# REPORT

## 전자공학도의 윤리 강령 (IEEE Code of Ethics)

(출처: http://www.ieee.org)

나는 전자공학도로서, 전자공학이 전 세계 인류의 삶에 끼치는 심대한 영향을 인식하여 우리의 직업, 동료와 사회에 대한 나의 의무를 짐에 있어 최고의 윤리적, 전문적 행위를 수행할 것을 다짐하면서, 다음에 동의한다.

1. **공중의 안전, 건강 복리에 대한 책임**: 공중의 안전, 건강, 복리에 부합하는 결정을 할 책임을 질 것이며, 공중 또는 환경을 위협할 수 있는 요인을 신속히 공개한다.
2. **지위 남용 배제**: 실존하거나 예기되는 이해 상충을 가능한 한 피하며, 실제로 이해가 상충할 때에는 이를 이해 관련 당사자에게 알린다. (이해 상충: conflicts of interest, 공적인 지위를 사적 이익에 남용할 가능성)
3. **정직성**: 청구 또는 견적을 함에 있어 입수 가능한 자료에 근거하여 정직하고 현실적으로 한다.
4. **뇌물 수수 금지**: 어떠한 형태의 뇌물도 거절한다.
5. **기술의 영향력 이해**: 기술과 기술의 적절한 응용 및 잠재적 영향에 대한 이해를 높인다.
6. **자기계발 및 책무성**: 기술적 능력을 유지, 증진하며, 훈련 또는 경험을 통하여 자격이 있는 경우이거나 관련 한계를 전부 밝힌 뒤에만 타인을 위한 기술 업무를 수행한다.
7. **엔지니어로서의 자세**: 기술상의 업무에 대한 솔직한 비평을 구하고, 수용하고, 제공하며, 오류를 인정하고 수정하며, 타인의 기여를 적절히 인정한다.
8. **차별 안하기**: 인종, 종교, 성별, 장애, 연령, 출신국 등의 요인에 관계없이 모든 사람을 공평하게 대한다.
9. **도덕성**: 허위 또는 악의적인 행위로 타인, 타인의 재산, 명예, 또는 취업에 해를 끼치지 않는다.
10. **동료애**: 동료와 협력자가 전문분야에서 발전하도록 도우며, 이 윤리 헌장을 준수하도록 지원한다.

위 IEEE 윤리헌장 정신에 입각하여 report를 작성하였음을 서약합니다.


학 부: 전자공학과

제출일: 2021. 12. 03

과목명: 논리회로

교수명: 박성진 교수님

분 반: C108-2

학 번: 201821094

성 명: 안재혁

# [1] 문제 설명

• 설계 목적 : Hamming Encoder와 Bit Converter를 통해, 1-bit error가 발생한 9-bit Codeword 입력을 4-bit Unsigned Binary Number 출력으로 변환하는 Finger-counting Converter를 설계한다.

• 설계 순서 :

① 5-bit Hand Sign이 미리 Hamming Encoder로 Encoding 된 9-bit Hamming Codeword(with 1-bit error)를 입력으로 받는다.

② 신호로 받은 9-bit Codeword를 Hamming Decoder로 입력받은 후, Decoding 한다.

③ Hamming Decoder에서 Error Correcting 된 5-bit Hand Sign을 Unsigned Binary Converter에 입력으로 넣는다.

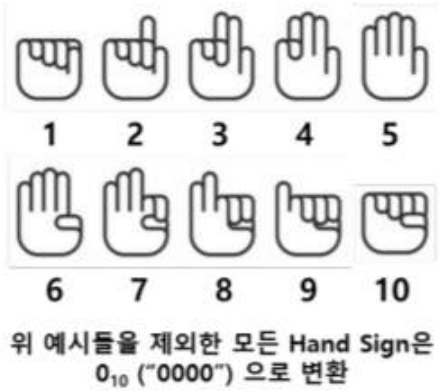
④ Converter를 통해 입력으로 받은 5-bit Hand Sign을 4-bit Unsigned Binary Number로 변환한다.

• 5-bit Hand Sign

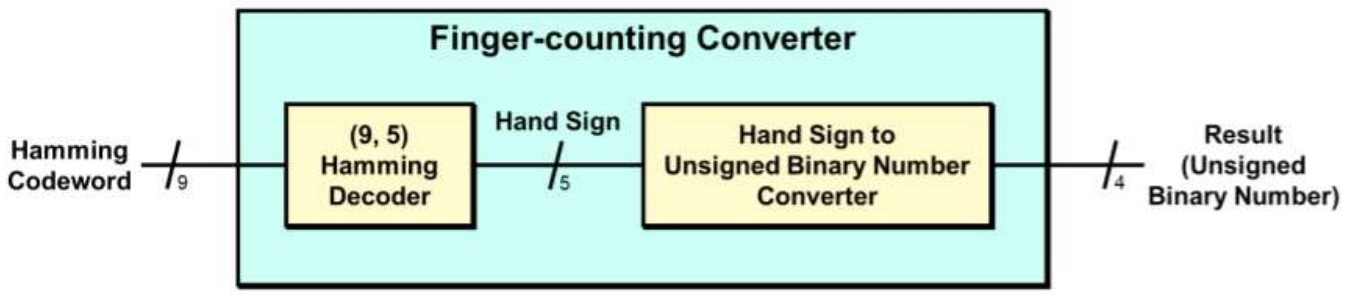
각 손가락마다 1-bit씩 H[4], H[3], H[2], H[1], H[0]을 정해놓는다(약지부터 오른쪽으로). 값은 손가락이 접혔을 경우(Folded) '0'으로, 펴있을 경우(Unfolded) '1'으로 간주한다.

오른쪽 그림과 같이, 각 손가락 상태에 따라 번호가 정해져 있고 10가지를 제외한 나머지 경우는 0("0000")으로 간주한다.

ex) 2번의 경우는 H[1] = 1, H[0] = 1이므로 5-bit hand sign으로 표현하면 "00011"이 되고, converter를 통해 2(10진수)를 나타내는 "0010"으로 변환되어야 한다.




• 회로도(Schematic)




※ (9,5) Hamming Codeword는 9-bit의 Hamming Codeword와 5-bit의 Information Bits(Hand sign)을 나타낸다.

※ 9-bit의 Hamming Codeword는 5-bit의 Information Bits와 4-bit의 Parity Bits로 이루어져 있다.

※ Binary Number는 "MSB, … , LSB"로 통일한다.

## [2] Code 분석

### 1) A : (9,5) Hamming Decoder

```
module a_hamdec(cw, hs); //Error correcting, Decoding -> 5 bit original hand sign
        input [8:0] cw;
        output [4:0] hs;

        // fill in your implementation below
reg [4:0] infor; // information bits
reg [3:0] par; // parity bits
reg [3:0] even_par; // put the compared bits using the even parity codes
reg [3:0] syndrome; // syndrome

always @ (*) begin
infor = { cw[8], cw[6:4], cw[2] }; // put information bits
par = { cw[7], cw[3], cw[1:0] }; // put parity bits

//(A xor B xor C xor D)
even_par[0] = infor[4]^infor[3]^infor[1]^infor[0]; // p1
//(A xor B)C'+(AB+A'B')C
even_par[1] = (infor[3]^infor[2])&(~infor[0])|(((infor[3]&infor[2])|((~infor[3])&(~infor[2])))&infor[0]);
 // p2
even_par[2] = (infor[3]^infor[2])&(~infor[1])|(((infor[3]&infor[2])|((~infor[3])&(~infor[2])))&infor[1]);
 // p3
even_par[3] = infor[4]; // p4

syndrome[0] = even_par[0]^par[0]; // syndrome code(A'B+AB'=A xor B)
syndrome[1] = even_par[1]^par[1];
syndrome[2] = even_par[2]^par[2];
syndrome[3] = even_par[3]^par[3];

// put the corrected information bits
if (syndrome == 4'd3) begin // syndrome == 3
infor = { cw[8], cw[6:4], (~cw[2]) }; end
else if (syndrome == 4'd5) begin // syndrome == 5
infor = { cw[8], cw[6], cw[5], (~cw[4]), cw[2] }; end
else if (syndrome == 4'd6) begin // syndrome == 6
infor = { cw[8], cw[6], (~cw[5]), cw[4], cw[2] }; end
else if (syndrome == 4'd7) begin // syndrome == 7
infor = { cw[8], (~cw[6]), cw[5], cw[4], cw[2] }; end
else if (syndrome == 4'd9) begin // syndrome == 9
infor = { (~cw[8]), cw[6:4], cw[2] }; end

end
assign hs = infor[4:0]; // assign the original hand sign

endmodule
```

• 원리

(9,5) Hamming Decoder는 syndrome을 이용하여 1-bit error가 발생한 곳을 찾아 error correcting 하고, 5-bit의 information bits를 출력으로 내보낸다.

• 순서

① 입력으로 받은 9-bit codeword(1-bit error 발생)를 나누어, reg [4:0] infor에 information 5-bits를 저장하고 reg [3:0] par에 parity 4-bits를 저장한다.

② 입력받은 information bits를 이용하여 다시 parity bits를 구하고, reg [3:0] even_par에 각각 하

나의 bit씩(p1, p2, p3, p4) 넣는다.
③ [3:0] even_par와 [3:0] par를 한 bit씩 비교하여, reg [3:0] syndrome에 같은 bit일 경우는 '0'을 넣고 다른 bit일 경우는 '1'을 넣는다.(XOR gate 사용)
④ [3:0] syndrome에 따라 나눈 if문을 통해 error가 발생한 bit를 수정하여 [3:0] infor에 넣는다.
⑤ 수정된 [4:0] infor를 [4:0] hs에 대입하고 출력으로 내보낸다.

• 구조 및 동작

```
        input [8:0] cw;
        output [4:0] hs;

        // fill in your implementation below
reg [4:0] infor; // information bits
reg [3:0] par; // parity bits
reg [3:0] even_par; // put the compared bits using the even parity codes
reg [3:0] syndrome; // syndrome
```

=>

| | |
|---|---|
| [8:0] cw | 입력 input |
| [4:0] hs | 출력 output |
| [4:0] infor | information bits 저장할 공간 |
| [3:0] par | parity bits 저장할 공간 |
| [3:0] even_par | 입력받은 information bits로 다시 구한 parity bits를 저장할 공간 |
| [3:0] syndrome | 2진수의 syndrome을 저장할 공간 |

```
always @ (*) begin
infor = { cw[8], cw[6:4], cw[2] }; // put information bits
par = { cw[7], cw[3], cw[1:0] }; // put parity bits

//(A xor B xor C xor D)
even_par[0] = infor[4]^infor[3]^infor[1]^infor[0]; // p1
//(A xor B)C'+(AB+A'B')C
even_par[1] = (infor[3]^infor[2])&(~infor[0])|(((infor[3]&infor[2])|((~infor[3])&(~infor[2])))&infor[0]);
// p2
even_par[2] = (infor[3]^infor[2])&(~infor[1])|(((infor[3]&infor[2])|((~infor[3])&(~infor[2])))&infor[1]);
// p3
even_par[3] = infor[4]; // p4
```

=> 입력으로 받은 9-bit codeword를 [4:0] infor와 [3:0] par에 나누어 저장한다.
[3:0] even_par에는 parity bits를 찾는 원리를 이용하여 나타낸 논리식을 통해 1-bit씩 값을 대입하였다.
① even_par[0]
9-bit codeword에서 p1(code 1번째)은 9, 7, 5, 3번째의 bit에 의해 결정되므로, 4개의 bit를 입력으로 하여 나타낸 논리식은 "A'B'C'D+A'B'CD'+A'BC'D'+A'BCD+ABC'D+ABCD'+AB'C'D'+AB'CD"이다.
또한 이 논리식은 XOR gate를 사용하여 "A⊕B⊕C⊕D"로 나타낼 수 있다.
(infor[4] = A, infor[3] = B, infor[1] = C, infor[0] = D)

② even_par[1]

9-bit codeword에서 p2(code 2번째)은 7, 6, 3번째의 bit에 의해 결정되므로, 3개의 bit를 입력으로 하여 나타낸 논리식은 “(A’B+AB’)C’+(AB+A’B’)C”이다.
(infor[3] = A, infor[2] = B, infor[0] = C)

③ even_par[2]
9-bit codeword에서 p3(code 4번째)은 7, 6, 5번째의 bit에 의해 결정되므로, 3개의 bit를 입력으로 하여 나타낸 논리식은 마찬가지로 “(A’B+AB’)C’+(AB+A’B’)C”이다.
(infor[3] = A, infor[2] = B, infor[1] = C)

④ even_par[3]
9-bit codeword에서 p4(code 8번째)은 오로지 9번째의 bit에 의해 결정되므로, infor[4]와 동일한 bit를 가진다.

```
syndrome[0] = even_par[0]^par[0]; // syndrome code(A'B+AB'=A xor B)
syndrome[1] = even_par[1]^par[1];
syndrome[2] = even_par[2]^par[2];
syndrome[3] = even_par[3]^par[3];
```

=> 5개의 information bit를 통해 수정된 parity bits를 error가 발생했던 parity bits와 비교하면서 syndrome을 구한다. 한 bit씩 비교하면서 값이 같을 경우는 [3:0] syndrome에 ‘0’을 대입하고 다를 경우는 ‘1’을 대입하는 원리인데, 이를 even_par와 par를 input으로 하는 XOR gate를 이용하여 구하였다. 주석에 쓰인 A와 B는 각각 even_par와 par을 뜻하며, Karnaugh map을 통해 나타난 논리식이 A’B+AB’인 것을 알 수 있다.

| B \ A | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

=> A’B+AB’ = A XOR B

```
// put the corrected information bits
if (syndrome == 4'd3) begin // syndrome == 3
infor = { cw[8], cw[6:4], (~cw[2]) }; end
else if (syndrome == 4'd5) begin // syndrome == 5
infor = { cw[8], cw[6], cw[5], (~cw[4]), cw[2] }; end
else if (syndrome == 4'd6) begin // syndrome == 6
infor = { cw[8], cw[6], (~cw[5]), cw[4], cw[2] }; end
else if (syndrome == 4'd7) begin // syndrome == 7
infor = { cw[8], (~cw[6]), cw[5], cw[4], cw[2] }; end
else if (syndrome == 4'd9) begin // syndrome == 9
infor = { (~cw[8]), cw[6:4], cw[2] }; end

end
assign hs = infor[4:0]; // assign the original hand sign
```

=> if문 경우의 수를 syndrome이 나타내는 bit 자리에 따라 나누어 해당 자리의 bit 값에 invert를 취해주는 코드이다. information bits인 3, 5, 6, 7, 9로 나누었으며 수정된 information bits를 [4:0] infor에 대입하고 [4:0] hs에 출력으로 내보냈다.

## 2) B-1 : b_1_converter

```
module b_1_converter(hs, n);
        input [4:0] hs;
        output [3:0] n;

        // fill in your implementation below
wire [3:0] temp;

// 2-level AND-OR circuit
assign temp[0] = ((~hs[4])&(~hs[3])&(~hs[2])&(~hs[1])&hs[0])|((~hs[4])&(~hs[3])&hs[2]&hs[1]&hs[0])
|(hs[4]&(~hs[3])&(~hs[2])&(~hs[1])&(~hs[0]))|(hs[4]&hs[3]&hs[2]&(~hs[1])&(~hs[0]))
|(hs[4]&hs[3]&hs[2]&hs[1]&hs[0]); // first bit
assign temp[1] = ((~hs[4])&(~hs[3])&hs[1]&hs[0])
|(hs[4]&hs[3]&hs[2]&(~hs[0]))|((~hs[4])&(~hs[3])&(~hs[2])&(~hs[1])&(~hs[0])); // second bit
assign temp[2] = ((~hs[4])&hs[3]&hs[2]&hs[1]&hs[0])|(hs[4]&hs[3]&hs[2]&hs[1])
|(hs[4]&hs[3]&hs[2]&(~hs[0])); // third bit
assign temp[3] = (hs[4]&(~hs[2])&(~hs[1])&(~hs[0]))
|((~hs[4])&(~hs[3])&(~hs[2])&(~hs[1])&(~hs[0])); // forth bit

assign n = temp[3:0]; // put the total output after combine the each bits

endmodule
```

(※ 코드가 긴 부분은 줄바꿈을 이용했습니다.)

• 원리

b_1_converter는 5-bit Hand Sign을 4-bit Unsigned Binary Number로 변환하는 기능을 한다. 5-bit에서 하나의 bit마다 입력으로 두어 총 5개를 입력으로 하는 2-level AND-OR 회로를 이용하였으며, 4-bit 중 하나의 bit를 출력으로 하는 AND-OR 회로를 사용하였으므로 총 4개의 2-level AND-OR 회로를 통해 converting을 수행한다.(Dataflow Description 방식)

• 순서

① 출력으로 나타낼 [3:0] n 대신, 임시로 저장할 [3:0] temp를 선언한다.
② 2-level AND-OR 회로를 통해 하나의 bit씩 값을 temp에 입력받아, 총 4개의 bit를 temp에 저장한다.
③ 입력받은 [3:0] temp의 값을 [3:0] n에 대입하여 출력으로 내보낸다.

• 구조 및 동작

```
        input [4:0] hs;
        output [3:0] n;

        // fill in your implementation below
wire [3:0] temp;
```

=>

| | |
|---|---|
| [4:0] hs | 입력 input |
| [3:0] n | 출력 output |
| [3:0] temp | 2-level AND-OR로 구한 값들을 저장할 공간 |

```
// 2-level AND-OR circuit
assign temp[0] = ((~hs[4])&(~hs[3])&(~hs[2])&(~hs[1])&hs[0])|((~hs[4])&(~hs[3])&hs[2]&hs[1]&hs[0])
|(hs[4]&(~hs[3])&(~hs[2])&(~hs[1])&(~hs[0]))|(hs[4]&hs[3]&hs[2]&(~hs[1])&(~hs[0]))
|(hs[4]&hs[3]&hs[2]&hs[1]&hs[0]); // first bit
assign temp[1] = ((~hs[4])&(~hs[3])&hs[1]&hs[0])
|(hs[4]&hs[3]&hs[2]&(~hs[0]))|((~hs[4])&(~hs[3])&(~hs[2])&(~hs[1])&(~hs[0])); // second bit
assign temp[2] = ((~hs[4])&hs[3]&hs[2]&hs[1]&hs[0])|(hs[4]&hs[3]&hs[2]&hs[1])
|(hs[4]&hs[3]&hs[2]&(~hs[0])); // third bit
assign temp[3] = (hs[4]&(~hs[2])&(~hs[1])&(~hs[0]))
|((~hs[4])&(~hs[3])&(~hs[2])&(~hs[1])&(~hs[0])); // forth bit
```

=> 2-level AND-OR circuit을 통해 4-bit 중 하나의 bit씩 [4:0] temp에 값을 입력받는 과정이다. 입력으로 받은 5-bit를 하나의 bit씩 나누어 각각 hs[4], hs[3], hs[2], hs[1], hs[0]으로 나타내었다. 총 5개의 bit를 입력으로 하고 한 개의 bit를 출력으로 하는 2-level AND-OR 회로를 구성하였다.

## • minimum 2-level AND-OR 논리식 과정

원리 : Karnaugh map으로 나타낸 sum of product는 minimal 2-level AND-OR 회로의 논리식을 나타낸다.

입력(input) : 입력받은 5-bit Hand Sign(“00000”부터 “11111”까지 총 10가지)에서 1-bit씩 나누어 hs[4] = A, hs[3] = B, hs[2] = C, hs[1] = D, hs[0] = E 으로 가정
출력(output) : 출력되어야 할 4자리의 Unsigned Binary Number(“0001”부터 “1010”까지 총 10가지) 중 하나의 bit(1자리).

※ 예외처리 : 10가지(“00000” ~ “11111”)의 input과 10가지(“0001” ~ “1010”)의 output에 대하여 karnaugh map을 그리고 논리식을 구하면, 10가지의 input을 제외한 나머지 경우의 input은 자동으로 “0000”으로 출력

① temp[0] (1번째 bit)

| ABC / DE | 000 | 001 | 011 | 010 | 110 | 100 | 101 | 111 |
|---|---|---|---|---|---|---|---|---|
| 00 | | | | | | 1 | | 1 |
| 01 | 1 | | | | | | | |
| 11 | | 1 | | | | | | 1 |
| 10 | | | | | | | | |

묶을 수 있는 항이 없기 때문에, [A’B’C’D’E+A’B’CDE+A’B’C’D’E’+ABCD’E’+ABCDE]이 최소화된 2-level AND-OR 논리식이다.

② temp[1] (2번째 bit)

| DE \ ABC | 000 | 001 | 011 | 010 | 110 | 111 | 101 | 100 |
|---|---|---|---|---|---|---|---|---|
| 00 | 1 | | | | | 1 | | |
| 01 | | | | | | | | |
| 11 | 1 | 1 | | | | | | |
| 10 | | | | | | 1 | | |

“00011”과 “00111”을 묶으면 A’B’DE이고, “11100”과 “11110”을 묶으면 ABCE’이다. 또한 “00000”은 묶을 수 있는 항이 없으므로 A’B’C’D’E’이다. 따라서 논리식은 [A’B’DE+ABCE’+A’B’C’D’E’]이라고 할 수 있다.

③ temp[2] (3번째 bit)

| DE \ ABC | 000 | 001 | 011 | 010 | 110 | 100 | 101 | 111 |
|---|---|---|---|---|---|---|---|---|
| 00 | | | | | | | | 1 |
| 01 | | | | | | | | |
| 11 | | | 1 | | | | | 1 |
| 10 | | | | | | | | 1 |

“11111”과 “11110”을 묶으면 ABCD이고, “11110”과 “11100”을 묶으면 ABCE’이 된다. 또한 “01111”은 묶을 수 있는 항이 없으므로 A’BCDE이다. 따라서 논리식은 [A’BCDE+ABCD+ABCE’]이라고 할 수 있다.

④ temp[3] (4번째 bit)

| DE \ ABC | 000 | 001 | 011 | 010 | 110 | 100 | 101 | 111 |
|---|---|---|---|---|---|---|---|---|
| 00 | 1 | | | | 1 | 1 | | |
| 01 | | | | | | | | |
| 11 | | | | | | | | |
| 10 | | | | | | | | |

“11000”과 “10000”을 묶으면 AC’D’E’이고, “00000”은 묶을 수 있는 항이 없으므로 A’B’C’D’E’이다. 따라서 논리식은 [AC’D’E’+A’B’C’D’E’]이다.

```
assign n = temp[3:0]; // put the total output after combine the each bits
```

=> 각 AND-OR 회로를 통해 입력받은 값으로 저장된 [3:0] temp를 [3:0] n에 대입하여 출력으로 내보낸다.

### 3) B-2 : b_2_converter

```
module b_2_converter(hs, n);
        input [4:0] hs;
        output [3:0] n;

        // fill in your implementation below
wire [3:0] n;
reg [3:0] state; // store the 4-bit unsigned binary number

always @ (hs) begin
case(hs) // convert hand sign to 4-bit unsigned binary number
5'b00001: state = 4'd1;
5'b00011: state = 4'd2;
5'b00111: state = 4'd3;
5'b01111: state = 4'd4;
5'b11111: state = 4'd5;
5'b11110: state = 4'd6;
5'b11100: state = 4'd7;
5'b11000: state = 4'd8;
5'b10000: state = 4'd9;
5'b00000: state = 4'd10;
default: state = 4'd0; // exception
endcase
end

assign n = state[3:0]; // output the 4-bit unsigned binary number

endmodule
```

• 원리

b_2_converter는 5-bit Hand Sign을 4-bit Unsigned Binary Number로 변환하는 b_1_converter와 같은 기능을 한다. case 구문을 통해 입력받은 5-bit에 따라서, 그에 알맞는 4-bit unsigned binary number으로 변환하고 출력으로 내보낸다.

• 순서

① 출력값인 [3:0] n을 wire로 선언하고, 4-bit unsigned binary number를 대입할 [3:0] state를 선언한다.

② case 구문을 통해 입력으로 받은 5-bit를 그에 알맞은 4-bit unsigned binary number로 변환하고 [3:0] state에 저장한다.

③ [3:0] state에 저장된 값을 [3:0] n에 대입하고 출력으로 내보낸다.

• 구조 및 동작

```
        input [4:0] hs;
        output [3:0] n;

        // fill in your implementation below
wire [3:0] n;
reg [3:0] state; // store the 4-bit unsigned binary number
```

=>

| | |
|---|---|
| [4:0] hs | 입력 input |
| [3:0] n | 출력 output |
| [3:0] state | 출력값인 4-bit unsigned binary number를 임시로 저장할 공간 |

```
always @ (hs) begin
case(hs) // convert hand sign to 4-bit unsigned binary number
5'b00001: state = 4'd1;
5'b00011: state = 4'd2;
5'b00111: state = 4'd3;
5'b01111: state = 4'd4;
5'b11111: state = 4'd5;
5'b11110: state = 4'd6;
5'b11100: state = 4'd7;
5'b11000: state = 4'd8;
5'b10000: state = 4'd9;
5'b00000: state = 4'd10;
default: state = 4'd0; // exception
endcase
end
```

=> 입력받은 5-bit의 hs를 기준으로 case 구문을 이용하였다. 경우에 따라 [3:0] state를 1~10까지의 4-bit unsigned binary number를 저장하도록 하였다. 예외 처리는 case 구문의 “default”인 경우를 이용하여, [3:0] state에 “0000”을 저장하는 것으로 하였다.

```
assign n = state[3:0]; // output the 4-bit unsigned binary number
```

=> case 구문을 통해 4-bit unsigned binary number가 저장된 [3:0] state를 [3:0] n에 대입하고 output으로 출력한다.

### 4) C : top

```
module top(cw, result);
        input [8:0] cw;
        output [3:0] result;

        // fill in your implementation below
wire [4:0] correct;
wire [3:0] result;

a_hamdec hamdec (.cw(cw), .hs(correct)); // module instantiation
b_1_converter converter1 (.hs(correct), .n(result));

endmodule
```

• 원리

top module은 앞서 구현한 (9,5) hamming decoder와 converter를 module instantiation 하여 회로를 구성하고, 본래 설계 목적인 Finger-counting Converter 기능을 수행한다.(Structural Description 방식 사용)

• 순서

① a_hamdec module의 출력과 b_1_converter(또는 b_2_converter) module을 연결할 wire [4:0] correct을 선언한다.
② b_1_converter(또는 b_2_converter) module의 output을 저장할 [3:0] result를 저장한다.
③ 선언된 module instantiation을 통해 finger-counting converter 기능을 실행시키도록 한다.

• 구조 및 동작

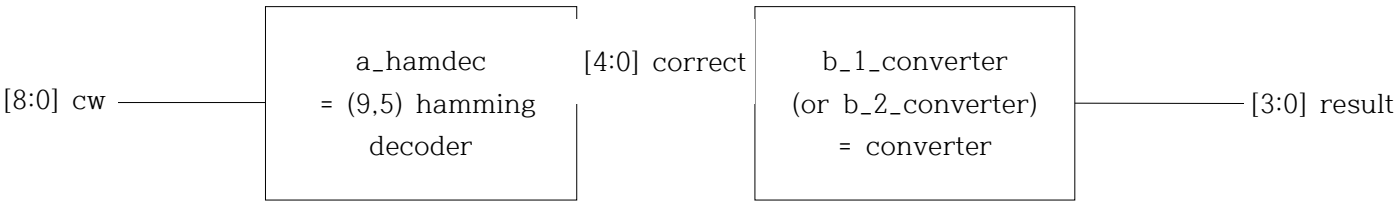
```
        input [8:0] cw;
        output [3:0] result;

        // fill in your implementation below
wire [4:0] correct;
wire [3:0] result;
```

=>

| | |
|---|---|
| [8:0] cw | 입력 input |
| [3:0] result | 출력 output |
| [4:0] correct | a_hamdec의 출력값과 converter의 입력값을 저장할 공간 |

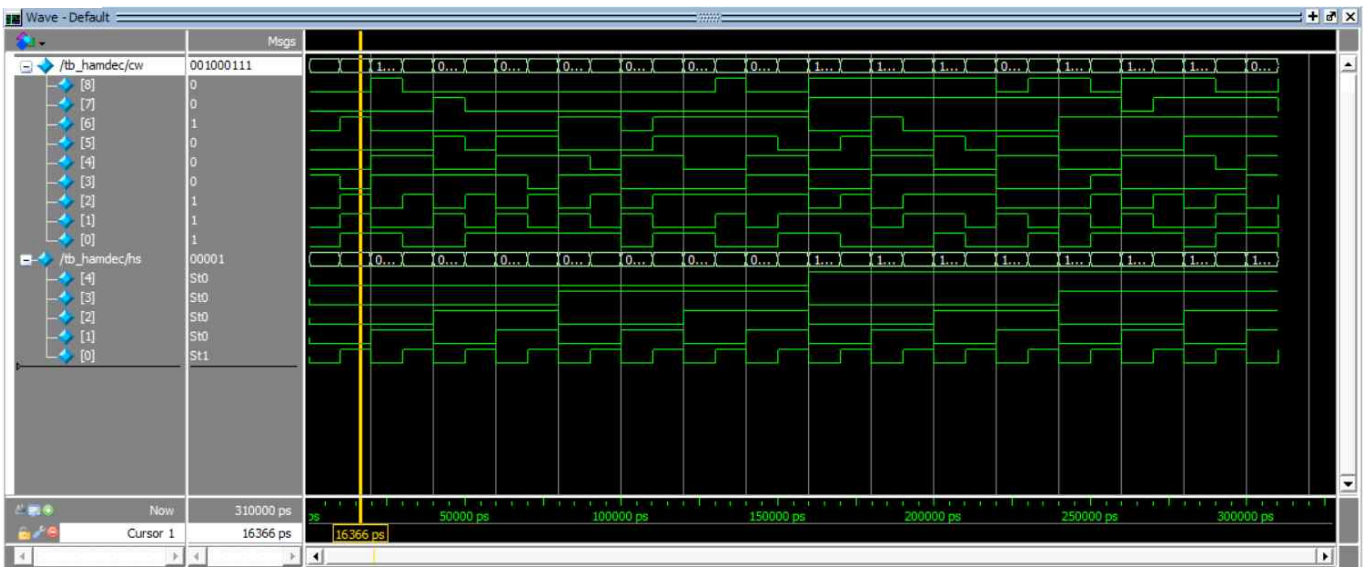
```
a_hamdec hamdec (.cw(cw), .hs(correct)); // module instantiation
b_1_converter converter1 (.hs(correct), .n(result));
```

=> 앞서 구현한 a_hamdec와 b_1_converter(또는 b_2_converter) module을 module instantiation을 통해 회로에 연결한다. a_hamdec은 9-bit codeword인 [8:0] cw을 입력으로 받아, 1-bit error가 수정된 5-bit information bits인 [4:0] correct를 출력하는 구조이다. 또한 converter는 a_hamdec이 출력한 [4:0] correct를 입력으로 받고 4-bit binary number가 담겨있는 [3:0] result를 출력으로 한다. 이를 block diagram으로 나타내면 아래와 같다.



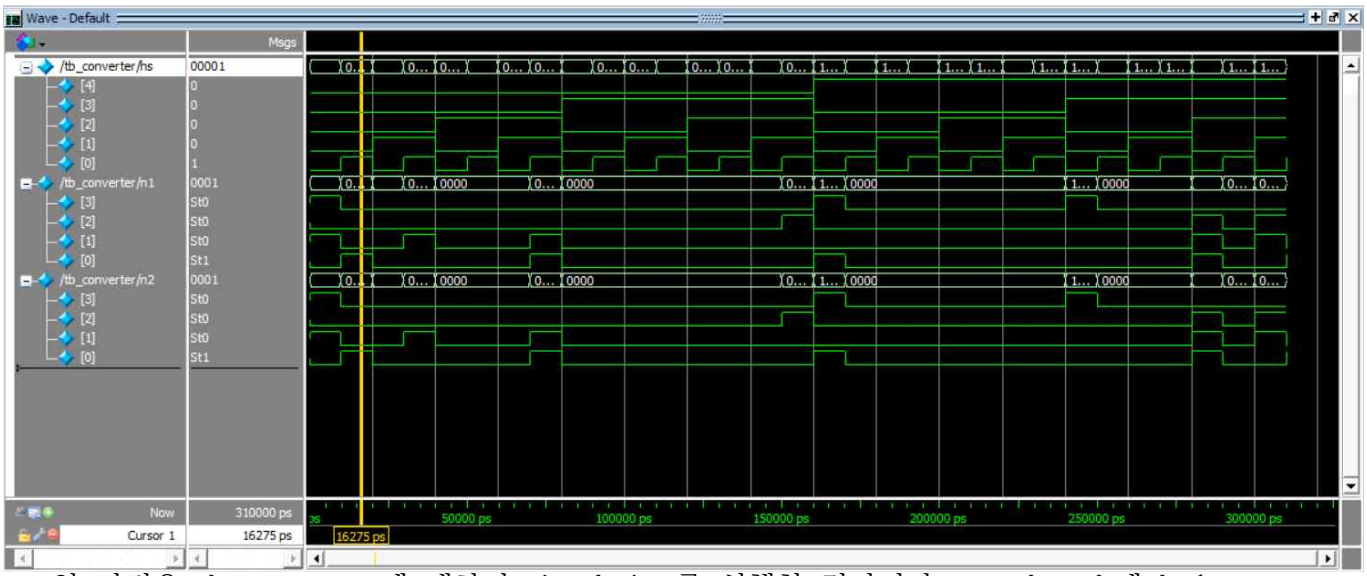

## [3] Sub-module의 simulation 분석

### 1) A : (9,5) Hamming Decoder



=> 위 사진은 tb_hamdec에 대하여 simulation을 실행한 결과이다. 왼쪽 표기판에 /tb_hamdec/cw는 input이 9-bit codeword “001000111”일 때를 나타낸 것이며, 이는 7번째 bit position에 error가 발생하였으므로 error correcting 된 codeword는 “000000111“이 되어야 한다. /tb_hamdec/hs에 ”00001“으로 output이 표기된 것으로 보아, 7번째 bit 값이 1->0으로 correcting 되었고 error 없는 information bits를 정상적으로 출력하였다고 할 수 있다.

## 2) B : b_1_converter & b_2_converter




=> 위 사진은 tb_converter에 대하여 simulation를 실행한 결과이다. test bench에 b_1_converter와 b_2_converter를 모두 module instantiation하여 정상적으로 작동하는지 확인하였다. 마찬가지로 /tb_converter/hs에 input으로 5-bit hand sign ”00001“을 넣고 converting 한 결과, b_1_converter와 b_2_converter 모두 4-bit unsigned binary number인 ”0001“이 출력된 것을 볼 수 있다. ”00001“은 엄지손가락 한 개만 펴진 경우이므로 첫 번째를 뜻하는 ”0001“이 출력되어야 하고 두 convert 모두 올바르게 출력되었다. 사진에 나타난 signal을 살펴보면 b_1_converter와 b_2_converter 모두 같은 모양이므로 모든 경우에서 같은 값으로 출력되었음을 알 수 있다. 따라서 converter는 정상적으로 작동하였다고 할 수 있다.

## [4] Top module 분석

### 1) Test Bench Code(tb_top.v)

```
`timescale 1ns/1ps
module tb_top();
        reg [8:0] cw;
        wire [3:0] result;

        // fill in your implementation below
top top_test (cw, result); // Module Instantiation

initial begin // Test Case
                cw = 9'b001000111;      // information = 00001 , result = 0001
                #10
                cw = 9'b000011100;      // information = 00011 , result = 0010
                #10
                cw = 9'b000110101;      // information = 00111 , result = 0011
                #10
                cw = 9'b001011111;      // information = 01111 , result = 0100
                #10
                cw = 9'b101111110;      // information = 11111 , result = 0101
                #10
                cw = 9'b011111001;      // information = 11110 , result = 0110
                #10
                cw = 9'b111110000;      // information = 11100 , result = 0111
                #10
                cw = 9'b111000010;      // information = 11000 , result = 1000
                #10
                cw = 9'b110000011;      // information = 10000 , result = 1001
                #10
                cw = 9'b000001000;      // information = 00000 , result = 1010
                #10
                cw = 9'b011001101;      // information = 11001 , result = 0000
                #10
                cw = 9'b101010011;      // information = 11010 , result = 0000
                #10
                cw = 9'b111010101;      // information = 11011 , result = 0000

end

endmodule
```

• 구조 및 동작

```
        reg [8:0] cw;
        wire [3:0] result;
```

=> tb_top() test bench는 input인 9-bit codeword를 저장할 reg [8:0] cw과 output으로 4-bit unsigned binary codeword를 출력할 wire [3:0] result를 선언하였다.

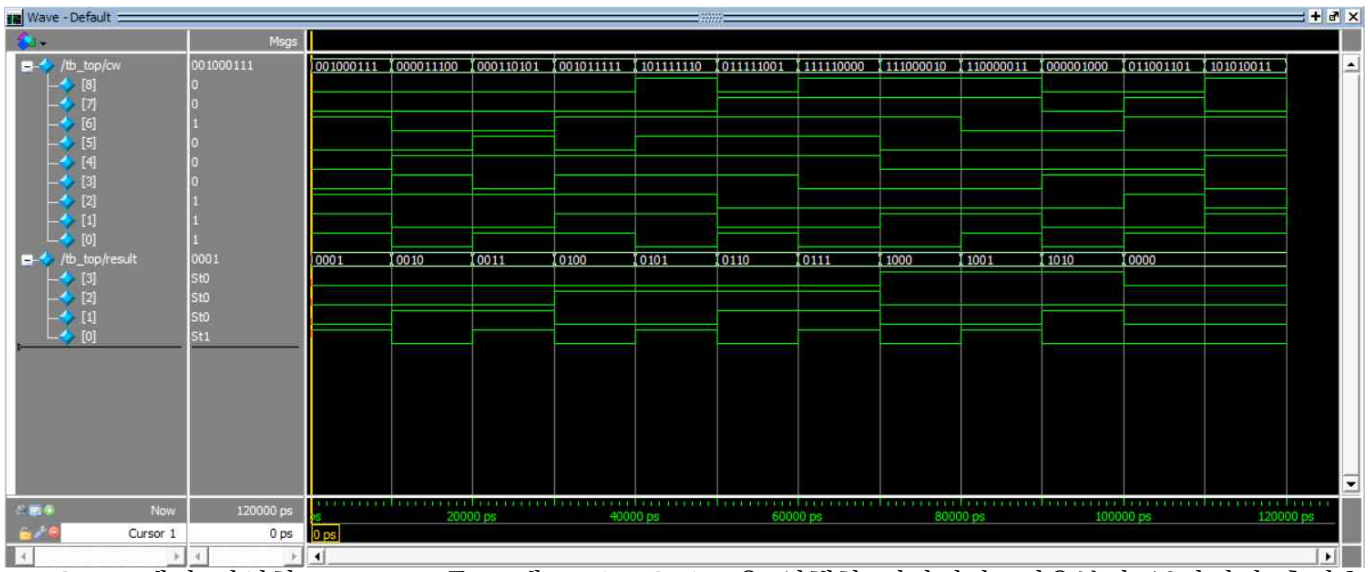
```
top top_test (cw, result); // Module Instantiation
```

=> top module을 top_test 이름으로 module instantiation 하였는데, 선언한 top module 안에는 finger-counting converter 기능을 수행할 a_hamdec과 b_1_converter(또는 b_2_converter) module이 모두 담겨있다. 따라서 입력으로 [8:0] cw를 넣으면, codeword에 해당하는 4-bit number가 [3:0] result에 출력될 것이다.

```
initial begin // Test Case
                cw = 9'b001000111;      // information = 00001 , result = 0001
                #10
                cw = 9'b000011100;      // information = 00011 , result = 0010
                #10
                cw = 9'b000110101;      // information = 00111 , result = 0011
                #10
                cw = 9'b001011111;      // information = 01111 , result = 0100
                #10
                cw = 9'b101111110;      // information = 11111 , result = 0101
                #10
                cw = 9'b011111001;      // information = 11110 , result = 0110
                #10
                cw = 9'b111110000;      // information = 11100 , result = 0111
                #10
                cw = 9'b111000010;      // information = 11000 , result = 1000
                #10
                cw = 9'b110000011;      // information = 10000 , result = 1001
                #10
                cw = 9'b000001000;      // information = 00000 , result = 1010
                #10
                cw = 9'b011001101;      // information = 11001 , result = 0000
                #10
                cw = 9'b101010011;      // information = 11010 , result = 0000
                #10
                cw = 9'b111010101;      // information = 11011 , result = 0000

end
```

=> initial 구문을 통해 test case를 만든 코드이다. input인 [8:0] cw에 1-bit error가 발생한 9-bit codeword를 넣는데, 주석에 명시된 것처럼 error가 수정된 information bits를 거쳐 정상적인 4-bit result가 출력되어야 한다. test case는 hand sign이 나타내는 신호인 1~10까지(”0001“~”1010“) 순서대로 출력하고 나머지 3가지는 예외 처리하여 ”0000“이 출력되도록 작성하였다.

## 2) Simulation 결과(tb_top.v)




=> tb_top에서 작성한 test case를 토대로 simulation을 실행한 결과이다. 처음부터 10가지의 출력은 ”0001“부터 ”1010“까지 이루어지며, 그 뒤 3가지의 출력은 예외 처리이므로 ”0000“으로 출력되는 것을 확인할 수 있다. 위와 같이 simulation이 올바르게 되는 것으로 보아, top test bench의 finger-counting converter는 정상적으로 작동한다는 것을 알 수 있다.

## [5] 기능별 달성도

### 1) A : (9,5) Hamming Decoder

=> Hamming Decoder의 기능은 1-bit error가 발생한 9-bit Hamming Codeword를 Error correcting과 Decoding을 통해 5-bit original hand sign을 출력하는 것이다. tb_hamdec을 통해 Hamming decoder의 simulation 결과를 보면, error가 발생한 bit position을 찾고(syndrome) error correcting 하여 5-bit information bits가 올바르게 출력되는 것을 알 수 있다. 따라서 Hamming Decoder는 기능 부분에서 문제가 없으므로 10점을 주고 싶다.

### 2) B : 5-bit Hand Sign to 4-bit Unsigned Binary Number Converter

=> Converter의 기능은 error correcting 된 5-bit hand sign을 2-level AND-OR 회로를 통해 sign에 맞는 4-bit number로 바꾸는 것이다. 마찬가지로 tb_converter를 통해 두 가지의 converter(b_1_converter, b_2_converter)를 simulation한 결과, 해당 hand sign에 맞는 4-bit number가 출력되었고, 두 converter 모두 같은 값을 출력하였다. 또한 1~10을 제외한 나머지 경우를 ”0000“으로 출력하는 예외 처리도 올바르게 된 것을 확인하였다. 따라서 해당 모듈은 converter 기능을 올바르게 수행하고 있다고 할 수 있다. b_1_converter은 dataflow description 방식으로 구현했지만, verilog를 다루는데 아직 미숙하여 생각보다 코드를 잘 구성하지 못한 것 같아 아쉬운 점은 남아있다. 그러므로 기능은 문제없지만 개인적으로 코드 구성에 의해 9.5점을 주고 싶다.

### 3) C : Top Module, D : Test Bench

=> Top 모듈은 앞서 구현한 (9,5) Hamming Decoder와 Converter를 수행하여 finger-counting converter 기능을 하는 module이다. Hamming Decoder와 Converter를 module instantiation 하여 structural description 방식으로 회로를 설계한다. 설계한 top module을 test bench로 확인한 결과, 1-bit error가 발생한 9-bit codeword를 입력으로 넣었을 때, 4-bit unsigned binary number가 올바르게 출력되는 것을 알 수 있다. 따라서 설계한 Top module과 test bench 모두 올바르게 코드를 작성하였고 기능도 정상적으로 동작하기 때문에 10점을 주고 싶다.