 35	13 15 24 41 44	
2 15 28 38 55	20 26 48 55 75	5 17 29 44 54	2 22 25 48 82	

(a) R_index



60 101 293 0 179	310 326 67 0 266	23 272 224 248 0	0 1 49 358 60	297 29 0 262 4
310 299 74 0 206	54 337 172 0 80	11 85 193 293 0	0 18 137 282 65	117 0 107 236 27
329 346 42 0 222	271 54 0 70 305	49 122 115 0 287	0 292 330 262 159	201 73 0 193 6
98 291 93 0 8	153 316 190 176 0	300 310 245 0 137	280 327 0 36 228	203 358 0 143 345
43 121 324 0 82	325 287 156 263 0	300 212 253 352 0	180 0 289 36 135	171 354 118 0 31
208 106 224 151 0	357 71 132 0 284	309 292 302 183 0	0 147 257 86 70	123 198 26 0 294
74 257 124 0 22	71 243 275 0 117	295 31 335 288 0	107 0 308 9 283	28 242 24 0 36
234 316 95 0 146	230 192 291 0 341	265 195 121 20 0	114 0 89 128 330	101 7 0 124 119
28 345 63 0 311	87 28 350 237 0	257 181 156 149 0	165 0 113 154 302	183 353 207 0 139
19 19 0 64 7	122 122 286 219 0	247 174 341 64 0	0 15 284 222 272	62 144 2 0 199
332 220 184 359 0	234 299 205 0 330	48 112 244 212 0	96 115 0 344 223	
83 160 227 190 0	232 34 175 0 127	320 296 117 131 0	86 0 196 76 298	
27 281 179 0 163	158 31 171 147 0	125 10 61 219 0	352 0 187 93 136	
260 215 45 0 129	249 273 67 314 0	114 215 223 163 0	176 0 30 282 144	
332 134 0 260 229	313 72 0 189 36	0 70 268 98 310	173 55 0 76 280	
194 25 176 0 61	131 208 187 83 0	0 13 355 195 354	0 143 8 257 346	
225 121 0 236 275	325 131 83 64 0	0 16 184 328 349	157 0 118 71 87	
187 306 289 0 58	216 75 291 164 0	0 189 254 52 254	235 0 104 108 199	
250 309 87 0 199	257 196 317 318 0	0 41 153 304 81	209 85 0 184 48	
168 221 200 246 0	259 28 3 265 0	0 255 248 271 186	259 117 0 161 350	

(b) P_index

그림 2.1 부호화율 1/2에서 패리티 계산을 위한 R_index, P_index

그림 2.1 에서 보인 R_index와 P_index를 이용한 패리티 생성식은 다음과 같은 절차로 구해진다.

K 의 정보 비트에 의해 $N-K$ 의 패리티 비트가 다음 식과 같이 생성된다. 여기에서 위의 R_index와 P_index를 사용하기 위해 정보 비트 $K=[(\mathrm{K}_{0,359}),(\mathrm{K}_{1,359}),\ldots,(\mathrm{K}_{q,359})]$ 로 순서대로 360개씩 그룹화한다.

$$\sum_{i=0}^{359}\sum_{j=0}^{(dc-2)-1}\mathrm{K}_{R\_index[0,j],(P\_index[0,j]+i)\%360}=\mathrm{P}_{0,i}\ \text{(0-th row)} \qquad (2.1)$$

$$\sum_{i=0}^{359}\sum_{j=0}^{(dc-2)-1} K_{R\_index[x,j],(P\_index[x,j]+i)\%360} \oplus P_{x-1,i} = P_{x,i} \quad (x \neq 0 \text{ 인 row}) \qquad (2.2)$$

위의 식(2.1)과 식(2.2)에서 dc는 row_weight의 수를 나타내고 $R\_index[x,j]$ 는 R_index 의 x 번째 row에서 j 번째 값을 나타내고, $P_index[x,j]$는 P_index 의 x 번째 row에서 j 번째 값을 나타낸다. 위의 식처럼 패리티를 계산하게 되면 각각의 row에 포함되는 360개의 패리티가 동시에 생성된다. 그 이유는 DVB-S2 기반 LDPC는 H 매트릭스 상에서 q 의 간격으로 360개씩 그룹을 이루기 때문이다. 이렇게 계산된 패리티는 360개씩 메모리의 첫번째 address 부터 차례대로 저장된다. Step 1의 전체적인 구조는 다음 그림 2.2와 같다.

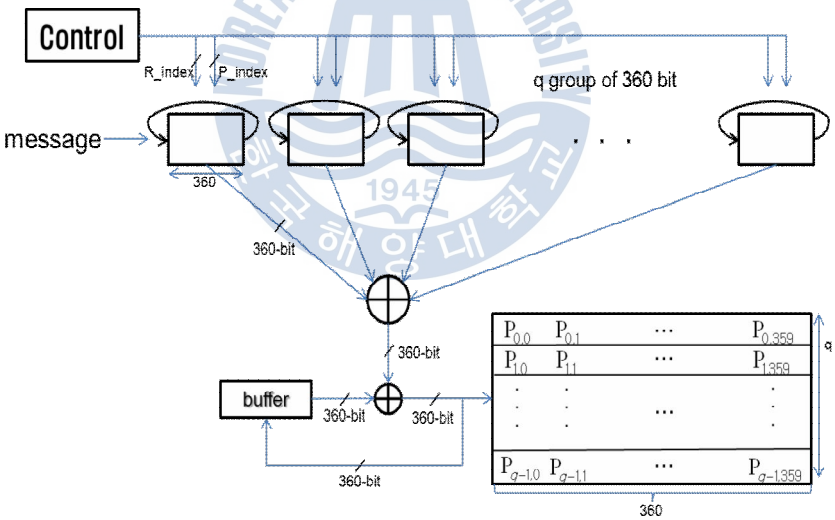


그림 2.2 고속 부호화기 구조

그림 2.2를 바탕으로 다음과 같은 세 가지 step으로 나눌 수 있다.

Step 1 : 인코더에서 출력되는 패리티 비트는 360개씩 개의 address가 그림 2.2의 패리티 메모리에 저장된다. 패리티는 이전의 패리티에 따라

서 다음 패리티를 식 (2.3)에서 구할 수 있다.

$$P_i = P_i \oplus P_{i-1} \tag{2.3}$$

즉, 이전의 패리티를 알고 있어야 다음 패리티를 구할 수 있다는 것이다. 하지만 식 (2.3)과 같이 패리티를 구하게 되면 부호화율이 1/2인 경우에는 패리티의 길이가 32400이기 때문에 이에 소모되는 클럭은 32400의 클럭이 소요되게 되어 고속 부호화를 할 수 없게 된다.

식 (2.1), 식 (2.2)와 같이 패리티를 생성하게 되면 첫번째 address에서 $P_0 \sim P_{q-1}$ 를 제외한 $P_q \sim P_{N-K}$ 는 식 (2.3)처럼 이전의 패리티를 고려하지 않은 상태로 패리티 연산을 하였기 때문에 $P_0 \sim P_{q-1}$ 를 제외한 모든 패리티는 이전의 패리티를 고려하여 다시 계산을 해줘야 한다. 하지만 패리티를 생성하는 연산 자체는 비트연산의 ex-or 연산이므로 예를 들어, P_{89} 가 '1'일 경우 $P_{90} \sim P_{179}$ 까지의 패리티 비트는 모두 반전이 된다. 마찬가지로 모든 연산이 끝난 P_{179} 의 값이 '1'일 경우 $P_{180} \sim P_{269}$ 까지의 패리티 비트는 모두 반전이 된다. 이러한 성질을 이용하면 메모리의 마지막 address에서 읽어 온 360개의 패리티 비트들은 식 (2.4)와 같이 계산되어진다.

$$\text{Step 2 : } \mathrm{P}_{q-1,k} = \mathrm{P}_{q-1,k} \oplus \mathrm{P}_{q-1,k-1} \ (k = 1 \sim 359) \tag{2.4}$$

즉, P_{89} 를 이용하여 P_{89} 가 '1'이라면 P_{179} 의 값은 반전, 연산된 P_{179} 의 값이 '1'이라면 P_{269} 의 값은 반전되는 방식으로 총 359번의 연산으로 마지막 address의 패리티 비트들을 완벽히 연산할 수 있다.

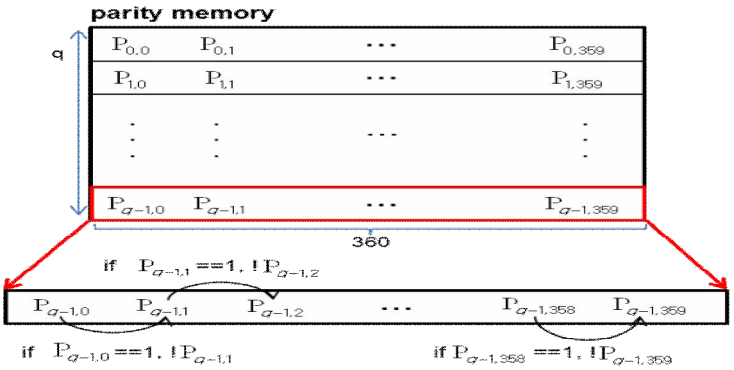


그림 2.3 단계 2의 과정에 대한 연산과정

Step 3 : $P_{i,j} = P_{i,j} \oplus P_{q-1,j} \quad i = 0 \sim q-2, j = 1 \sim 359$ (2.5)

마지막 패리티 비트들에 대한 연산이 끝나면 다시 메모리의 첫번째 address의 패리티 비트들부터 읽으면서 각각 359개의 패리티 비트들을 반전 혹은 비 반전을 하여 모든 패리티 비트들을 완벽히 연산할 수 있다.

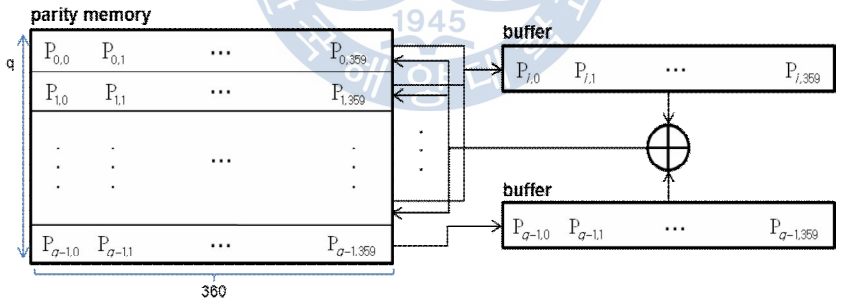


그림 2.4 단계 3의 과정에 대한 연산과정

결과적으로, 모든 패리티를 생성하는데 3q+359+α의 클럭만을 사용하여 부호화 할 수 있다. 이 때, α는 각종 연산과 메모리 읽기/쓰기에 사용되는 클럭이다. 위의 알고리즘을 토대로 기본 주파수 클럭 100MHz에서 약 10Gbps급의 부호기 구조를 설계할 수 있다.

제 3장 효율적인 고속 LDPC 복호기

제 3-1 절 기존 복호 알고리즘

다음 그림 3.1은 기존 LDPC의 부호화율 2/3에서 복호기 구현을 위한 전체 블록도를 나타낸다.

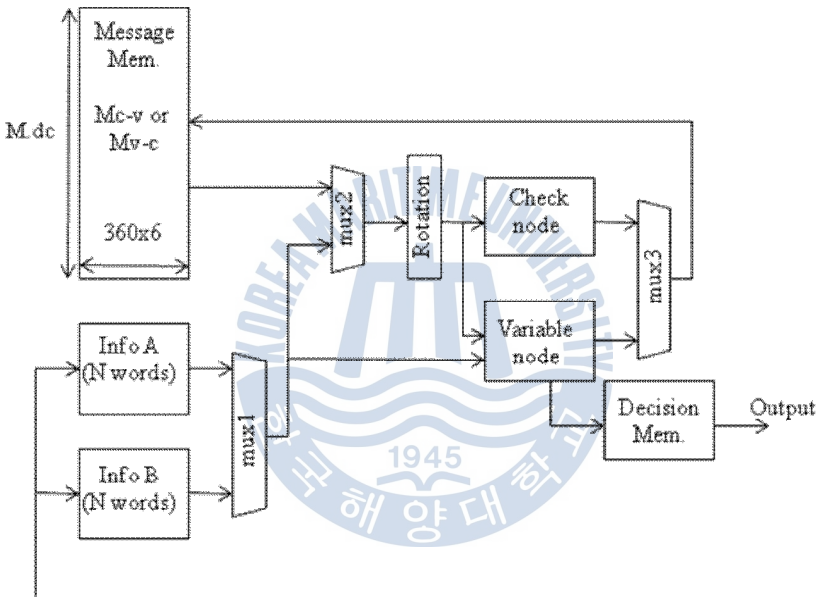


그림 3.1 복호기 구현을 위한 LDPC의 전체 블록도

위의 블록도의 흐름을 보면 Info A, Info B 메모리를 통해 수신된 비트가 Rotation을 거쳐 체크 노드 계산이 되어지고 체크 노드 계산이 끝나면 Message Mem.에 저장이 된다. 그리고 Rotation을 거쳐 비트 노드 계산이 되어지고 비트 노드 계산이 끝나면 비트는 다시 Message Mem.에 저장이 된다. 이렇게 되면 한번의 iteration이 끝나게 된다. 이런 과정이 iteration 횟수만큼 반복되어지고 마지막 iteration이 끝났을 때 Decision Mem.을 통해 최종 수신 비트가 출력되게 된다. Info 메모리를 보면 Info

A와 Info B라는 2개의 메모리로 되어있다. 이는 수신된 비트를 Info A 메모리와 Info B 메모리를 통해 번갈아 가면서 수신 받음으로써 데이터를 지연 없이 바로 처리할 수 있게 하는 것이다. 처음 iteration에서 Info A 메모리를 통해 체크 노드와 비트 노드가 계산되어지는 동안 Info B 메모리에서는 다음 N개의 비트를 받게 된다. 그래서 Info A 메모리를 통한 체크 노드와 비트 노드 계산이 끝나게 되면 Info B 메모리에 저장되어있던 수신 비트가 체크 노드와 비트 노드 계산 과정을 거치게 되고 Info A 메모리에서는 다음 N개의 비트를 수신 받게 된다.

다음 표 3.1은 부호화율 2/3에서 구현을 위한 메시지 메모리에 필요한 parameter를 나타낸다.

표 3.1 구현을 위한 메시지 메모리

	Size	Word length	Total RAM	Total ROM
Message memory	600	2,160	1,296,000	
Intrinsic information A	180	1,800	324,000	
Intrinsic information B	180	1,800	324,000	
Decision	180	360	64,800	
RAM Forward	8	2,160	17,280	
FIFO Variable node	12	2,160		
ROM_{CN}	600	8		4,800
ROM_{VN}	600	6		3,600
ROM_{p}	600	9		5,400
TOTAL			2,026,080	13,800

DVB-S2 규격의 부호화율 2/3에서 복호를 위한 Row_weight 수는 10, Col_weight 수는 13이다. 위의 표를 보면 알 수 있듯이 구현을 위한 총 edge의 size는 600이고, 체크 노드 저장을 위해 필요한 address의 총 size는 600, 비트 노드의 저장을 위해 필요한 address의 총 size는

600이다. 또한 permutation을 위해 필요한 address의 총 size도 600이다. 체크 노드에 사용되는 총 ROM은 4800, 비트 노드에 사용되는 총 ROM은 3600 그리고 permutation에 사용되는 총 ROM은 5400이다. 따라서 구현을 위한 ROM의 총 수는 13800이고 RAM의 총 수는 2026080이다.

표 3.2 구현을 위한 DVB-S2 LDPC의 parameter

Code Parameters	Size of LDPC	64800
	Code rate	0.667
	Degree of check dc	10
	Max degree of variable dv	13
Architecture Parameters	Level of parallelism P	360
	Clock frequency(MHz)	200
	Latency of rotation	3
	Max(or average) iteration	30
	Bit size of message	6
	Bit size of intrinsic	5
Variables	Number of group check M	60
	Number of group variable N	180
	Number of group edges E	600
	N_b cycle check node half iteration	613
	N_b cycle variable node half iteration	615
	Total iteration	1,228
	Total nit iteration	36,840
	Decoding throughput	352

표 3.2는 DVB-S2 규격 부호화율 2/3, N size 64800에서 구현을 위한 파라미터를 나타낸 것이다. 구조적으로 볼 때 P는 360이고 클럭 주파수는 200MHz이다. Rotation에서의 지연 시간은 3clock, 최대(또는 평균) 반복은 30회이고 메시지의 비트는 6 bit, intrinsic의 비트는 5 bit이다. 그리고 체크노드 그룹은 60, 비트노드 그룹은 180, edge 그룹은 600개이고, 한번의 iteration에서 체크노드 연산에 필요한 사이클은 613, 비트노드 연산에 필요한 사이클은 615이다. 따라서 한번의 iteration에서 총 1228 사이

클이 필요하고 최대 iteration이 30회 이기 때문에 DVB-S2 규격 부호화율 2/3에서 필요한 총 사이클은 36840이 되게 된다. 이렇게 구현한 복호 throughput은 352 MHz가 된다.

다음은 위의 구조의 이해를 돕기 위해, 그림 3.1에서 제시된 edge 메모리와 bit node 계산을 위한 index가 저장되어 있는 ROM 의 구조를 나타낸다.

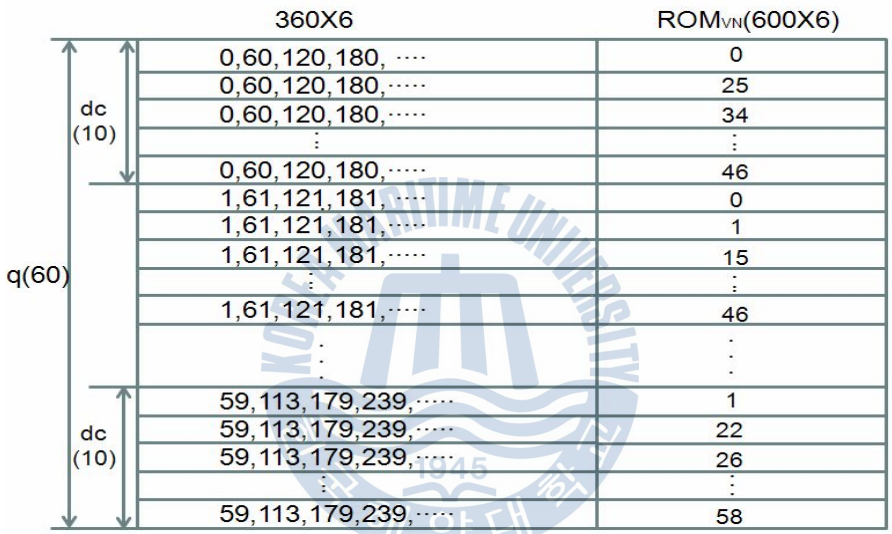
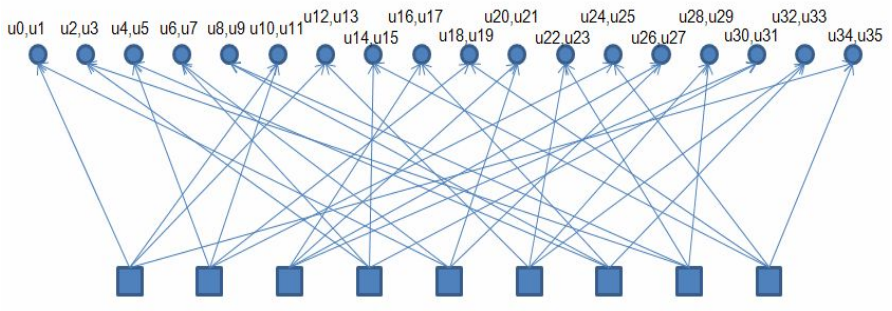


그림 3.2 구현을 위한 메모리 구조

DVB-S2 규격의 부호화율 2/3에서는 Row_weight 수(dc)는 10이고 Col_weight 수(dv)는 13 그리고 q는 60이다. 그림의 왼쪽 부분을 보게 되면 메모리를 dc개를 한 그룹으로 하여 q개의 그룹으로 분할하였다. 그리고 그림의 오른쪽 부분은 비트노드 계산을 위한 index를 나타내고 있다. 즉, 어느 그룹의 몇 번째 메모리의 데이터를 가져와서 비트노드 계산을 할지를 나타내고 있는 것이다. 아래의 예제는 구현을 위한 메모리 구조의 예시를 나타낸다.



(a) Bipartite graph

Message MEM.
u0,u2,u5
u10,u6,u8
u12,u14,u17
u34,u31,u33
u4,u1,u3
u11,u7,u9
u18,u20,u23
u24,u27,u29
u16,u13,u15
u20,u22,u19
u26,u28,u25
u30,u32,u35

(b) Edge 메모리

ROM_{VN}
0
1
0
1
0
2
1
2
1
2
0
2

(c) 비트노드 계산을 위한 index

그림 3.3 구현을 위한 메모리 구조의 예

위의 그림은 Row_weight 4, Col_weight 2 그리고 q가 3일 때의 메모리 구조를 예를 들었다. 그림 3.3(a)의 Bipartite graph를 보고 edge 메모리를 그림 3.3(b)와 같이 구성한 다음 그림 3.3(c)의 index를 이용해 비트노드 계산을 한다. Row_weight가 4이기 때문에 edge 메모리를 위에서부터 4개씩 한 그룹으로 묶어서 순서대로 0번째 그룹, 1번째 그룹, 2번째 그룹이라 한다. 그리고 그림 3.3(c)의 index에서 0, 1, 2가 4번씩 나오게 되는데 첫 번째 0이 나오면 0번째 그룹의 첫 번째 edge u_0, u_2, u_5를 가지고 비트노드를 계산하고, 두 번째 0이 나오면 0번째 그룹의 두 번째 edge

u_{10}, u_6, u_8을 가지고 비트노드 계산을 하는 것이다. 이런 방식으로 나머지 비트노드들도 위에서 설명한 방식과 같이 edge 메모리의 edge를 이용하여 비트노드를 계산하게 된다.

제 3-2 절 Horizontal Shuffle Scheduling 알고리즘

기존의 LDPC 복호기의 구조는 다음 그림 3.4와 같다.

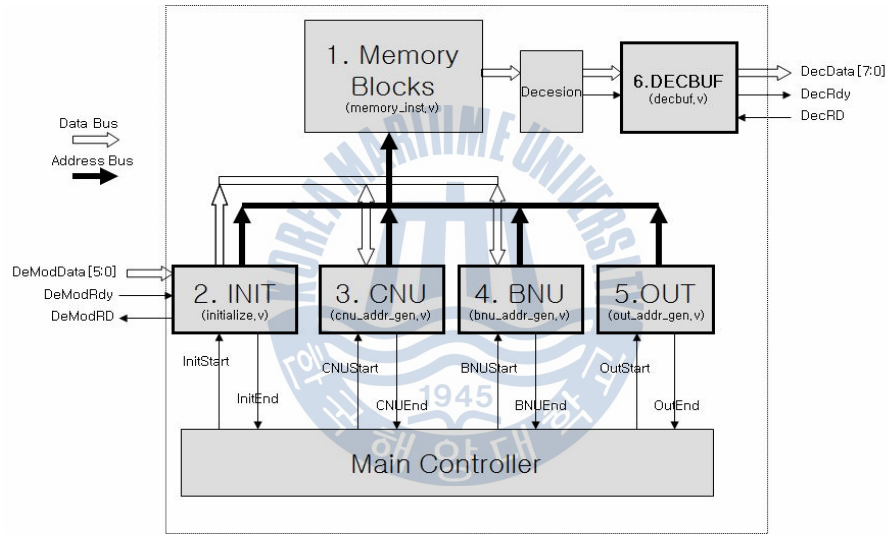


그림 3.4 기존의 LDPC 복호기의 구조

기존의 LDPC 복호기의 복호 순서는 우선 수신데이터를 이용하여 비트 노드를 초기화 한 후 각각의 체크 노드에 연결된 비트 노드 값을 이용하여 체크 노드 업데이트를 한다. 체크 노드 업데이트 후 다시 각각의 비트 노드에 대해 업데이트를 하고, 이러한 연산을 계속 반복한다. 기존의 복호 방식에 의해 체크 노드 업데이트 연산이 모두 끝난 후 비트 노드 업데이트를 하기 때문에 한번의 반복에도 많은 지연이 발생하여 고속의 LDPC 복호를 할 수 없다.

이를 극복하기 위해 본 절에서는 Horizontal Shuffle Scheduling(HSS) 복호 방법을 연구하였다. HSS 방식은 기존의 방식과는 달리 체크 노드 업데이트 연산을 하면서 비트 노드 업데이트 연산을 동시에 하는 것이 가능하다. HSS 복호 방식의 흐름도는 다음 그림 3.5와 같다[4].

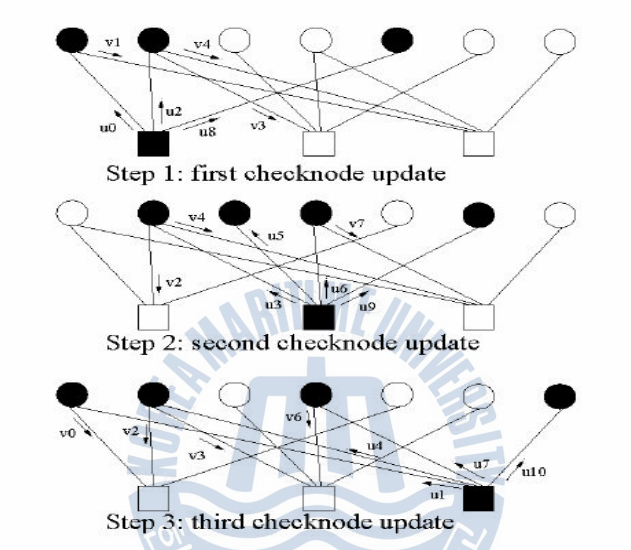


그림 3.5 HSS 복호 방식의 흐름도

LDPC 복호 과정은 반복에 의한 복호이다. 그러므로 적어도 각 노드들이 한번씩 업데이트가 되면 한번의 반복이 끝나는 것이다. 그리고 전체 복호 과정은 지정된 반복 횟수만큼 반복이 되거나 그렇지 않으면 충분히 신뢰성 있는 데이터를 구했을 때 끝나게 된다. 그 후, 비트 노드의 값을 이용하여 각 비트를 결정한다. HSS 방식을 이용한 LDPC 복호 과정 중 각 비트 노드의 값은 다음 식에 의해 구할 수 있다.

$$S_i = LLR_i + \sum_{i=1}^{dv} u_i \tag{3.1}$$

여기서 S_i 는 비트 노드의 최종 값을 나타내고, LLR 은 수신 데이터를 나타낸다. i 는 비트 노드이고, dv 는 비트 노드에 연결된 edge의 수이다. 그리고 u_i 는 체크 노드 업데이트를 통해 얻어진 각 edge의 값이다. 위 그림 3.5를 통해 간단한 예를 들어 보면, 첫 번째 체크 노드 업데이트를 하기 위해 v_0, v_2, v_8 각각의 edge 값을 가지고 체크 노드 업데이트를 한다. 이 때, 각 edge의 값은 수신 데이터 LLR_0, LLR_1, LLR_4 이고, 체크 노드 업데이트 방법은 다음 식으로 알 수 있다.

$$u_j = \oplus_{k=1,k\neq j}^{dc} v_k \tag{3.2}$$

v는 비트 노드에서 체크 노드로 향하는 edge를 나타내고, dc 는 체크 노드에 연결된 edge의 수이다. 위 식에서 $\oplus$는 다음과 같이 구할 수 있다.

$$|v_i| \oplus |v_j| = \min(|v_i|,|v_j|) - offset \tag{3.3}$$

$$sign(|v_i| \oplus |v_j|) = sign(v_i) \times sign(v_j) \tag{3.4}$$

체크 노드 업데이트의 값을 이용하여, 각 비트 노드의 값은 $S_0 = LLR_0 + u_0$, $S_1 = LLR_1 + u_2$, $S_4 = LLR_4 + u_8$ 이 된다. 그 후, 두 번째 체크 노드 업데이트를 위해 v_3, v_5, v_6, v_8 을 가지고 온다. v는 v' 를 이전 반복에서의 edge 값이라 하면 다음 식에 의해 구해질 수 있다.

$$v_i = S_i - v_i' \tag{3.5}$$

두 번째 체크 노드의 업데이트가 끝나면 각 edge의 값을 이용하여

다시 S_1, S_2, S_3, S_5가 구해지고, 세 번째 체크 노드에 대한 업데이트를 하여 S_0, S_1, S_3, S_7역시 업데이트 되어진다. 이와 같이 한 번의 반복이 끝나게 된다. 이러한 과정이 반복되면서 모든 반복이 끝나거나, 신뢰성 있는 데이터가 나올 때, 복호 과정은 끝나게 된다.

구현을 위한 HSS 방식의 구조는 다음 그림 3.6과 같다.

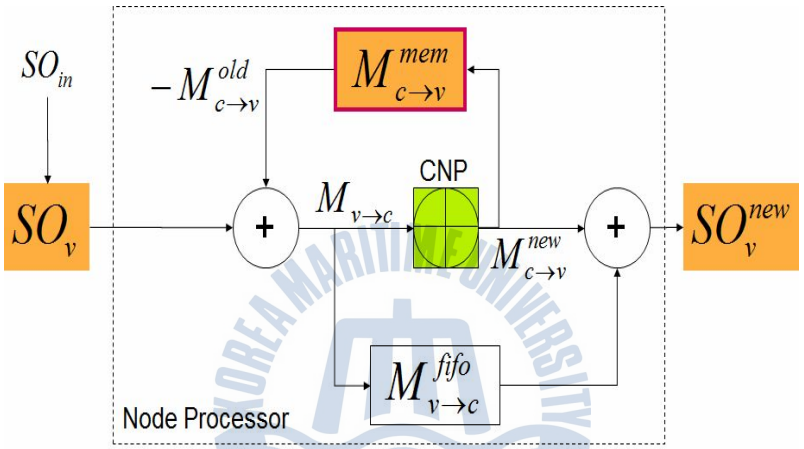


그림 3.6 HSS 방식의 노드 연산 구조

DVB-S2 규격의 LDPC 부호는 메트릭스 구조 상 부분화 시켜서 병렬 연산이 가능하다. 즉, H 메트릭스에서 부분화 되는 만큼 병렬로 연산하여 속도를 증가시킬 수 있다. 하지만, 이렇게 나누게 되면 HSS 방식을 적용하여 구현 할 경우 부분화 된 집합 내에서 한번에 두 개 이상의 비트 노드와 체크 노드가 연결이 되는 경우가 발생한다. 식 3.1 부터 식 3.5 까지를 보면 HSS 방식은 한번에 하나씩 비트 노드와 체크 노드 간에 업데이트를 실행함을 알 수 있다. 그래서 한번에 두 개 이상의 비트 노드와 체크 노드가 연결이 되면 각 데이터 간에 메모리 연결 충돌이 발생하여 HSS 연산에서 에러가 발생할 수 있다. 그림 3.7은 이러한 경우를 보여준다. 이 경우 이러한 메모리 연결 충돌을 방지하기 위하여

따로 두 개의 임시 프로세서를 사용할 수 있다. 간단한 예를 들어 한 번에 두 개의 비트 노드와 체크 노드가 연결되었다면, 그림 3.7에서 첫 번째 비트 노드를 S 라 하고, 이 비트 노드에서 체크 노드로 가는 세 개의 edge 값을 각각 v_0, v_1, v_2 체크 노드에서 비트 노드로 업데이트 되는 세 개의 edge 값을 각각 u_0, u_1, u_2 라 하자.

$$S' = LLR + u_0' + u_1' + u_2' \tag{3.6}$$

이라 할 때, 여기서 $u_0' + u_1' + u_2'$ 는 이전 반복에서의 edge 값이고, S' 는 이전 반복에서의 비트노드 값이다. 현재 반복에서의 u_2 를 구하기 위해 세 번째 연결된 체크 노드로 가는 edge v_2 를 구하게 위해서는 다음과 같은 식이 나와야 한다.

$$S = LLR + u_0 + u_1 + u_2' \tag{3.7}$$

위 식은 정상적인 상태에서 HSS 방식을 적용하였을 때 나올 수 있다. 하지만 그림 3.7에서처럼 두 개의 edge가 한 번에 계산 되어 질 때는 위와 같은 식이 나올 수 없다. 그래서 따로 두 개의 임시 프로세서를 사용한다면, 다음과 같이 구할 수 있다. 임시 프로세서 두 개를 각각 S_1, S_2 라 하면,

$$\begin{gathered} v_0 = S' - u_0', \quad v_1 = S' - u_1' \\ S_1 = LLR + u_0 + u_1' + u_2' \\ S_2 = LLR + u_0' + u_1 + u_2' \\ S = S_1 + S_2 - S' \end{gathered} \tag{3.8}$$

과 같이 구할 수 있다.

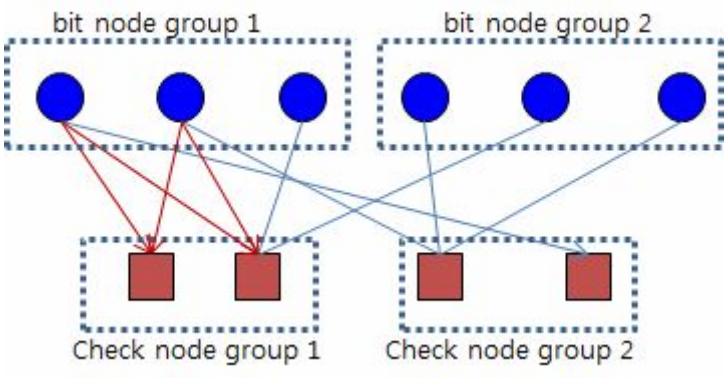


그림 3.7 메모리 연결 충돌의 예

HSS 방식은 한 번의 반복에서 비트 노드가 row weight 만큼 업데이트 되기 때문에 기존의 방식에 비해 좋은 성능을 가진다. 이는 기존의 알고리즘 보다 요구되는 반복횟수가 많이 줄어듦을 의미한다.

다음 그림 3.8부터 3.16까지는 기존의 알고리즘과 HSS 방식의 반복 횟수에 의한 성능을 비교한 것이다.

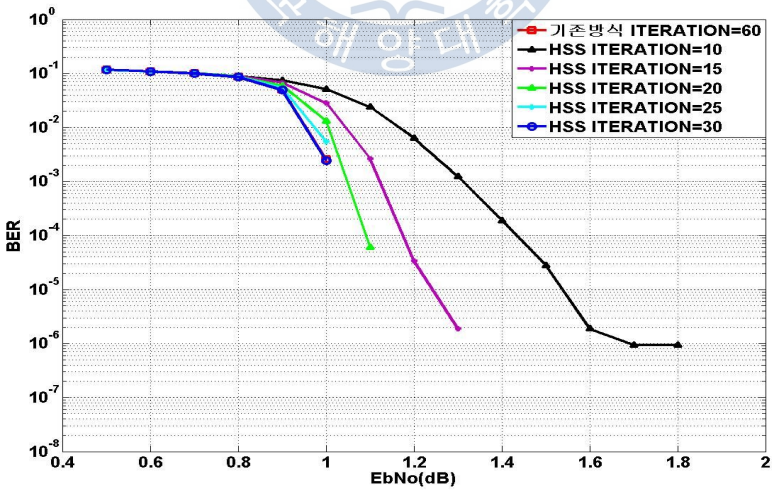


그림 3.8 기존방식과 HSS방식의 반복횟수에 따른 성능 비교(r=1/2)

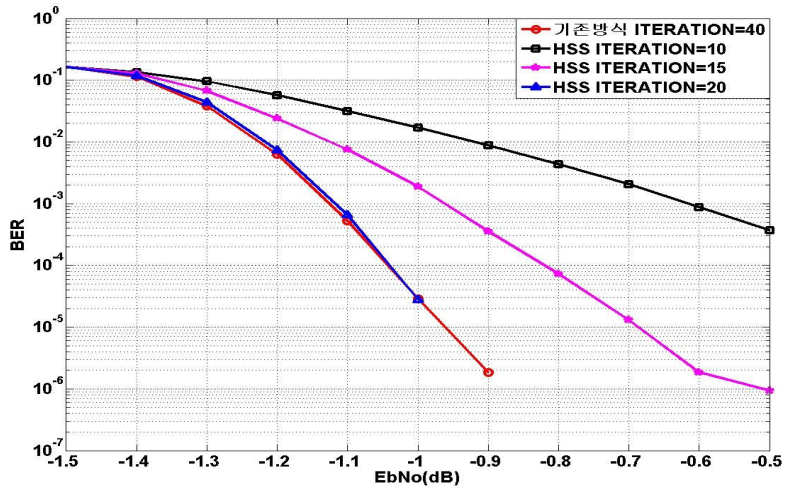


그림 3.9 기존방식과 HSS방식의 반복횟수에 따른 성능비교(r=1/3)

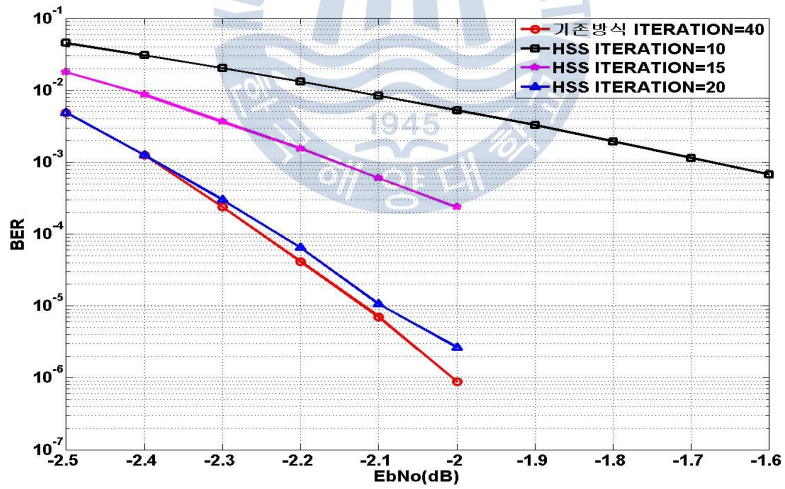


그림 3.10 기존방식과 HSS 방식의 반복횟수에 따른 성능비교(r=1/4)

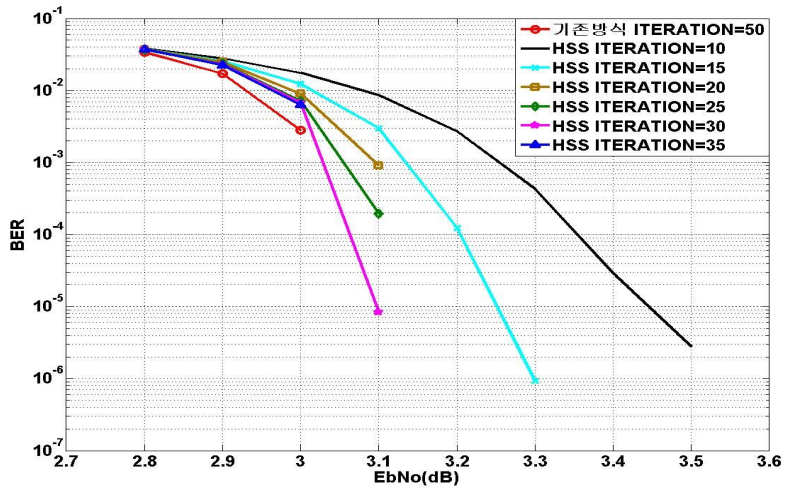


그림 3.11 기존방식과 HSS 방식의 반복횟수에 따른 성능비교(r=2/3)

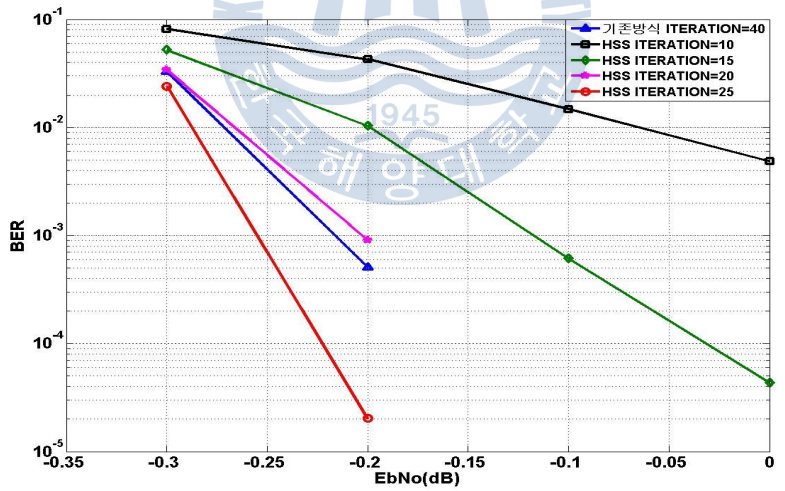


그림 3.12 기존방식과 HSS 방식의 반복횟수에 따른 성능비교(r=2/5)

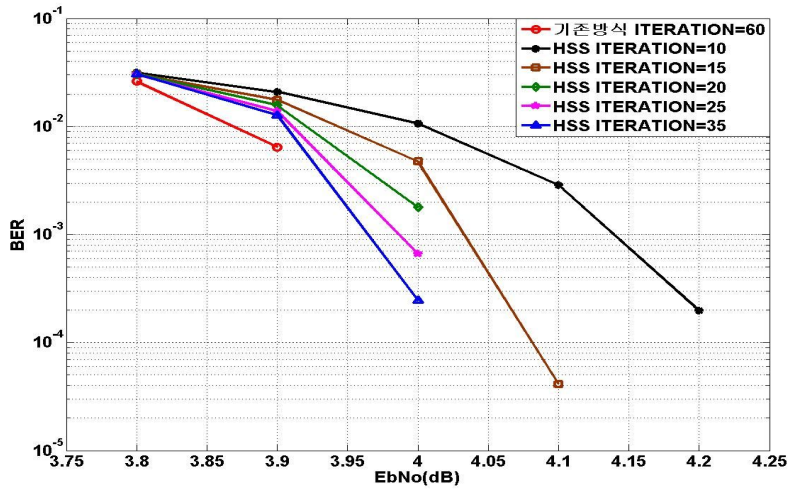


그림 3.13 기존방식과 HSS 방식의 반복횟수에 따른 성능비교(r=3/4)

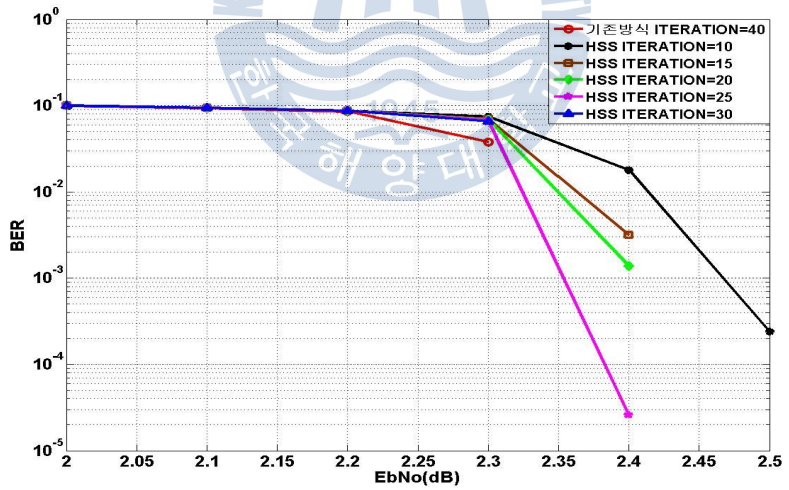


그림 3.14 기존방식과 HSS 방식의 반복횟수에 따른 성능비교(r=3/5)

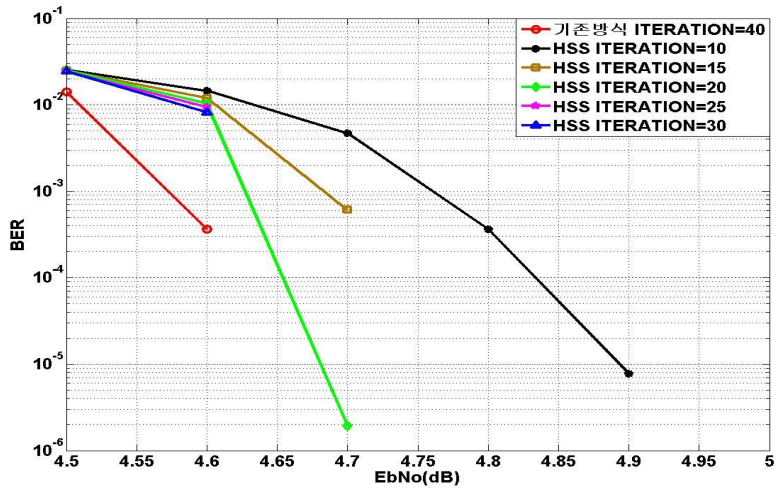


그림 3.15 기존방식과 HSS 방식의 반복횟수에 따른 성능비교(r=4/5)

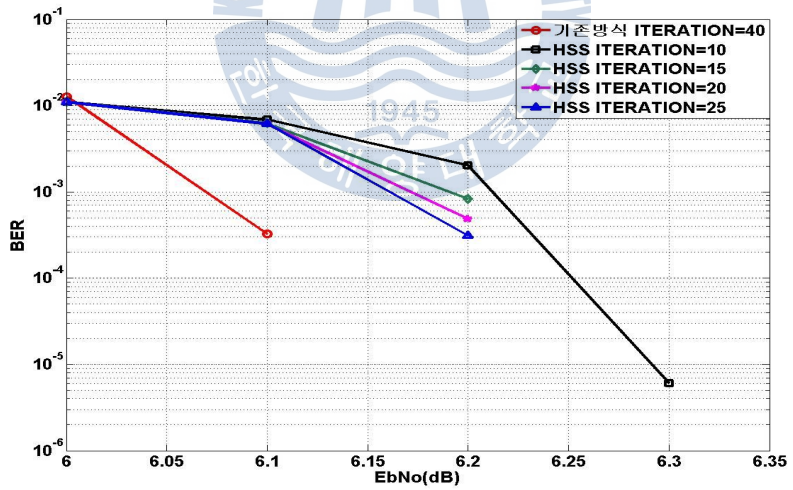


그림 3.16 기존방식과 HSS 방식의 반복횟수에 따른 성능비교(r=8/9)

성능비교에 사용된 데이터는 약 100만개이고, AWGN 환경에서 시뮬레이션을 하였다. 시뮬레이션의 모든 부호화 율에서 한 블록의 크기는 64800이다. 기존의 방식은 반복 횟수를 고정 하였고, HSS 방식은 반복 횟수를 조정하면서 성능을 비교하였다.

성능 비교 결과 1/2의 경우 기존의 방식과 HSS 방식의 반복 횟수 30회에서 거의 비슷한 성능을 보임을 알 수 있다. 이는 HSS 방식을 사용할 때, 한 번의 반복에서의 속도 증가 효과가 있음과 동시에 반복 횟수 역시 줄일 수 있으며, 기존의 방식에 비해 성능 열화도 거의 발생하지 않음을 의미한다. 아래 표는 각 부호화 방식에서 HSS 알고리즘 적용 시 요구되는 반복횟수를 나타낸다.

표 3.3 HSS 알고리즘 적용 시 요구되는 반복횟수

부호화 율	기존방식 (회)	HSS(회)	감소량(%)
1/4	40	20	50
1/3	40	20	50
1/2	60	30	50
2/3	50	25	50
2/5	40	25	38
3/4	60	30	50
3/5	40	30	25
4/5	40	25	38
5/6	50	30	40
8/9	40	25	38

다음 그림 3.17은 HSS 방식을 적용한 LDPC 복호기의 구조를 나타낸다.

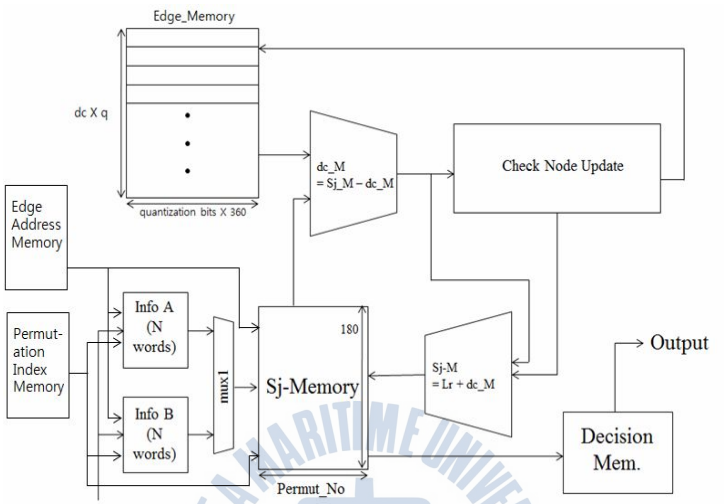


그림 3.17 HSS 방식을 적용한 LDPC 복호기의 구조

그림 3.17의 HSS 방식을 적용한 복호기의 구조를 보면 크게 메모리 부분과 CNU 부분으로 나눌 수 있다. 특히 수신 데이터를 저장하는 메모리와 Sj 메모리에서 데이터를 읽어 올 때는 H 매트릭스에 따라 랜덤하게 메모리 엑세스가 되어야 하기 때문에 이를 구현할 수 있는 여러 인덱스들이 필요하다.

HSS 방식을 적용한 LDPC 복호기는 BNU 계산을 위한 블록을 따로 만들지 않는다. 그 이유는 CNU 블록에서 나온 출력 값들을 바로 Sj 메모리에 업데이트 시키기 때문에 BNU를 위한 연산만을 따로 하지 않기 때문이다. 즉, LDPC 복호를 위한 연산 블록으로는 CNU 만을 필요로 한다.

제 4 장 서로 다른 메모리를 이용한 dc 병렬 알고리즘 제안

다음 표는 구현을 위한 FPGA 칩의 사양을 나타낸다.

표 4.1 Virtex-6 FPGA 칩 사양

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices[2]	Block RAM Blocks			MMCMs[4]	Interface Blocks for PCI Express	Ethernet MACs[5]	Maximum Transceivers		Total I/O Banks[6]	Max User I/O[7]
		Slices[1]	Max Distributed RAM (Kb)		18 Kb[3]	36 Kb	Max (Kb)				GTX	GTH		
XC6VLX75T	74,496	11,640	1,045	288	312	156	5,616	6	1	4	12	0	9	360
XC6VLX130T	128,000	20,000	1,740	480	528	264	9,504	10	2	4	20	0	15	600
XC6VLX195T	199,680	31,200	3,040	640	688	344	12,384	10	2	4	20	0	15	600
XC6VLX240T	241,152	37,680	3,650	768	832	416	14,976	12	2	4	24	0	18	720
XC6VLX365T	364,032	56,880	4,130	576	832	416	14,976	12	2	4	24	0	18	720
XC6VLX550T	549,888	85,920	6,200	864	1,264	632	22,752	18	2	4	36	0	30	1200
XC6VLX760	758,784	118,560	8,280	864	1,440	720	25,920	18	0	0	0	0	30	1200
XC6VSX315T	314,880	49,200	5,090	1,344	1,408	704	25,344	12	2	4	24	0	18	720
XC6VSX475T	476,160	74,400	7,640	2,016	2,128	1,064	38,304	18	2	4	36	0	21	840
XC6VHX250T	251,904	39,360	3,040	576	1,008	504	18,144	12	4	4	48	0	8	320
XC6VHX255T	253,440	39,600	3,050	576	1,032	516	18,576	12	2	2	24	24	12	480
XC6VHX380T	382,464	59,760	4,570	864	1,536	768	27,648	18	4	4	48	24	18	720
XC6VHX565T	566,784	88,560	6,370	864	1,824	912	32,832	18	4	4	48	24	18	720

위의 칩 중에서 본 논문에서 사용할 칩은 XC6VLX550T이다. 구현을 위해 XC6VLX550T칩은 하나의 블록 RAM의 사양 중에서 18Kb를 사용할 경우 1264개의 블록 RAM을 사용할 수 있고 36Kb를 사용할 경우 632개의 블록 RAM을 사용할 수 있다. 본 논문에서는 512 X 36의 18Kb 블록을 기반으로 LDPC의 부호화율에 따라 알맞은 알고리즘을 제시하였다.

제 4-1 절 edge메모리와 Sj메모리 각각의 병렬구조

기존의 HSS를 적용한 LDPC 복호기에서는 체크 노드 연산과정에서 edge 메모리와 Sj 메모리를 각각 하나씩 사용하였기 때문에 체크 노드 연산 과정에서 dc개의 데이터를 읽어오고 체크 노드 연산 후 다시 edge 메모리와 Sj 메모리에 저장하는데 지연이 발생하게 된다. 이러한 문제를 해결하여 고속 복호를 위해 체크 노드 연산에 필요한 dc개의 데이터를 한번에 읽어와 연산하고 한번에 저장할 수 있는 방안인 edge 메모리와 Sj 메모리를 각각 dc개로 분리하였다.

다음 그림은 edge 메모리와 Sj 메모리를 각각 dc개로 분리한 LDPC 복호기이다.

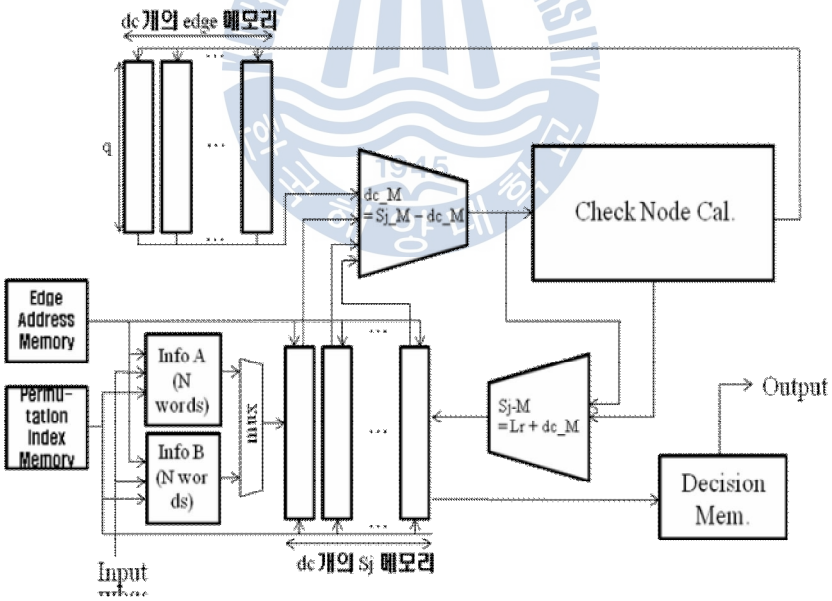


그림 4.1 dc개로 분리한 edge메모리와 Sj메모리 구조

edge 메모리와 Sj 메모리를 각각 dc개로 분리하게 되면 체크 노드 연산과정에서 동시에 dc개의 데이터를 연산할 수 있게 된다. 이렇게 되면

한번의 클럭으로 dc개의 데이터를 동시에 처리 가능하기 때문에 고속 복호가 가능하게 된다. 동시에 dc개의 데이터를 읽고 저장할 수 있게 되면 체크 노드 연산을 병렬로 처리할 수 있기 때문에 이 과정에서도 소요되는 클럭을 줄일 수 있다. 이때에 필요한 클럭 수는 $2^q \geq dc$ 를 만족하는 가장 작은 q로 나타낼 수 있다. 이에 따라 부호화율이 1/2인 경우 dc가 7이기 때문에 3 클럭의 연산으로 체크 노드 연산을 할 수 있게 된다.

다음 그림은 부호화율이 1/2인 경우 dc가 7일 때 체크 노드 연산과정을 나타낸다.

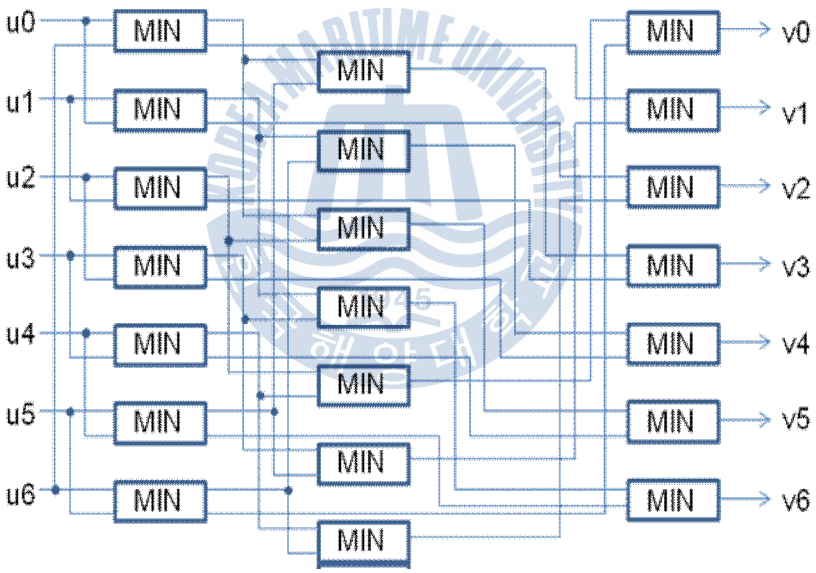
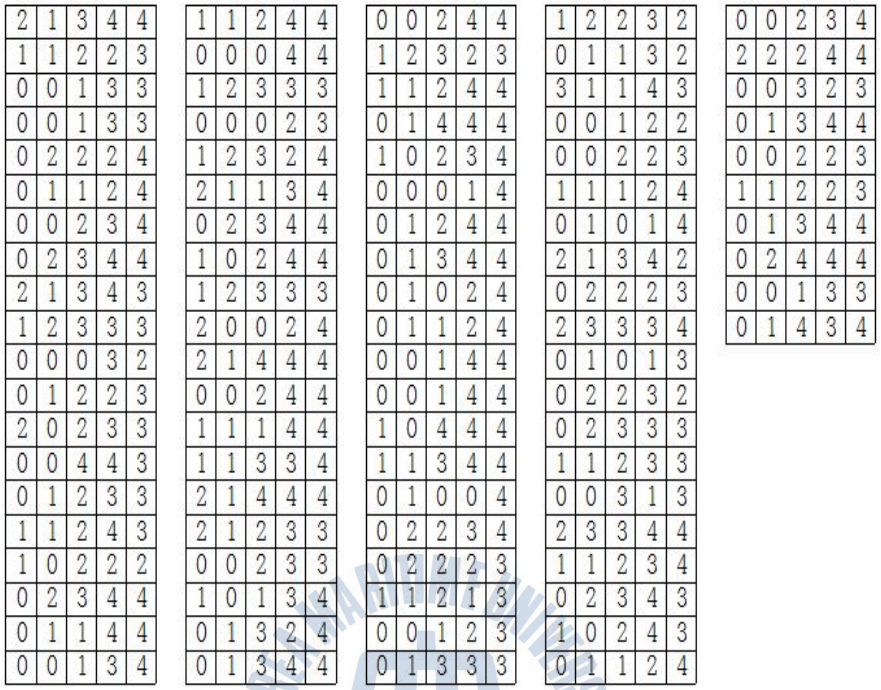


그림 4.2 체크 노드 연산의 parallel 구조

그림에서 보듯이 dc개의 체크 노드 연산에 3 클럭이 소요됨을 알 수 있다. 그림 4.1과 같이 edge 메모리와 Sj 메모리를 각각 dc개로 분리하기 위해서는 인덱스가 필요하다.



2	1	3	4	4
1	1	2	2	3
0	0	1	3	3
0	0	1	3	3
0	2	2	2	4
0	1	1	2	4
0	0	2	3	4
0	2	3	4	4
2	1	3	4	3
1	2	3	3	3
0	0	0	3	2
0	1	2	2	3
2	0	2	3	3
0	0	4	4	3
0	1	2	3	3
1	1	2	4	3
1	0	2	2	2
0	2	3	4	4
0	1	1	4	4
0	0	1	3	4

1	1	2	4	4
0	0	0	4	4
1	2	3	3	3
0	0	0	2	3
1	2	3	2	4
2	1	1	3	4
0	2	3	4	4
1	0	2	4	4
1	2	3	3	3
2	0	0	2	4
2	1	4	4	4
0	0	2	4	4
1	1	1	4	4
1	1	3	3	4
2	1	4	4	4
2	1	2	3	3
0	0	2	3	3
1	0	1	3	4
0	1	3	2	4
0	1	3	4	4

0	0	2	4	4
1	2	3	2	3
1	1	2	4	4
0	1	4	4	4
1	0	2	3	4
0	0	0	1	4
0	1	2	4	4
0	1	3	4	4
0	1	0	2	4
0	1	1	2	4
0	0	1	4	4
0	0	1	4	4
1	0	4	4	4
1	1	3	4	4
0	1	0	0	4
0	2	2	3	4
0	2	2	2	3
1	1	2	1	3
0	0	1	2	3
0	1	3	3	3

1	2	2	3	2
0	1	1	3	2
3	1	1	4	3
0	0	1	2	2
0	0	2	2	3
1	1	1	2	4
0	1	0	1	4
2	1	3	4	2
0	2	2	2	3
2	3	3	3	4
0	1	0	1	3
0	2	2	3	2
0	2	3	3	3
1	1	2	3	3
0	0	3	1	3
2	3	3	4	4
1	1	2	3	4
0	2	3	4	3
1	0	2	4	3
0	1	1	2	4

0	0	2	3	4
2	2	2	4	4
0	0	3	2	3
0	1	3	4	4
0	0	2	2	3
1	1	2	2	3
0	1	3	4	4
0	2	4	4	4
0	0	1	3	3
0	1	4	3	4

그림 4.3 그림 4.1의 구조를 구현하기 위한 인덱스

그림 4.3의 인덱스는 Sj 메모리를 dc개로 분리했기 때문에 구현을 위해 필요한 인덱스이다. 예를 들어 그림 2.1에 나타낸 R_index에서 26이 0번째 row에서는 1번째에 나타나고 12번째 row에서는 2번째에 나타난다. 이렇게 되면 0번째 row에서 26은 dc개로 나눈 Sj메모리 중에 1번째 Sj메모리에 업데이트된 값이 저장이 된다. 다음 계산을 위해서는 업데이트된 값을 가져와야 하는데 1번째 Sj메모리에 업데이트 된 값이 저장되어 있기 때문에 12번째 row에서 26을 계산하기 위해서는 1번째 Sj메모리에서 26에 해당하는 값을 가져와야 한다. 이와 마찬가지로 모든 row에 대하여 몇 번째 Sj메모리에서 값을 가져와야 하는지를 그림 4.3에 나타내었다.

다음은 기존의 HSS 알고리즘과 그림 4.1의 구조를 적용시켰을 때 HSS 알고리즘의 성능을 비교한 그림이다. 부호화율은 1/2이고 이때 dc

의 개수는 7이다.

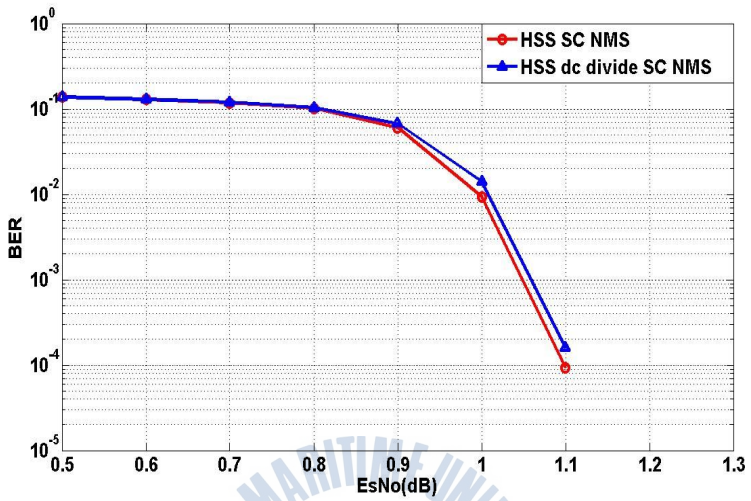


그림 4.4 HSS 알고리즘과 그림 4.1 구조의 성능비교

edge 메모리와 Sj 메모리를 dc개로 나눈 알고리즘과 기존의 HSS 알고리즘의 성능을 비교한 결과 거의 비슷한 성능을 보임을 알 수 있다.

하지만 그림 4.1과 같은 구조를 사용하게 되면 메모리 관점에서 edge 메모리와 Sj 메모리가 각각 dc개 만큼 필요하기 때문에 dc가 10을 초과하는 경우에는 메모리량이 많아져 XC6VLX550T칩의 메모리가 부족하여 구현에 어려움이 있게 된다. 그렇기 때문에 dc가 10을 초과하는 경우에는 메모리를 효율적으로 사용할 수 있는 새로운 알고리즘이 필요하다.

제 4-2 절 edge메모리와 Sj메모리를 공유한 병렬구조

edge 메모리와 Sj메모리를 각각 dc개로 나눴을 때 dc가 10을 초과하게 되면 메모리량이 많아져 구현에 있어 어려움이 있게 된다. 메모리관점에서의 문제를 해결하기 위해 Sj 메모리와 edge 메모리를 하나의 메모리로 사용하는 방안을 제시한다. 다음 그림 4.5는 Sj 메모리와 edge 메모리를 dc개로 나눈 후 메모리를 공유하는 방식의 구조이다.

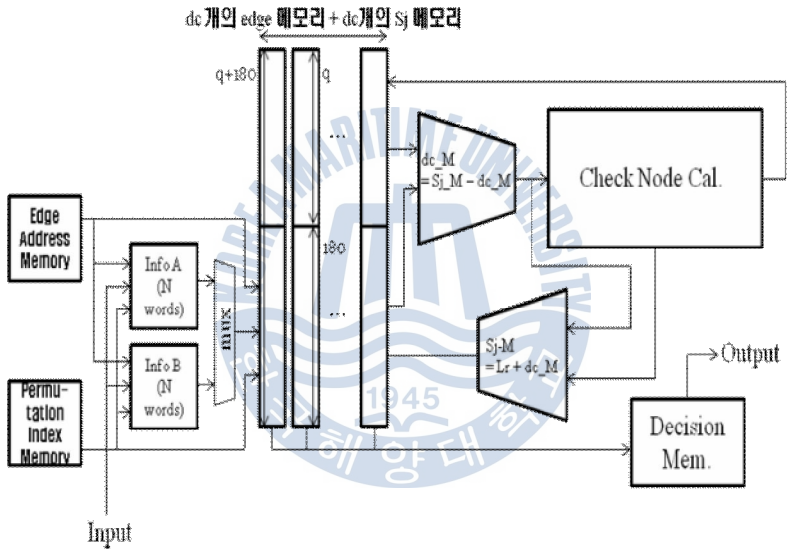


그림 4.5 dc개로 나눈 Sj 메모리와 edge 메모리를 공유한 구조

Sj 메모리와 edge 메모리를 공유하면 전체적인 메모리량은 변함없다. 하지만 메모리의 수를 줄일 수 있게 된다. Sj 메모리와 edge 메모리를 각각 dc개로 나누면 2 x dc개의 메모리가 필요하지만 Sj 메모리와 edge 메모리를 공유하게 되면 필요한 메모리 수를 dc개로 줄일 수 있게 된다. 이 방식을 이용하게 되면 dc의 개수가 20개까지 체크 노드 연산의 parallel 구조를 적용하여 구현할 수 있게 된다.

dc개로 나눈 Sj 메모리와 edge 메모리를 공유한 구조 또한 Sj 메모리와 edge 메모리를 각각 dc개로 나누었을 때와 마찬가지로 그림 4.3과 같은 인덱스가 필요하게 된다.

0	1	2	2	2	4	6	6	6	9	9	11
0	2	1	1	5	5	6	6	8	9	11	11
0	1	1	2	5	5	6	6	8	9	9	9
1	0	3	1	1	4	6	6	6	9	10	11
1	2	4	3	5	5	6	6	6	9	10	11
1	1	1	3	4	4	8	7	8	9	9	10
0	3	3	3	2	4	6	7	8	9	9	9
2	3	3	2	3	4	6	6	6	9	9	10
1	2	2	2	4	3	6	6	6	9	9	10
0	0	0	1	1	4	6	7	8	9	10	10
0	2	4	3	4	2	7	7	8	9	10	10
0	0	1	3	3	3	6	6	7	9	9	10
0	1	4	4	5	5	7	8	7	9	9	10
1	3	3	4	4	4	6	6	8	9	11	10
0	0	2	3	3	3	7	7	7	11	9	11
1	0	1	4	4	4	6	8	8	10	10	9
0	3	3	2	5	5	6	8	8	9	11	11
1	0	0	5	4	4	8	8	8	9	10	11
1	1	1	2	4	4	6	6	7	10	9	10
0	0	2	3	4	5	6	8	8	9	10	10
1	2	2	4	4	5	7	7	7	11	10	11
0	0	1	3	4	5	6	8	8	9	9	11
0	1	2	1	3	5	7	8	7	10	9	10
0	0	0	5	5	5	6	6	7	10	10	10
0	4	3	3	4	5	7	8	7	11	10	11
3	1	2	5	3	5	7	7	7	10	9	10
0	2	4	4	4	5	7	6	7	11	11	10
2	0	1	3	4	3	6	6	7	9	10	11
2	2	2	4	4	4	8	8	7	10	11	11
0	1	2	2	5	5	8	8	8	9	9	10
1	0	0	0	3	4	7	6	8	9	10	11
0	2	1	3	5	5	7	6	8	9	11	11
0	1	0	1	3	5	6	7	7	11	11	11
3	1	1	5	5	5	6	6	7	10	11	11
1	2	3	5	5	4	6	6	7	10	10	10
2	1	2	1	5	5	6	8	8	9	10	11
0	1	1	3	4	5	7	8	8	10	11	11
0	2	2	3	3	5	6	8	8	10	11	11
1	0	4	5	3	5	7	7	8	11	11	11
0	0	1	4	5	5	6	7	8	9	11	11
0	0	2	2	3	5	7	7	8	9	10	11
1	3	2	4	5	5	7	7	8	9	11	11
0	0	0	3	3	5	6	8	8	9	9	11
0	0	1	2	4	5	6	7	8	10	10	11
0	1	3	3	5	5	7	8	8	9	10	11

그림 4.6 그림 4.5의 구조를 구현하기 위한 인덱스

위 인덱스도 앞서 설명한 그림 4.3의 인덱스처럼 Sj 메모리를 dc개로 나누었기 때문에 앞에서 업데이트 된 값이 몇 번째 Sj 메모리에 저장되었는지를 확인하고 그에 해당하는 값을 가져오기 위한 인덱스이다.

다음은 그림 4.5의 알고리즘을 적용시켰을 때의 성능과 기존의 HSS 방식의 성능을 비교한 그림이다. 부호화율은 3/4을 사용하였고 이때의 dc 개수는 14이다.

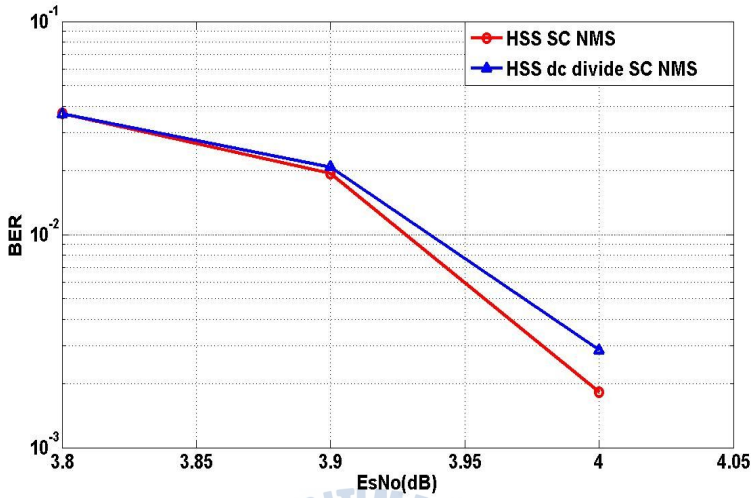


그림 4.7 HSS 알고리즘과 그림 4.5 구조의 성능비교

Sj 메모리와 edge 메모리를 dc개로 나눈 후 공유한 알고리즘의 성능이 기존의 HSS 알고리즘과 거의 비슷한 것을 알 수 있다. 하지만 이 방식의 경우에는 dc의 개수가 20인 경우까지만 구현할 수 있다. 부호화율이 5/6인 경우 dc의 개수는 22, 부호화율이 8/9은 dc의 개수는 27, 부호화율이 9/10인 경우에는 dc의 개수가 30이다. 그렇기 때문에 dc의 개수가 20을 초과하는 부호화율 5/6, 8/9, 9/10을 구현하기 위해서는 또 다른 알고리즘이 필요하다.

제 4-3 절 edge메모리와 Sj메모리를 이중으로 공유한 병렬구조

제 1절의 알고리즘은 dc의 개수가 10, 제 2절의 알고리즘은 dc의 개수가 20까지의 부호화율에서 구현이 가능하다. dc의 개수가 20을 초과하는 부호화율을 구현하기 위해 하나의 메모리에 edge 메모리와 Sj메모리를 각각 2개를 포함하는 알고리즘을 제시한다. 다음 그림은 두 개의 Sj 메모리와 edge 메모리를 동일한 메모리에 적용한 알고리즘이다.

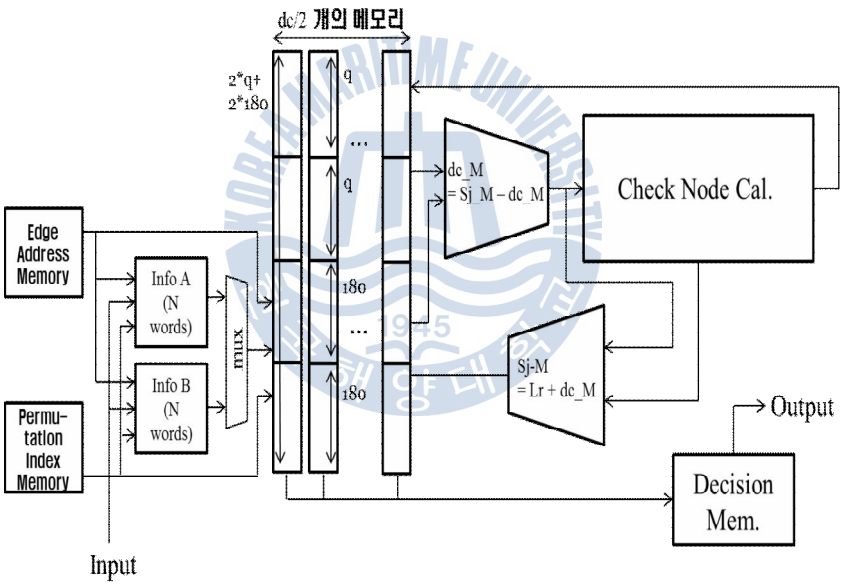


그림 4.8 두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘

두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘 또한 전체적인 메모리량을 변함이 없다. 하지만 Sj 메모리와 edge 메모리를 각각 2개씩 하나의 메모리 안에 넣었기 때문에 총 필요한 메모리 수는 dc개의 절반만이 필요하게 된다. 부호화율이 9/10인 경우 dc의 개수가 30이기

때문에 30의 절반인 15개의 메모리가 필요하므로 구현이 가능하게 된다. 이 경우에는 위에서 살펴본 알고리즘과는 달리 두 개씩을 하나의 메모리가 포함하기 때문에 Sj 메모리와 edge 메모리는 1번째 메모리부터 15번째 메모리는 위쪽에, 16번째 메모리부터 30번째 메모리는 아래쪽에 저장되게 된다.

다음은 그림 4.8의 알고리즘을 구현하기 위한 인덱스를 나타낸다.

0	0	2	2	4	4	6	7	7	9	10	10	10	13	15	13	16	16	16	19	19	20	22	22	22	25	26	25
1	0	1	3	4	4	6	8	7	7	10	10	11	14	13	15	17	16	17	19	21	20	22	23	24	25	25	27
0	0	1	3	4	5	6	7	7	7	11	11	12	13	13	13	17	16	18	19	19	21	22	23	24	25	25	27
1	1	3	3	4	5	4	7	7	8	11	12	11	14	13	14	16	18	18	19	20	19	23	23	23	25	25	26
0	0	1	2	4	4	4	9	8	9	11	12	11	13	14	14	16	16	17	20	20	20	22	23	23	25	27	26
2	2	2	3	4	6	6	7	8	8	10	10	11	13	13	14	16	16	17	19	20	20	24	22	22	25	25	27
1	0	1	1	4	5	6	7	7	9	10	10	12	14	14	15	16	17	17	19	20	20	23	23	24	25	26	25
0	0	2	2	6	4	5	7	7	7	10	10	12	15	14	14	16	16	18	21	20	20	24	24	22	25	27	25
2	3	2	3	5	5	5	8	8	8	10	12	10	15	15	15	17	17	18	19	20	20	22	23	23	26	26	27
0	1	1	2	5	5	5	8	9	8	10	12	12	13	13	14	18	16	18	19	21	20	22	23	23	26	26	27
1	1	3	3	6	6	5	8	8	9	10	12	11	13	15	14	16	18	17	19	20	21	22	24	24	27	27	27
1	0	2	2	5	5	6	7	8	8	10	11	12	13	13	13	16	16	17	21	21	20	23	23	23	26	27	27
0	2	1	3	6	6	6	7	9	8	11	11	11	16	15	15	18	18	18	20	20	21	22	22	24	25	27	27
0	3	3	3	4	4	5	8	9	7	10	12	11	14	14	14	16	17	17	19	21	21	23	24	24	25	26	26
0	0	0	2	4	5	5	8	8	9	11	12	12	13	15	15	17	17	18	19	21	21	22	24	23	25	26	27
0	1	2	3	4	4	6	9	9	9	11	11	12	14	14	15	17	18	18	19	19	21	22	22	24	26	26	27
1	2	3	3	4	6	6	7	9	9	10	10	12	13	15	14	16	17	18	19	21	21	22	24	24	25	25	27
0	2	1	3	5	5	6	7	9	9	10	12	12	15	15	15	18	16	18	19	21	21	22	24	24	26	26	27

그림 4.9 그림 4.8의 구조를 구현하기 위한 인덱스

그림 4.9의 인덱스는 부호화율이 9/10이고 dc의 개수가 30일 때의 인덱스이다. 두 개의 Sj 메모리와 edge 메모리를 공유한 구조이기 때문에 Sj 메모리와 edge 메모리를 저장할 수 있는 공간은 1번째부터 15번째이기 때문에 위의 인덱스에서 값이 15이상일때는 Sj 메모리는 Sj[180+R_index][S_index-15]가 되어야 한다.

다음은 두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘과 기존 HSS 알고리즘의 성능을 비교한 그림이다. 부호화율은 9/10이고 이때 dc의 개수는 30이다.

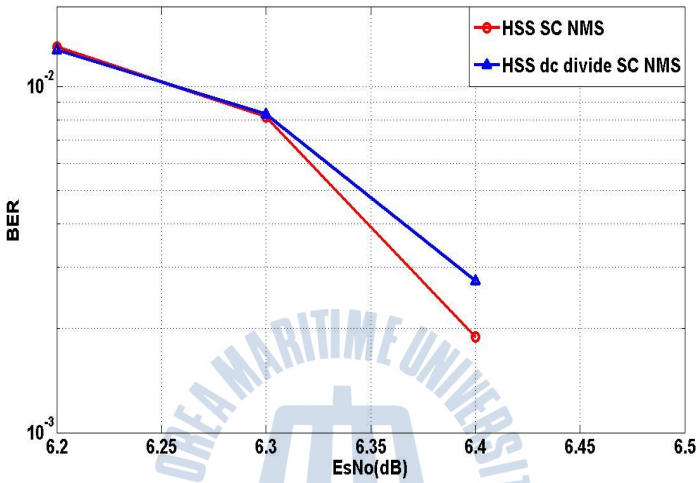


그림 4.10 HSS 알고리즘과 그림 4.8 구조의 성능비교

두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘의 성능이 기존 HSS 알고리즘의 성능과 거의 비슷함을 알 수 있다. 두 개의 Sj 메모리와 edge 메모리를 공유한 경우 dc의 개수가 20을 초과했을 때에도 구현이 가능함을 알 수 있다.

제 4-4 절 알고리즘 및 부호화율에 따른 FPGA 메모리 할당

본 절에서는 앞에서 살펴 본 3가지 알고리즘을 토대로 부호화율에 따라 어떤 방식이 적합한지를 알아보겠다.

다음 표 4.2에서는 각 부호화율에 대한 dc의 개수와 q의 크기를 나타내었다.

표 4.2 부호화율에 따른 dc의 개수와 q의 크기

부호화율	dc	q
1/2	7	90
1/3	5	120
2/3	10	60
1/4	4	135
3/4	14	45
2/5	6	108
3/5	11	72
4/5	18	36
5/6	22	30
8/9	27	20
9/10	30	18

표 4.2의 부호화율에 따른 dc의 개수와 q에 따라서 위에서 살펴본 알고리즘 중에서 적합한 알고리즘을 택할 수 있다.

다음 표 4.3은 Virtex-6 FPGA 칩을 토대로 각각의 부호화율에 적합한 알고리즘을 나타내었다.

표 4.3 부호화율에 따른 18Kb 블록 기반에 적합한 방식

부호화율	블록 RAM	Method
1/2	180 x 360 x 7 x 6 90 x 360 x 7 x 6	1번 방식
1/3	180 x 360 x 5 x 6 120 x 360 x 5 x 6	1번방식
2/3	180 x 360 x 10 x 6 60 x 360 x 10 x 6	1번방식
1/4	180 x 360 x 4 x 6 135 x 360 x 4 x 6	1번방식
2/5	180 x 360 x 6 x 6 108 x 360 x 6 x 6	1번방식
3/4	(180+45) x 360 x 14 x 6	2번방식
3/5	(72+180) x 360 x 11 x 6	2번방식
4/5	(36+180) x 360 x 18 x 6	2번방식
5/6	(30*2+180*2) x 360 x 22 x 6	3번방식
8/9	(20*2+180*2) x 360 x 27 x 6	3번방식
9/10	(18*2+180*2) x 360 x 30 x 6	3번방식

구현을 위해 XC6VLX550T칩 중에서 512 X 36의 18Kb 블록을 기반으로 하였다. 18Kb 블록을 사용할 경우 1264개의 512 X 36의 메모리를 사용할 수 있다. 표 4.3에서 1번 방식은 dc개의 Sj 메모리와 dc개의 edge 메모리로 나눈 알고리즘, 2번 방식은 dc개의 Sj 메모리와 edge 메모리를 공유한 알고리즘이고 3번 방식은 두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘이다.

각각의 방식을 예로 들면 1번 방식 중에서 부호화율이 1/2인 경우 180 x 360의 Sj 메모리가 6비트씩 dc개인 7개가 필요하고 90(q) x 360의

edge 메모리도 6비트씩 dc개인 7개가 필요하다. 이때, 512 X 36의 메모리는 각각 420개씩 총 840개가 필요하기 때문에 1번방식으로 구현이 가능하다.

2번 방식의 대표적인 부호화율 3/4에서는 dc의 개수가 10을 넘기 때문에 칩의 메모리 사용량이 초과하게 되어 1번 방식으로는 구현을 할 수가 없다. 그렇기 때문에 2번 방식을 사용하게 되면 (45+180) x 360의 Sj 메모리와 edge 메모리를 공유한 메모리가 6비트씩 dc개인 14개가 필요하다. 이때, 512 X 36의 메모리는 840개가 필요하여 2번 방식으로 구현이 가능하다.

마지막으로 3번 방식의 대표적인 부호화율 9/10에서는 dc의 개수가 30이기 때문에 2번 방식을 사용할 경우 1800개의 메모리가 필요하기 때문에 구현을 할 수가 없다. 그러나 3번 방식을 사용할 경우 (2*18+2*180) x 360의 메모리가 6비트씩 dc/2인 15개가 필요하고 512 X 36의 메모리는 900개가 필요하기 때문에 3번 방식으로 구현이 가능하다.

제 5 장 복호 알고리즘에 따른 속도 및 메모리

제 3장과 제 4장에서 효율적인 고속 LDPC 복호 알고리즘에 대해 알아보았다. 제 5장에서는 위에서 살펴본 5가지의 복호 알고리즘에 따른 메모리량과 속도를 비교하여 고속 복호를 위한 최적의 알고리즘을 알아보겠다.

다음 표 5.1은 위에서 제시한 3가지 복호 알고리즘에 따른 각각의 메모리 사용량을 비교한 표이다.

표 5.1 복호 알고리즘에 따른 메모리 비교

	요구되는 메모리	각 메모리 수	요구되는 메모리 총 개수	메모리의 양
HSS(부호화율 1/2)	dc x 360 x q 180 x 360	1개 1개	2개	7 x 360 x 90 x 6 +180 x 360 x 6 = 1749600
dc 개의 Sj 메모리와 dc 개의 edge 메모리	q x 360 180 x 360	dc 개 dc 개	2 x dc개	90 x 360 x 7 x 6 + 180 x 360 x 7 x 6 = 4082400
HSS(부호화율 3/4)	dc x 360 x q 180 x 360	1개 1개	2개	14 x 360 x 45 x 6 +180 x 360 x 6 = 1749600
dc 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘	(q+180) x 360	dc 개	dc 개	270 x 360 x 14 x 6 = 8164800
HSS(부호화율 9/10)	dc x 360 x q 180 x 360	1개 1개	2개	30 x 360 x 18 x 6 +180 x 360 x 6 = 1555200
두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘	(2·q+2·180) x 360	dc/2 개	dc/2 개	396 x 360 x 15 x 6 = 12830400

위의 표를 보게 되면 부호화율이 1/2인 HSS 알고리즘의 경우 edge 메모리와 Sj 메모리가 1개씩 총 2개가 필요하고 필요한 총 메모리의 사용량은 1,749,600 bit이다. dc개로 분리하여 edge 메모리와 Sj 메모리를

각각 나눈 구조의 경우는 edge메모리와 Sj메모리가 각각 dc개만큼 필요하기 때문에 전체적인 메모리의 사용량은 4,082,400 bit이다.

부호화율이 3/4일 때 dc개로 나눈 edge 메모리와 Sj 메모리를 공유한 알고리즘의 전체적인 사용량은 8164800 bit이고, 부호화율이 9/10일 때 두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘의 경우 총 메모리 사용량은 12,830,400 bit이다. 메모리 사용량에서는 HSS 방식에 비해 훨씬 많은 양의 메모리를 사용함을 알 수 있다. 하지만 dc개로 나눈 edge 메모리와 Sj 메모리를 공유한 알고리즘이 dc개로 분리하여 edge 메모리와 Sj 메모리를 각각 나눈 알고리즘보다 메모리의 수가 더 작고 두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘이 dc 개로 나눈 edge 메모리와 Sj 메모리를 공유한 알고리즘보다 메모리 수가 더 작음을 알 수 있다. 그렇기 때문에 부호화율에 따른 dc의 개수에 따라 적합한 알고리즘을 선택하여 사용할 수 있다. 다음 표 5.2는 5가지 복호 알고리즘에 대한 각각의 속도를 비교한 표이다.

표 5.2 복호 알고리즘에 따른 속도비교

	요구되는 clock (1회 반복)	HSS 적용 가능 여부	최대 반복 시 요구되는 clock	10ns clock에서 throughput
HSS(부호화율 1/2)	662 clock	가능	19860 clock (30회)	326 Mbit/s
dc 개의 Sj 메모리와 dc 개의 edge 메모리	94 clock	가능	2820 clock (30회)	2.29 Gbit/s
HSS(부호화율 3/4)	698 clock	가능	20970 (30회)	309 Mbit/s
dc 개의 Sj 메모리와 edge 메모리를 공유한 구조	100 clock	가능	3000 clock (30회)	2.16 Gbit/s
HSS(부호화율 9/10)	688 clock	가능	20670 (30회)	313 Mbit/s
두 개의 Sj 메모리와 edge 메모리를 공유한 구조	96 clock	가능	2880 clock (30회)	2.25 Gbit/s

부호화율에 따른 각각의 복호 알고리즘 속도를 살펴보면 부호화율1/2에서 HSS 알고리즘은 1회 반복에 662 클럭이 필요하며 전체 반복횟수를 기존 LDPC에 비해 절반으로 줄일 수 있기 때문에 최대 반복 시 요구되는 클럭은 19860 클럭이 소요된다. 이에 비해 edge 메모리와 Sj 메모리를 각각 dc개로 나눈 경우에는 1회 반복에 94 클럭이 필요하고 최대 반복 시 2820 클럭이 소요된다.

부호화율 3/4일 때 HSS 알고리즘은 1회 반복에 698 클럭이 필요하고 최대 반복 시 20970 클럭이 소요된다. 이에 비해 dc개로 나눈 edge 메모리와 Sj 메모리를 공유한 경우에는 1회 반복에 100 클럭이 필요하고 최대 반복 시 3000 클럭이 소요된다. 또한, 부호화율이 9/10에서 HSS 알고리즘은 1회 반복에서 688 클럭, 최대 반복 시 20670 클럭이 필요하다. 이에 비해 두 개의 Sj 메모리와 edge 메모리를 공유한 알고리즘에서는 1회 반복에서 96 클럭, 최대 반복 시 2880 클럭이 소요된다.

edge 메모리와 Sj 메모리를 dc개로 나눈 알고리즘은 1회 반복에서 상대적으로 HSS 알고리즘 보다 적은 클럭이 필요하다. 그 이유는 체크 노드 연산과정에서 dc개 만큼 분리되어 한번에 데이터를 읽고 저장하는 것이 가능하고 또한 체크 노드 연산을 병렬로 처리 가능하기 때문에 HSS 알고리즘보다 적은 클럭이 필요하게 되고 고속 복호가 가능하다.

제 6 장 결 론

초고화질 실감 방송서비스를 전국 단일시청권으로 제공하기 위한 방송전송 핵심기술의 대두로 100Mbps 급 이상의 채널 적응형 위성방송 전송기술의 연구가 활발히 진행중이다. 이에 본 논문에서는 100Mbps 급 이상의 전송 기술을 위해 DVB-S2 기반을 둔 고속 LDPC 부호화 알고리즘과 고속 LDPC 복호화 알고리즘을 연구하였고 최적의 메모리 구조 설계에 대해 연구하였다.

본 논문에서는 첫째로 고속 부호화 방식을 연구하였다. 기존의 LDPC 부호화 방식에서는 이전의 패리티 값을 알아야 다음 패리티 값을 계산할 수 있기 때문에 속도면에서 고속으로 할 수 없는 단점이 있다. 이에 고속 복호를 위해 기존의 직렬구조에서 360 개의 부분 병렬을 이용하여 부호화 구조에서 메모리를 효율적으로 적용하는 알고리즘을 제안하였고 기본 주파수 클럭 100MHz 에서 약 10Gbps 급의 부호기 구조를 설계하였다.

둘째로 기존의 LDPC 복호 방식에서는 체크 노드 업데이트 연산 후 비트 노드 연산을 하기 때문에 한번의 반복에서 많은 지연이 발생하는 문제점이 있다. 이에 비트 노드 계산을 체크 노드 계산 후에 하지 않고 체크 노드 계산 중에 수행하는 HSS 방식을 기반으로 하는 복호기 구조에서 기존의 방식보다 고속화 할 수 있는 방안을 연구하여 시뮬레이션 결과 약 30%~50% 정도의 반복횟수를 동일한 성능에서 감소시킬 수 있음을 확인할 수 있었다.

셋째로, 복호기 구조에서 HSS 방식을 기반으로 보다 복호과정을 고속화할 수 있는 알고리즘을 제안하였다. 기존의 dc 개의 직렬 구조에서 dc 개의 병렬 구조로 Sj 메모리와 dc 메모리를 나누어 체크 노드 연산을 고속으로 할 수 있는 구조이다. dc 개의 Sj 메모리와 dc 개의 edge 메모리를 나눈 구조에서는 칩의 사양을 고려하여 dc 의

개수가 10 이하의 부호화율까지 구현이 가능하고 dc 개의 Sj 메모리와 dc 개의 edge 메모리를 공유한 구조에서는 dc 의 개수가 20 이하까지 구현이 가능하며 두 개의 Sj 메모리와 edge 메모리를 공유한 구조에서는 모든 부호화율에서 구현이 가능함을 알 수 있다. 또한, dc 개의 직렬구조에서 dc 개의 병렬 구조로 메모리 구조를 변경함으로써 복호 속도를 개선할 수 있으나 이는 메모리의 낭비를 초래하며, 메모리 효율성 면에서는 다소 떨어진다 할 수 있다. 따라서 메모리 효율성의 측면을 감안하더라도 고속화를 극대화 할 수 있는 알고리즘으로 기존의 방식보다 약 7 배 이상의 고속 복호가 가능하리라 사료된다.

향후 고속화를 위한 복호 속도 뿐만 아니라 고속 복호를 유지하면서 메모리의 효율성까지 만족시킬 수 있는 알고리즘에 대한 연구가 필요하리라 사료된다.

참 고 문 헌

[1] Digital Video Broadcasting(DVB). “Second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broadband satellite applications (DVB-S2).” European Standard (Telecommunications series) ETSI EN 302 307 V1.2.1(2009-08),2009.

[2] R. G. Gallager, “Low-Density Parity-Check Codes,”IRE trans.information theory,vol.8, PP.21-28,1962.

[3] D. J. C. Mackay and R. M. Neal, “Near Shannon Limit Performance of Low-Density Parity-Check Codes,” Electron. Letter, Vol.32, PP. 1645-1646,Aug.1996.

[4] Arthur S., Francois V., David D., Pascal U." DVB-S2 compliant LDPC decoder integrating Horizontal Shuffle Scheduling,", ISPACS2006, pp. 1013-1016.

[5] J. Dielissen, A. Hekstra, and V. Berg. “Low cost LDPC decoder for DVB-S2.” In design, Automation and Test in Europe, 2006. DATE’06. Proceedings, volume 2, pages 1-6, Munich, Germany, March 2006.

[6] O. Eljamaly and P. Sweeney. “Alternative approximation of check node algorithm for DVB-S2 LDPC decoder.” In Second International Conference on Systems and Networks Communications (ICSNC 2007), pages 157-162, October 2007.

[7] M.P.C Fossorier, M. Mihaljevic, and H. Imai.”Reduced complexity iterative decoding of low-density parity check codes based on belief propagation.” IEEE Transactions on communications, 47 : 673-680, May 1999.

[8] M. Gomes, G. Falcao, V. Silva, V. Ferreira, A. Sengo, and M. Falcao. ”Flexible parallel architecture for DVB-S2 LDPC decoders.” In Global Telecommunications Conference, 2007. GLOBECOM’07. IEEE, pages 3265-3269, Washington, USA, November 2007.

[9] S.L.Fogal, S. Dolinar, and K. Andrews. “Buffering requirement for variable-iteration ldpc decoder.” In Information Theory and Application Workshop, ITA. 2008, pages 523-530, San Diego, CA, January 2008.

[10] D.E. Hocever. “A reduces complexity decoder architecture via layered decoding of LDPC codes.” In Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on, pages 107-112, Austin, USA, October 2004.

[11] Xiao-Yu Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia, “Efficient implementations of the sum-product algorithm for decoding LDPC codes.” In Global Telecommunications Conference, 2001. GLOBECOM’01. IEEE, volume 2, pages 1036-1036E vol.2,2001.

[12] C. Marchand, L. Conde-Canencia, and E. Boutillon. “Architecture and finite precision optimization for layered LDPC decoders.” In Signal Processing Systems, 2010. SiPS 2010. IEEE Workshop on. Accepted, San Francisco, USA, October 2010.

[13] C.Marchand, J.-B. Dore, L. Conde-Canencia, and E. Boutillon. “Conflict resolution for pipelined layered LDPC decoders.” ” In Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on. Pages 220-225, Tampere, Finlande, November 2009.

[14] V. Savin. “Self-corrected min-sum decoding of LDPC codes.” Pages 146-150,2008.

[15] J. Chen and M. Fossorier. “Density evolution of two improved bp-based algorithms for LDPC decoding“, IEEE communication letters, March 2002.

[16] S.M.Kim, C.S.Park, and S.Y.Hwang, “A Novel Partial Parallel Architecture for High-throughput LDPC Decoder for DVB-S2”, IEEE Transactions on consumer electronics, Volumn56 No.2,May,2010

[17] F. Kienle, T. Brack, and N. Wehn, “A synthesizable IP core for DVB-S2 LDPC code decoding,”Proc. Design, Automation and Test in Europe 2005, Vol. 3, pp. 100 -105, Mar. 2005.

[18] Takashi Yokokawa, Misa Nakane, and Makiko Kan, “A Low Complexity and Programmable Encoder Architecture of the LDPC codes for DVB-S2.”, in 3rd turbo coding Conference, Munich, Germany, Apirl 2006

[19] P. Urad, E. yeo, L. Paumier, P. Georgelin, T. Michel, V. Lebars, E. Yeo, and B. Gupta, “A 135Mb/s DVB-S2 Compliant CODEC based on 64800b LDPC and BCH Codes,” Proc. ISSCC 2005, San Francisco, USA, pp446-447, Feb. 2005.

[20] Byeong Su Lim, Min Hyuk Kim, Tae Doo Park, Gun Yeol Park, Ji Won Jung, “An Efficient Design Methodology of High-Speed LDPC decoder”, ICSSC2012, Otawa, Canada, September, 2012

감사의 글

대학에 입학한지 6년이란 세월이 지나갔습니다. 6년이란 시간 동안 주위의 많은 분들로 인하여 마음과 신체, 성격, 그리고 지식 등 많은 성장을 할 수 있었습니다. 제가 이렇게 성장하기까지 많은 도움을 주신 분들께 진심으로 감사 드립니다.

먼저, 제가 이 자리에 있기까지 세심한 지도와 따뜻한 격려를 해주시고 부족한 부분을 채울 수 있도록 이끌어주신 정지원 교수님께 감사의 인사말을 올립니다. 부족한 것도 많았고 실수도 많았지만 교수님의 독려와 지도 덕분에 무사히 대학원을 졸업하고 취직도 할 수 있었습니다. 다시 한 번 정지원 교수님께 감사 드립니다. 그리고 논문을 보완하여 보다 충실한 내용이 될 수 있도록 논문 심사를 맡아주신 김기만 교수님, 윤영 교수님께 감사 드립니다. 또한 학부시절 가르침을 주신 김동일 교수님, 조형래 교수님, 강인호 교수님, 민경식 교수님께도 감사 드립니다.

남들보다 늦게 대학원에 입학하여 연구실 생활하는데 힘든 점이 많았는데 2년동안 많은 도움을 준 민혁이형, 태두형, 철승이형 감사합니다. 민혁이형은 취직도 했으니 좋은 여자 만나서 빨리 결혼했으면 좋겠습니다. 태두형도 남은 시간 잘 준비하셔서 좋은 회사에 취직하세요. 저는 이제 연구실을 떠나지만 앞으로 연구실을 지킬 군열이형, 해찬이형, 태훈이도 고맙고 연구실 잘 이끌어 갔으면 좋겠습니다. 또한 같이 대학교 생활을 같이 했던 창욱이형, 군열이형, 대군이형, 동주형, 재옥이형, 현구형, 현호형, 영범이형, 정원이형 등 어린 저에게 친동생처럼 잘 대해주셔서 학교생활 잘할 수 있었어요. 고맙습니다. 조양팸으로 뭉쳤던 동우형, 준영이형, 준용이형, 진호형 형들이 있어서 학교생활 재밌게 했고 계도 하니깐 자주자주 보아요. 그리고 우리 07학번 동기들 승구, 승환이, 안성, 김성, 민수, 태훈이, 상민이, 형우, 인기, 소희, 예슬이누나, 자랑이누나 등 모든 동기들도 고마워.

마지막으로 25년동안 저를 위해 항상 희생하시는 아버지, 어머니, 누나. 지금까지 사랑한다는 말을 해본 적이 없었던거 같은데 사랑한다는 말을 전

합니다. 부산에서 6년간 생활하는 동안 저를 많이 챙겨주신 작은아버지, 작은어머니 정말 감사합니다. 꼭 보답하겠습니다.

많은 분들의 관심과 격려에 힘입어 끝이 아닌 새로운 시작을 하려 합니다. 실망시키지 않고 기대에 보답하기 위해 항상 노력하고 열심히 하는 병수가 되겠습니다.