J. Korean Soc. Aeronaut. Space Sci. 49(4), 335-342(2021)
DOI:https://doi.org/10.5139/JKSAS.2021.49.4.335
ISSN 1225-1348(print), 2287-6871(online)

HLS를 이용한 텔레메트리 표준 106-17 LDPC 복호기 설계

구영모[1], 김성종[2], 김복기[3]

Telemetry Standard 106-17 LDPC Decoder Design Using HLS

Young Mo Gu[1], Seongjong Kim[2] and Bokki Kim[3]
Inha Technical College[1], DANAM SYSTEMS[2,3]

ABSTRACT

By using HLS when developing a communication system FPGA, HDL code can be automatically generated from a little modified C/C++ source code used for performance verification, which has the advantage of shortening the development period. In this paper, a method of designing a telemetry standard 106-17 LDPC decoder in C language is proposed using Xilinx's Vivado HLS, and by synthesizing Spartan-7 and Kintex-7 as target devices, throughput and FPGA utilization rate was compared.

초 록

통신 시스템 FPGA 개발 시 HLS를 이용하면 성능 검증용 C/C++ 소스 코드를 일부 수정하여 자동으로 HDL 코드를 생성할 수 있으므로 개발 기간을 단축할 수 있는 장점이 있다. 본 논문에서는 텔레메트리 표준 106-17 LDPC 복호기를 Xilinx사의 Vivado HLS를 이용하여 C언어로 설계하는 방법을 제시하였고, Spartan-7와 Kintex-7 디바이스를 타겟으로 합성하여 throughput과 FPGA 이용률을 비교하였다.

Key Words : HLS(고급 합성), FPGA, Telemetry(텔레메트리), LDPC(저밀도 패리티 검사 부호), Decoder(복호기)

Ⅰ. 서 론

Low Density Parity Check (LDPC) 부호[1,2]는 패리티 검사 행렬의 1의 밀도가 매우 낮은 선형 부호의 일종으로 우주 통신을 위한 Consultative Committee for Space Data Systems (CCSDS)[3], 텔레메트리 표준 106-17[4]을 포함한 다양한 방송 및 통신 분야의 표준[5,6]에서 오류 정정 부호로 채택되었다.

통신 시스템을 Field Programmable Gate Array (FPGA)를 이용하여 하드웨어로 구현할 때는 먼저 Matlab이나 C/C++ 언어로 구현한 컴퓨터 시뮬레이터를 이용하여 시스템 성능 검증을 한 후 이를 VHDL이나 Verilog와 같은 Hardware Design Language (HDL)로 변경하고 이를 FPGA 디바이스에 맞게 합성하여 구현한다. 그런데 High-Level Synthesis (HLS)를 이용하면 성능 검증 소스 코드를 일부 수정하여 자동으로 HDL 코드를 생성하므로 HDL 설계 시 필요한 복잡한 타이밍이나 제어를 고려하지 않아도 되고, 하드웨어 구조를 변경하는 것도 성능 검증 소스 코드를 일부 수정하면 되므로 개발 기간을 단축할 수 있는 것이 HLS를 이용한 시스템 설계의 장점이다[7].

본 논문에서는 Xilinx사의 Vivado HLS를 이용하

† Received : October 27, 2020 Revised : February 1, 2021 Accepted : February 23, 2021
[1] Professor, [2] Senior Research Engineer, [3] Director
[1] Corresponding author, E-mail : ymgu@inhatc.ac.kr, ORCID 0000-0002-3605-783X

여 C 언어로 텔레메트리 표준 106-17 LDPC 복호기를 설계하는 방법을 제시하고, 이를 Xilinx사의 Spartan-7과 Kintex-7를 타겟으로 합성하여 그 결과를 비교한다.

II. HLS를 이용한 LDPC 복호기 설계

2.1 LDPC 부호와 복호 알고리듬

LDPC 부호는 선형 오류정정부호의 일종으로 패리티 검사 행렬 H에 의해 정의되는데 부호 벡터를 c=[c_0, c_1, ..., c_{n-1}]라고 하면 식 (1)의 관계가 성립한다. 식 (1)에서 c^T는 행렬 c의 전치행렬(transposed matrix)를 의미한다.

$$Hc^T = 0 \tag{1}$$

간단한 4 x 8 패리티 검사 행렬을 예로 들면 식 (1)을 식 (2)와 같이 다시 쓸 수 있고 이를 전개하면 식 (3)과 같이 4개의 패리티 검사 식 f_0, f_1, f_2, f_3로 정리할 수 있다.

$$\begin{bmatrix} 0&1&0&1&1&0&0&1 \\ 1&1&1&0&0&1&0&0 \\ 0&0&1&0&0&1&1&1 \\ 1&0&0&1&1&0&1&0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

$$\begin{aligned} f_0 &: c_1 + c_3 + c_4 + c_7 = 0 \\ f_1 &: c_0 + c_1 + c_2 + c_5 = 0 \\ f_2 &: c_2 + c_5 + c_6 + c_7 = 0 \\ f_3 &: c_0 + c_3 + c_4 + c_6 = 0 \end{aligned} \tag{3}$$

식 (3)은 Fig. 1과 같은 Tanner Graph로 표현할 수 있는데[8] f_0, f_1, f_2, f_3를 check node라 하는데 패리티 검사 행렬의 행을 의미하고, c_0, c_1, ⋯, c_7을 variable node라고 하는데 열을 의미한다. 패리티 검사 행렬의 1은 Tanner graph에서 check node와 variable node가 연결되어 있음을 의미한다.

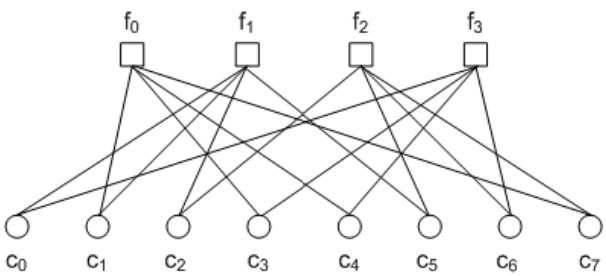

Fig. 1. Tanner graph of 4x8 parity check matrix and equations

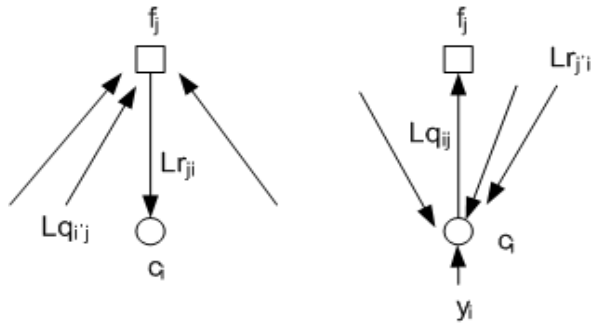

Fig. 2. LLR metric belief propagation

LDPC 부호의 복호는 check node와 variable node 간에 복호 확률을 반복적으로 주고받으면서 수행하는데 이를 iterative belief propagation이라고 한다[1]. 위와 같은 복호 방법은 확률을 사용하고 각 단계마다 확률 곱셈을 사용하므로 그 과정이 매우 복잡하다. 대신 확률을 로그 Log Likelihood Ratio (LLR) 메트릭으로 변환하면 곱셈이 덧셈으로 변환되기 때문에 복잡도가 크게 감소한다[9]. Fig. 2에서 Lq_{ij}를 variable node c_i에서 check node f_j로 전달하는 메트릭, Lr_{ji}를 check node f_j에서 variable node c_i로 전달하는 메트릭, y_i를 채널 수신 값, 채널을 Additive White Gaussian Noise(AWGN) 채널로 가정하면 복호 알고리듬은 다음과 같다.

단계1: 각 variable node c_i에서 Lq_i=y_i로 설정한다.

단계2: 각 check node f_j에서 variable node c_i를 제외한 다른 연결된 variable node들로부터 전달받은 $Lq_{i'j}$로부터 식 (4)와 같이 Lr_{ji}를 계산하여 variable node c_i에 전달한다.

$$Lr_{ji} = 2\tanh^{-1}\left(\prod \tanh\frac{Lq_{i'j}}{2}\right) \tag{4}$$

단계3: 각 variable node c_i에서 연결된 check node들로부터 전달받은 $Lr_{j'i}$로 식 (5)와 같이 LQ_i를 계산하여 LQ_i>0이면 c_i를 0으로 판정하고 LQ_i<0이면 c_i를 1로 판정한다. 판정한 c_i로 모든 패리티 검사 조건을 만족하면 복호를 정지하고 그렇지 않으면 각 variable node c_i에서 check node f_j를 제외한 다른 check node들로부터 전달받은 $Lr_{j'i}$로부터 식 (6)과 같이 Lq_{ij}를 계산하여 check node f_j에 전달하고 단계2와 단계3을 반복한다.

$$LQ_i = y_i + \sum Lr_{ji} \tag{5}$$

$$Lq_{ij} = y_i + \sum Lr_{j'i} \tag{6}$$

식 (4)의 계산을 단순화하는 방법에 따라 다양한 sub-optimal 알고리듬이 존재하는데 본 논문에서는 식 (7)의 식을 이용하는 normalized min-sum 알고리듬을 적용한다[10]. 식 (7)에서 α는 1보다 작은

normalizing factor이고 sign(x), min(x), |x|는 각각 x의 부호, 최소값, 절대값을 계산하는 함수이다.

$$Lr_{ji} = \alpha \prod sign(Lq_{i'j}) \min(|Lq_{i'j}|) \quad (7)$$

2.2 텔레메트리 표준 106-17 LDPC 부호

CCSDS 규격의 LDPC 부호는 정보 비트수가 각각 1,024, 4,096, 16,384의 세 가지가 있는데 텔레메트리 표준 106-17 규격은 이 중에서 정보 비트수가 1,024, 4,096인 LDPC 부호만을 규격으로 채택하고 있다. 정보 비트 수 K와 부호율 r에 따른 부호 비트 수 N은 Table 1과 같다.

텔레메트리 표준 106-17 규격의 LDPC 부호도 식 (1)을 만족하는 1의 밀도가 매우 낮은 패리티 검사 행렬 H에 의해 정의되는데 Fig. 3은 N=2,048, K=1,024, 부호율 1/2인 패리티 검사 행렬이다. 빗금 친 실선 부분은 행렬 원소 값이 1이고 여백은 0이다.

부호율을 r=B/(B+2) (B=2,4,8)라고 할 때, 패리티 검사 행렬 H는 다시 크기가 MxM인 부행렬(sub-matrix)로 구성되며 그 크기는 3Mx(B+3)M이다. 정보 비트수 K와 부호율에 따른 부행렬 크기 M은 Table 2와 같다. Fig. 3의 패리티 검사 행렬은 크기가 512x512인 부행렬이 세로축으로 3개, 가로축으로 5개 (B=2)로 구성되어 있고 이를 식으로 표현하면 식 (8)과 같다. 식 (8)에서 I_M은 크기가 MxM인 단위행렬이고 0_M은 0 행렬이다. Π_k는 permutation 행렬인데 i 행과 $\pi_k(i)$ 열만 1이고 나머지는 0인 행렬이다. $\pi_k(i)$

Table 1. Telemetry Standard 106-17 LDPC Code length N per information length K

K	N		
	r=1/2	r=2/3	r=4/5
1,024	2,048	1,536	1,280
4,096	8,192	6,144	5,120

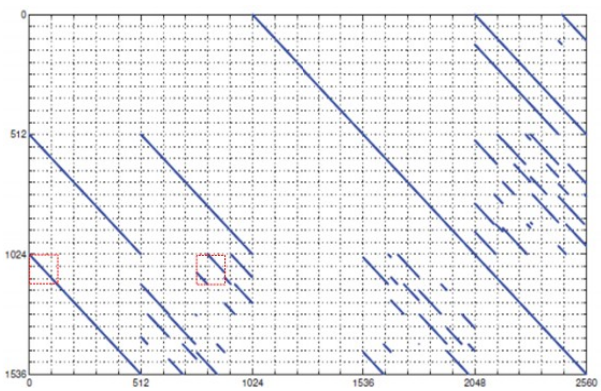

Fig. 3. Parity check matrix H for N=2,048, K=1,024, rate r=1/2 [3]

Table 2. Sub-matrix size M per information length K and code rate r

K	M		
	r=1/2	r=2/3	r=4/5
1,024	512	256	128
4,096	2,048	1,024	512

는 식 (9)와 같다. 식 (9)에서 ⌊x⌋는 x보다 작은 최대 정수이고 %는 modular 연산이다. (부호율 2/3, 4/5의 패리티 검사 행렬 $H_{2/3}$, $H_{4/5}$ 및 Θ_k와 ϕ_k 값은 [3,4] 참조)

$$H_{1/2} = \begin{pmatrix} 0_M & 0_M & I_M & 0_M & I_M \oplus \Pi_1 \\ I_M & I_M & 0_M & I_M & \Pi_2 \oplus \Pi_3 \oplus \Pi_4 \\ 0_M & \Pi_2 \oplus \Pi_3 & 0_M & \Pi_7 \oplus \Pi_8 & I_M \end{pmatrix} \quad (8)$$

$$\pi_k(i) = \frac{M}{4}\left((\theta_k + \lfloor 4i/M \rfloor)\%4\right) + \left(\phi_k(\lfloor 4i/M \rfloor + i)\right)\%\frac{M}{4} \quad (9)$$

CCSDS 규격 LDPC 부호의 특이한 점은 Table 1, 2에 의하면 정보 비트 수 K가 BM, 부호 비트 수 N이 (B+2)M이므로 식 (1)이 성립하려면 행렬 H의 크기는 3Mx(B+2)M이 되어야 하는데 실제 크기는 3Mx(B+3)M인 것이다. 그 이유는 원래 부호기 출력의 크기는 (B+3)M인데 송신기는 그 중에서 M 비트는 puncturing하여 송신하지 않고 (B+2)M 비트만 송신하기 때문이다[7]. 따라서 복호기 설계 시 이 점을 고려하여 단계1에서 puncturing된 c_i의 Lq_i는 0으로 초기화해야 한다. Fig. 3에서 M이 512인 패리티 검사 행렬의 크기가 1,536x2,048이 아니고 1,536x2,560인 이유는 송신기에서 부호기 출력 2,560 비트 중에서 512 비트는 puncturing하여 송신하지 않기 때문이다.

2.3 복호 알고리듬 성능 평가

LDPC 복호기는 단계2, 3이 반복적으로 동작하므로 단계2에서 각 check node의 Lr_{ji}를 계산하려면 Lq_{ij}가 필요하고, 단계3에서 Lq_{ij}을 계산하려면 단계2에서 계산된 Lr_{ji}가 필요하므로 Lr_{ji}와 Lq_{ij}를 모두 저장해야 한다. 그런데 Fig. 1의 Tanner graph의 check node와 variable node 간의 연결, 즉 패리티 검사 행렬의 1의 개수 만큼에 해당하는 Lr_{ji}와 Lq_{ij}를 모두 저장해야 하는데 대신 check node에서는 Lr_{ji}를, variable node에서는 LQ_i만을 저장하고, Lq_{ij}는 따로 저장하지 않고 식 (5), (6)을 이용하여 아래 식 (10)과 같이 계산하여 사용할 수 있다.

$$Lq_{ij} = LQ_i - Lr_{ji} \quad (10)$$

또한 단계2에서 모든 check node가 동시에 Lr_{ji}를 계산하여 variable node에 전달하고, 단계3에서 모든 variable node가 동시에 Lq_{ij}을 계산하여 check node

에 전달하여야 하지만 패리티 검사 행렬의 크기가 큰 경우 하드웨어 복잡도가 매우 크기 때문에 동시에 계산하는 대신 단계마다 순차적으로 계산하여 전달하는 방식으로 구현하는데 이를 layered decoding 이라고 하며 구현 복잡도가 크게 감소하고 수렴 속도가 빨라지는 장점이 있다[11]. 이를 정리하면 아래와 같다.

단계1: 각 variable node c_i에서 $LQ_i=y_i$, 각 check node에서 $Lr_{ji}=0$로 초기값을 설정한다.

단계2: 기존의 단계2와 단계3을 통합하여 각 check node에서 순차적으로 식 (10)을 이용하여 Lq_{ij}를 계산하고 식 (7)을 이용하여 새로 Lr_{ji}를 계산하여 기존 값을 갱신한다. 또한 이를 check node에 연결된 variable node에 전달하면 각 variable node는 전달받은 Lr_{ji}를 이용하여 아래 식 (11)과 같이 LQ_i를 갱신한다. 마지막 check node까지 계산이 끝나면 동시에 모든 variable node에서 LQ_i의 계산도 끝나므로 $LQ_i>0$이면 c_i를 0으로 판정하고, $LQ_i<0$이면 c_i를 1로 판정한다. 판정한 c_i로부터 모든 패리티 검사 조건을 만족하면 복호를 정지하고, 아니면 단계2를 반복한다. 단계1과 2를 정리한 복호기 구조는 Fig. 4와 같다.

$$LQ_i = Lq_{ij} + Lr_{ji} \tag{11}$$

위에서 설명한 layered decoding 알고리듬을 적용한 부호율 r과 정보 비트 수 K에 따른 fixed-point 시뮬레이션을 통한 복호 알고리듬 성능 평가 결과는 Fig. 5와 같다. 식 (7)의 normalizing factor는 0.75, 채널 입력 y_i의 비트 수는 6비트, LQ_i, Lr_{ji}, Lq_{ij}의 비트 수는 모두 9비트, 최대 반복 횟수(iteration number)는 20으로 하였다. 참고문헌 [3]의 성능 평가 결과(Fig. B-2, Fig. B-3)와 비교하면 경향은 전반적으로 비슷하지만 BER(bit error rate)이 10^{-6}일 때 약 0.2~0.3dB의 비트 에너지 대 잡음비(Eb/No) 성능 차이가 있다.

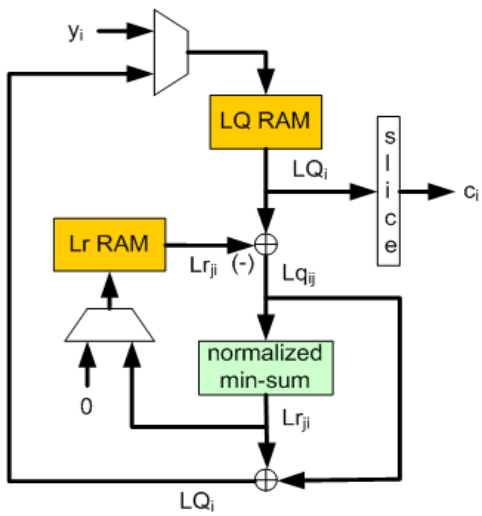

Fig. 4. Layered LDPC decoder structure

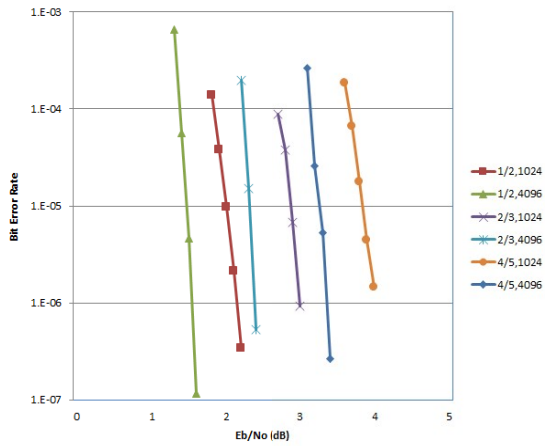

Fig. 5. Layered LDPC decoder performance

2.4 Layered LDPC 복호기 설계

통상적으로 FPGA를 이용한 하드웨어 구현은 VHDL이나 Verilog와 같은 HDL을 이용하여 설계하는데 Xilinx 사의 Vivado HLS는 C/C++소스 코드로부터 자동으로 HDL 코드를 생성하므로 복잡한 타이밍이나 제어부를 고려하지 않아도 되는 편리함이 있고, 하드웨어 구조를 쉽게 변경할 수 있다.

Figure 6은 Fig. 4의 layered LDPC 복호기의 구현 C 코드이다. 정수형 ap_int<6>, ap_int<9>는 각각 6비트, 9비트의 signed 정수를 의미하며 ap_uint<1>은 1비트의 unsigned 정수이다. 배열(array) edge_n은 check node에 연결된 variable node의 수를 저장하며, VN_NUM(별도의 헤더 파일에 선언)은 그 최대값으로

```
void decoder(ap_int<6> in[N], ap_uint<1> out[K]) {
    int i, j, itr, Q;
    int edge_n[3] = {3,VN_NUM,VN_NUM};
    int pc_table[VN_NUM];
    ap_int<9> LQ[(B+3)*M], Lr[3*M*VN_NUM], Lq[VN_NUM], Lrj[VN_NUM];

    void pc_table_calc(int, int *);
    void normalized_min_sum(ap_int<9> *, ap_int<9> *);

    l_LQ_init1: for(i=0;i<N;i++) LQ[i] = (ap_int<9>)in[i]; // LQi=yi, pipeline
    l_LQ_init2: for(i=0;i<M;i++) LQ[N+i] = 0; // pipeline
    l_Lr_init1: for(j=0;j<(3*M);j++) { // pipeline
        l_Lr_init2: for(i=0;i<VN_NUM;i++) Lr[VN_NUM*j+i] = 0; // Lrji=0, pipeline
    }

    l_itr: for(itr=0;itr<20;itr++) {
        l_p: for(j=0;j<(3*M);j++) { // pipeline
            Q = edge_n[(int)(j/M)];
            pc_table_calc(j, pc_table);
            for(i=0;i<VN_NUM;i++) {
                if(i<Q) {
                    Lq[i] = LQ[pc_table[i]] - Lr[VN_NUM*j+i]; // eq. (10)
                }
                else {
                    Lq[i] = 255;
                }
            }
            normalized_min_sum(Lq, Lrj); // eq. (7)
            for(i=0;i<VN_NUM;i++) {
                Lr[VN_NUM*j+i] = Lrj[i];
                if(i<Q) {
                    LQ[pc_table[i]] = Lq[i]+Lrj[i]; // eq. (11)
                }
            }
        }
    }
    l_out: for(i=0;i<K;i++) { // pipeline
        if(LQ[i]>0) out[i] = 0;
        else out[i] = 1;
    }
}
```

Fig. 6. HLS C source code for layered LDPC decoder

부호율 1/2, 2/3, 4/5에 따라 각각 6, 10, 18이다[4]. Fig. 3을 보면 처음 512개의 check node에 연결된 variable node 수는 3이고 나머지 1,024개에는 6임을 확인할 수 있다. 배열 pc_table은 check node에 연결된 variable node의 위치를 함수 pc_table_calc()에서 식 (9)에 의해 계산하여 저장한다. 배열 LQ, Lr, Lq는 각각 식 (10), (11)의 LQ_i, Lr_{ji}, Lq_{ij}를 저장하며, 배열 Lrj는 식 (7)에 의해 함수 normalized_min_sum()에서 계산된 Lr_{ji}를 저장한다.

Figure 7은 식 (7)을 이용하여 배열 Lq로부터 배열 Lrj를 계산하는 함수 normalized_min_sum()의 C 코드이다. 함수 SGN(), MIN(), iabs()는 각각 부호, 최소값, 절대값을 계산하는 함수이다. Normalizing factor α 값은 0.75로 하였다.

Vivado HLS는 합성을 최적화하는 다양한 directive를 제공한다[12]. PIPELINE directive는 for loop를 pipeline으로 구현하여 지연(delay)을 최소화하는데 이를 Fig. 6에서 주석으로 표시하였다.

Figure 6의 C 소스 코드를 부호율 r이 1/2, 정보 비트 수 K가 1,024일 때 Xilinx spartan7 xc7s100 디바이스를 타겟으로 합성한 결과는 Table 3과 같다. 클럭 주기 타겟을 각각 10ns, 5ns, 2.5ns로 설정하여 합성을 한 결과, 합성 클럭 주기 값이 각각 8.742ns, 4.328ns, 2.771ns로 합성되었다. 지연(latency) 값은 최대 반복 횟수가 20일 때의 값이다. K 출력 비트를 처리하는데 걸린 시간은 1 클럭 주기와 latency를 곱한 값이므로 throughput을 계산하는 식은 아래 식 (12)와 같다.

$$Throughput = \frac{\text{출력 비트수}}{\text{클럭주기} \times latency} \qquad (12)$$

입력 1,024 비트에 대한 throughput을 계산하면 각각 338.4Kbps, 473.5Kbps, 458.0Kbps이며 합성 시 클럭 주기 타겟 설정을 작게 변경하여 throughput을 높일 수 있는데 2.5ns의 경우 합성된 클럭 주기는 5ns의 경우보다 감소하지만 2.5ns 타겟을 맞추기 위

```
void  normalized_min_sum(ap_int<9> Lq[VN_NUM], ap_int<9> Lrj[VN_NUM]) {
    int i, k, ss, sum;
    ap_int<9> min, tmp;

    int SGN(ap_int<9>);
    ap_int<9> MIN(ap_int<9> , ap_int<9>);
    ap_int<9> iabs(ap_int<9>);

    for(i=0;i<VN_NUM;i++) {
        ss = 1;
        min = 255;
        for(k=0;k<VN_NUM;k++) {
            if(k!=i) {
                ss *= SGN(Lq[k]);
                min = MIN(min, iabs(Lq[k]));
            }
        }
        tmp = ss*min;
        Lrj[i] = (tmp/2)+(tmp/4); // Lrji *=alpha, alpha=0.75
    }
}
```

Fig. 7. HLS C source code for sub-routine normalized_min_sum()

Table 3. HLS Syntheis Results of C source code of Fig. 6

Target	10ns	5ns	2.5ns
Synthesized clock period	8.742ns	4.328ns	2.771ns
latency	346,126	499,729	806,934
Throughput (Kbps)	338.4	473.5	458.0
BRAM_18K (240)	12 (5%)	12 (5%)	12 (5%)
FF (128,000)	897 (0%)	1,568 (1%)	2,328 (1%)
LUT (64,000)	3,170 (5%)	3,420 (5%)	3,354 (5%)

해 critical path에 플립플롭(FF)을 다수 삽입하여 합성하므로 latency는 오히려 크게 증가하여 throughput은 5ns의 경우보다 낮다. BRAM_18K는 블록 램으로 Fig. 4의 LQ RAM과 Lr RAM 즉, Fig. 6의 배열 LQ와 Lr을 위해 사용되는 메모리이다. 플립플롭은 주로 배열 edge_n, Lq, Lrj, pc_table을 위해 사용된다. Look-Up Table (LUT)은 주로 덧셈 및 뺄셈 연산 및 함수 pc_table_calc(), normalized_min_sum()의 구현에 사용된다. 하드웨어 이용률은 전체적으로 5% 이하로 매우 낮지만 throughput도 500Kbps 이하로 낮아서 이를 높이려면 구조 변경이 필요하다.

2.5 Block Layered LDPC 복호기 설계

Layered LDPC 복호기는 check node 한 개씩 순차적으로 동작하므로 간단한 하드웨어로 구현 가능하지만 throughput도 매우 낮다. 이를 높이려면 여러 check node를 묶어서 동시에 처리해야 하는데 이를 block layered 복호기라 한다[13]. Fig. 3에서 보듯이 패리티 검사 행렬 H는 다시 크기가 L(=M/4)xL인 단위행렬 또는 단위행렬이 cyclic하게 right-shift된 행렬(붉은색 사각형)들로 구성된 quasi-cyclic한 구조로 볼 수 있으며[14] 따라서 세로 12개, 가로 4(B+3)개의 sub-matrix로 구성되어 있다. 예를 들면 Fig. 3의 패리티 검사 행렬은 크기가 128x128인 sub-matrix들이 세로 12개, 가로 20개로 구성되어 있는 구조로 볼 수 있다. 따라서 L개의 check node와 variable node들을 묶어서 동시에 처리하면 throughput은 layered LDPC 복호기보다 L배 증가시킬 수 있다. Fig. 8은 block layered LDPC 복호기 구조이다.

Figure 4의 layered LDPC 복호기 구조와의 차이점은 LQ RAM과 Lr RAM을 읽을 때 L개씩 묶어서 동시에 읽는 것과 cyclic하게 right-shift된 행렬에 해당하는 LQ RAM 데이터를 읽은 후에는 이를 단위행렬

에 해당하는 LQ RAM 데이터와 정렬시키기 위해 Fig. 9와 같이 left-shift한 후에 L개씩 묶어서 식 (10)과 (11)의 연산을 수행한 후 LQ RAM에 쓸 때에는 이를 다시 원래대로 right-shift 해야 하는 것이다. 이를 위해서 left-shifter와 right-shifter가 추가되었다. 식 (7)의 normalized min-sum 연산을 수행하는 Check Node Unit(CNU)도 L개가 필요한데 이 경우 FPGA 이용률이 매우 크게 증가하므로 그 수를 L보다 작은 CNU_NUM으로 제한하여 공유해야 한다. 따라서 throughput은 CNU_NUM의 값에 따라 증가하는 함수가 된다.

Figure 10은 Fig. 8의 block layered LDPC 복호기의 구현 C 코드이다. 복호기는 9비트 메트릭을 L개씩 묶어서 처리하므로 정수형 ap_int<W>에서 W는 9L이다. 함수 left_shifter()와 right_shift()는 각각 Fig. 8의 left shifter와 right shifter를 배럴(barrel) 시프트 구조로 구현한 것이다. Vivado HLS는 변수에서 특정 비트 영역을 선택할 수 있는 함수를 제공하는데 함수 range(x,y)는 변수의 MSB(most significant bit) x에서 y까지의 비트 영역을 선택한다. UNROLL

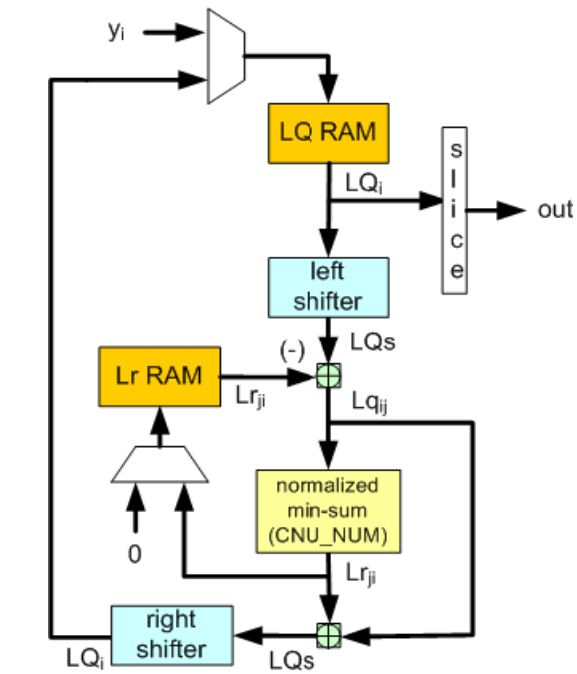

Fig. 8. Block layered LDPC decoder structure

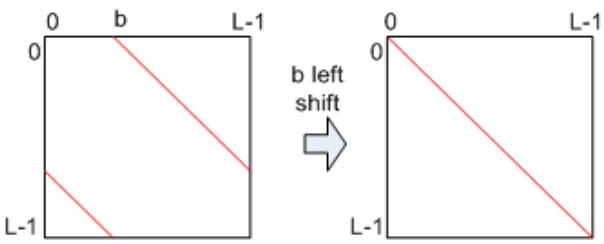

Fig. 9. Cyclically left-shifted sub-matrix of size L

```
void decoder(ap_uint<W> in[4*(B+2)], ap_uint<1> out[K]) {
    int itr, m, n, p, q, Q, j;
    ap_uint<W> LQ[4*(B+3)], Lr[12*VN_NUM], LQs[VN_NUM];
    int edge_n[3] = {3,VN_NUM,VN_NUM};
    int pc_table[VN_NUM];
    ap_int<9> Lq[VN_NUM*L], Lrj[VN_NUM*L], tmp;

    void pc_table_calc(int, int *);
    void left_shifter(ap_uint<W> *, int *, ap_uint<W> *);
    void right_shifter(ap_uint<W> *, int *, ap_uint<W> *);
    void normalized_min_sum(ap_int<9> *, ap_int<9> *);

    l_LQ_init1: for(m=0;m<4*(B+2);m++) LQ[m] = in[m]; // pipeline
    l_LQ_init2: for(m=0;m<4;m++) LQ[4*(B+2)+m] = 0; // pipeline
    l_Lr_init: for(m=0;m<12*VN_NUM;m++) Lr[m] = 0; // pipeline
    l_itr: for(itr=0;itr<20;itr++) {
        l_p: for(p=0;p<12;p++) {
            Q = edge_n[(int)(p/4)];
            pc_table_calc(p, pc_table);
            left_shifter(LQ, pc_table, LQs);
            l_Lq: for(q=0;q<VN_NUM;q++) { // pipeline
                if(q<Q) {
                    l_Lq1: for(n=0;n<L;n++) { // unroll
                        Lq[L*q+n] = (ap_int<9>)LQs[q].range(W-1-9*n, W-9-9*n) // eq. (10)
                                  - (ap_int<9>)Lr[VN_NUM*p+q].range(W-1-9*n,W-9-9*n);
                    }
                }
                else l_Lq2: for(n=0;n<L;n++) Lq[L*q+n] = 255; // unroll
            }
            normalized_min_sum(Lq, Lrj); // eq. (7)
            l_LQ: for(q=0;q<VN_NUM;q++) { // pipeline
                l_LQ1: for(n=0;n<L;n++) Lr[VN_NUM*p+q].range(W-1-9*n, W-9-9*n) = Lrj[L*q+n];
                if(q<Q) {
                    l_LQ2: for(n=0;n<L;n++) { // unroll
                        LQs[q].range(W-1-9*n, W-9-9*n) = Lq[L*q+n]+Lrj[L*q+n]; // eq. (11)
                    }
                }
            }
            right_shifter(LQs, pc_table, LQ);
        }
    }
    l_out: for(m=0;m<(K/L);m++) { // pipeline
        l_out1: for(n=0;n<L;n++) { // pipeline
            tmp = (ap_int<9>)LQ[m].range(W-1-9*n,W-9-9*n);
            if(tmp>0) out[L*m+n] = 0; else out[L*m+n] = 1;
        }
    }
}
```

Fig. 10. HLS C source code for block layered LDPC decoder of Fig. 8

directive[12]는 for loop를 순차적으로 구현하지 않고 동시에 병렬 처리하게 구현하도록 하는데 이를 Fig. 10에서 주석으로 표시하였다.

Figure 11은 함수 normalized_min_sum()의 C 코드인데 FPGA 이용률을 고려하여 합성 시 L개의 CNU를 모두 병렬로 구현하지 않고 두 개의 for 루프를 이용하여 CNU_NUM (별도의 헤더 파일에 선언) 개만 UNROLL directive를 사용하여 병렬로 구현하고 pipeline directive를 이용하여 이를 순차적으로 (L/CNU_NUM)번 공유하도록 코딩한 것이다.

Table 4는 Fig. 6의 C 소스 코드를 부호율 r이 1/2, 정보 비트 수 K가 1024일 때 Xilinx spartan-7

```
void  normalized_min_sum(ap_int<9> Lq[VN_NUM*L], ap_int<9> Lrj[VN_NUM*L]) {
    int i, k, m, n, ss;
    ap_int<9> min, tmp;

    int SGN(ap_int<9>);
    ap_int<9> MIN(ap_int<9> , ap_int<9>);
    ap_int<9> iabs(ap_int<9>);

    l_cnu_m: for(m=0;m<(L/CNU_NUM);m++) { // pipeline
        l_cnu_n: for(n=0;n<CNU_NUM;n++) { // unroll
            for(i=0;i<VN_NUM;i++) {
                ss = 1;
                min = 255;
                for(k=0;k<VN_NUM;k++) {
                    if(k!=i) {
                        ss *= SGN(Lq[L*k+CNU_NUM*m+n]);
                        min = MIN(min, iabs(Lq[L*k+CNU_NUM*m+n]));
                    }
                }
                tmp = ss*min;
                Lrj[L*i+CNU_NUM*m+n] = (tmp/2)+(tmp/4);
            }
        }
    }
}
```

Fig. 11. HLS C source code for sub-routine normalized_min_sum() of Fig. 10

Table 4. HLS synthesis results of C source code of Fig. 10 using Xilinx spartan-7 (r=1/2, K=1,024)

CNU_NUM	1	2	4
Synthesized clock period	4.355ns	4.355ns	4.355ns
latency	45,255	30,375	22,695
Throughput (Mbps)	5.20	7.74	10.36
BRAM_18K (240)	96 (40%)	96 (40%)	96 (40%)
FF (128,000)	43,024 (34%)	52,263 (41%)	55,129 (43%)
LUT (64,000)	38,259 (58%)	45,011 (70%)	53,363 (83%)

xc7s100 디바이스를 타겟으로 합성한 결과이다. 클럭 주기 타겟을 5ns로 설정하여 합성을 한 결과 합성 클럭 주기 값은 4.355ns로 합성되었다. Table 3과 비교하여 throughput이 크게 증가하였고, CNU_NUM 값에 비례하여 throughput과 LUT의 이용률이 증가하는 것을 볼 수 있다.

Table 5는 부호율 r이 1/2, 정보 비트 수 K가 4,096일 때 Xilinx kintex-7 xc7k325t 디바이스를 타겟으로 클럭 주기 타겟을 5ns로 설정하여 합성한 결과이다. Table 4와 비교하면 블록 크기 L이 증가한 만큼 throughput도 증가하고 FPGA 이용률도 증가한다. 따라서 spartan-7 디바이스를 타겟으로 합성하면 이용률이 100%를 넘어가기 때문에 용량이 큰 kintex-7 이용률로 변경하여 표시하였다.

Table 5. HLS synthesis results of C source code of Fig. 10 using Xilinx kintex-7 (r=1/2, K=4096)

CNU_NUM	1	2	4
Synthesized clock period	4.355ns	4.355ns	4.355ns
latency	141,111	80,151	49,431
Throughput (Mbps)	6.67	11.73	19.03
BRAM_18K (650)	384 (59%)	384 (59%)	384 (59%)
FF (407,600)	261,859 (64%)	291,434 (72%)	294,300 (72%)
LUT (203,800)	161,213 (79%)	167,514 (82%)	175,417 (86%)

III. 결 론

본 논문에서는 텔레메트리 표준 106-17 LDPC 복호기를 Xilinx사의 Vivado HLS를 이용하여 C언어로 설계하는 방법을 제시하였고, Spartan-7와 Kintex-7 디바이스를 타겟으로 합성하여 throughput과 FPGA 이용률에 따른 하드웨어 복잡도를 비교하였다. Layered LDPC 복호기는 check node 한 개씩 순차적으로 동작하므로 간단한 하드웨어로 구현 가능하지만 throughput도 매우 낮은 반면, L개의 check node를 묶어서 동시에 처리하는 block layered 복호기는 throughput이 증가하는 만큼 하드웨어 복잡도가 증가하였다. 제시한 HLS를 이용한 설계 방법은 C 소스 코드의 수정 없이 check node에서 variable node로 전달하는 메트릭을 계산하는 CNU의 개수만을 변경하여 다양한 throughput과 하드웨어 복잡도의 디바이스를 지원하는 것이 가능하다.

References

1) Gallager, R. G., “Low-density parity-check codes,” *IRE Transactions on Information Theory*, Vol. 8, January 1962, pp. 21~28.

2) MacKay, D. J. C. and Neal, R. M., “Near Shannon limit performance of low density parity check codes,” *Electronics Letters*, Vol. 32, No. 18, 1996, pp. 1645~1646.

3) CCSDS *131.1-O-2 Experimental Specification*, September 2007.

4) Telemetry Standards, *RCC Standard 106-17*, July 2017.

5) *ETSI EN 302 755 V1.3.1*, April, 2012.

6) *3GPP TS 36.201 V10.0.0*, December, 2010.

7) Gu, Y. M., Lee, W. and Kim, B., “Telemetry standard 106-17 LDPC encoder design using HLS,” *Journal of The Korean Society for Aeronautical and Space Sciences,* Vol. 48, No. 10, October 2020, pp. 831~835.

8) Tanner, R. M., “A recursive approach to low complexity codes,” *IEEE Tansactions on Information Theory*, September 1981, pp. 533~547.

9) Fossorier, M. P. C., Mihaljevic, M. and Imai, H., “Reduced complexity iterative decoding of low-density parity-check codes based on belief propagation,” *IEEE Transactions on Communications*, Vol. 47, No. 5, May 1999, pp. 673~680.

10) Chen, J. and Fossorier, M., “Density evolution of two improved bp-based algorithms for

LDPC decoding," *IEEE Communication letters*, Vol. 6, No. 5, May 2002, pp. 208~210.

11) Zhang, J. and Fossorier, M. P. C., "Shuffled iterative decoding," *IEEE Transactions on Communications*, Vol. 53, No. 2, February 2005, pp. 209~213.

12) Vivado HLS optimization methodology guide, *UG1270(v20174)*, December 2017.

13) Darabiha, A., Carusone, A. C. and Kschischang, F. R., "Block-interlaced LDPC decoders with reduced interconnect complexity," *IEEE Transactions on Circuits Systems II: Express Briefs*, Vol. 55, No. 1, January 2008, pp. 74~78.

14) Fossorier, M. P. C., "Quasicyclic low-density parity-check codes from circulant permutation matrices," *IEEE Transactions on Information Theory*, Vol. 50, No. 8, October 2006, pp. 1788~1794.