REPORT

보고서 작성 서약서

1. 나는 타학생의 보고서를 복사(Copy)하지 않았습니다.
2. 나는 타학생의 보고서를 인터넷에서 다운로드 하여 대체하지 않았습니다.
3. 나는 타인에게 보고서 제출 전에 보고서를 보여주지 않았습니다.
4. 보고서 제출 기한을 준수하였습니다.

나는 보고서 작성시 위법 행위를 하지 않고,
성.균.인으로서 나의 명예를 지킬 것을 약속합니다.

과 목 : 논리 회로 설계실험
과 제 명 : Design 1:4 DEMUX
담당교수 : 오 윤 호 교수님
학 과 : 전자 전기 공학부
학 년 : 3 학 년
학 번 : 2016311290
이 름 : 이 지 학
제 출 일 : 2021.03.26

1:4 DEMUX 설계

*이지학

*성균관대학교 전자전기공학부
dlwlgkr1@g.skku.edu

Design 1:4 DEMUX

*Ji-Hak Lee

*Dept. pf Electronic & Electrical Engineering, Sungkyunkwan University

목 차

*보고서 표지 포함 6 Page (표지 제외 5 Page)

1. 요 약 [Brief]

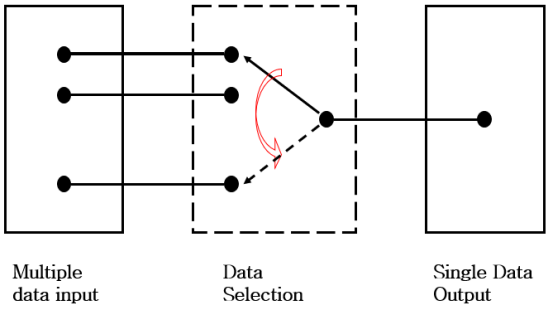
1:4 DEMUX 설계에 앞서 MUX로 통칭되는 Multiplexer와 DEMUX (Demultiplexer)의 차이점을 명확하게 이해하고 설계를 진행한다. Data Flow modeling 기법과 Gate-Level modeling 기법을 통해 설계를 진행하고, 각 Modeling 기법의 차이점과 설계 시 고려해야 할 사항들을 확인한다. Modeling을 하기에 앞서 해당 Module의 Truth Table과 Karnaugh Map을 이론적으로 점검하고 이를 Module 설계 시 어떻게 적용할지 고찰한다. Conditional operator, Conditional statement에 대한 이해를 바탕으로 Sample code(test bench 및 4:1 DEMUX Behavioral modeling)를 고찰하고 해당 Loop에 사용된 요소들의 쓰임에 대해 공부한다. 마지막으로 완성된 각각의 Module이 합리적으로 test 될 수 있는지 검증하고 이를 시뮬레이션 결과로 확인한다.

2. 이론적 접근

[Theoretical approach]

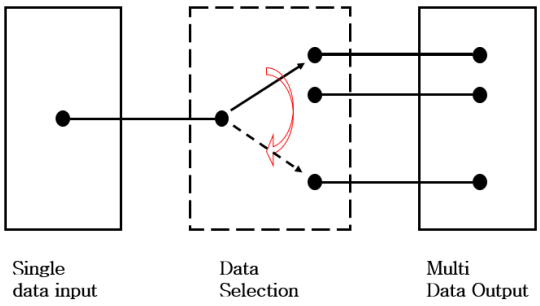
1) MUX: Multiplexer

=> 여러 개의 INPUT 중 하나를 선택하여 출력으로 내보내는 논리 회로이다.



2) DEMUX: Demultiplexer

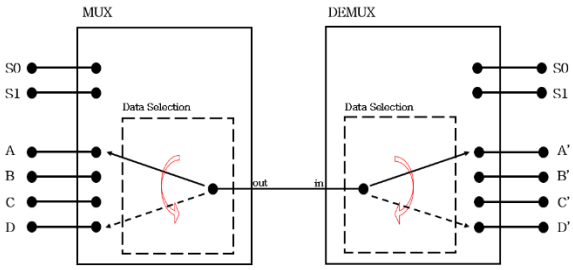
=> 하나의 INPUT을 받아 어느 OUTPUT으로 보낼지 선택하여 출력으로 내보내는 논리 회로이다.



3) MUX와 DEMUX

=> 통신 프로세스에 빗대어 MUX와 DEMUX를 설명해보면, MUX는 송신 단, DEMUX는 수신 단에 해당한다.

=> 다수의 사용자 간 통신에서 회선을 하나만 사용해야 하는 경우 MUX와 DEMUX를 함께 사용한다.

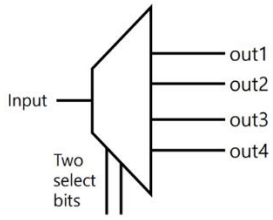


4) 1:4 DEMUX

1:4 DEMUX는 하나의 INPUT[a]을 받아, SELECT BITS[S0, S1]에 의해 선택된 OUTPUT Port[out1, out2, oout3, out4]로 전달하는 논리 회로이다.

* 설계 조건:

- Boolean expression must consist of AND, OR, NOT only.
- Other operators are not allowed and using parentheses is allowed.
- K maps must be correct



<TRUTH TABLE>

Select bits

S0	S1	OUT
0	0	out1
1	0	out2
0	1	out3
1	1	out4

(1) Verbal Description (Single input: a)

Out1 delivers input a if the select bits S0 is 0 and S1 is 0.

Out2 delivers input a if the select bits S0 is 1 and S1 is 0.

Out3 delivers input a if the select bits S0 is 0 and S1 is 1.

Out4 delivers input a if the select bits S0 is 1 and S1 is 1.

(2) Truth Table

TRUTH TABLE

S0, S1: two select bits / **a**: single input / **out1, out2, out3, out4**: multi outputs

S0	S1	out1
0	0	a
0	1	0
1	0	0
1	1	0

S0	S1	out2
0	0	0
0	1	0
1	0	a
1	1	0

S0	S1	out3
0	0	0
0	1	a
1	0	0
1	1	0

S0	S1	out4
0	0	0
0	1	0
1	0	0
1	1	a

(3) Karnaugh map

[out1]

S0S1 / a	00	01	11	10
0	a	0	0	0
1	a	0	0	0

[out2]

S0S1 / a	00	01	11	10
0	0	0	0	a
1	0	0	0	a

[out3]

S0S1 / a	00	01	11	10
0	0	a	0	0
1	0	a	0	0

[out4]

S0S1 / a	00	01	11	10
0	0	0	a	0
1	0	0	a	0

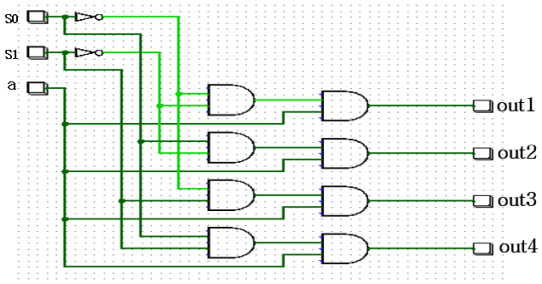
(4) Boolean Expression

out1 : $\overline{S0}\cdot\overline{S1}\cdot a$

out2 : $S0\cdot\overline{S1}\cdot a$

out3 : $\overline{S0}\cdot S1\cdot a$

out4 : $S0\cdot S1\cdot a$



3. Verilog 구현

[Verilog Implementations]

1) Gate Level Modeling

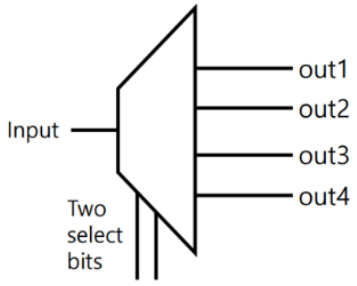
* 설계 조건:
Use AND, OR, NOT gate only. and use the source codes of AND, OR, NOT gates, which are provided by us.

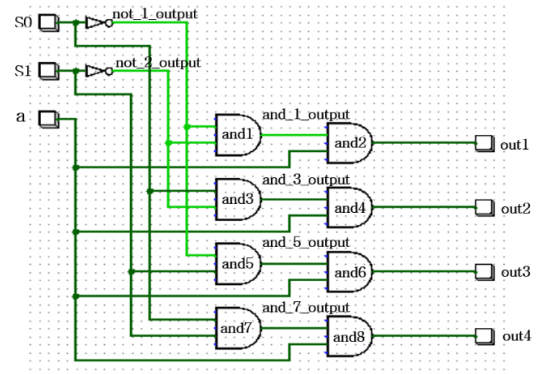
* 설계 환경:
Model Sim-Intel FPGA Starter Edition/Verilog HDL

* 실험 세부:

(1) 1:4 DEMUX의 Truth Table과 Karnaugh Map을 작성하고, 적합한 Boolean Expression을 도입해 본다.
(2) Boolean Expression을 기본 Gate로 표현해본다.
(3) Module화 된 각 Gate를 연결할 wire를 선언하고, Input과 Output을 고려하여 연결한다.

* Disassemble





* Verilog code 구현

```
//This is for the 1:4 demux module.

module one_to_four_demux_gatelevel_module (a, s0, s1, out1, out2, out3, out4);
        input a;
        input s0, s1;
        output out1, out2, out3, out4;

        wire not_1_output, not_2_output;
        wire and_1_output, and_3_output, and_5_output, and_7_output;   Gate와 Gate 사이를 연결할 Wire를 정의한다.
                                                                       * 사용하지 않는 Wire는 주석처리 하였습니다.
        //Fill this out.
        not_gate not_1(.a(s0), .out(not_1_output));
        not_gate not_2(.a(s1), .out(not_2_output));

        and_gate and_1(.a(not_1_output), .b(not_2_output), .out(and_1_output));
        and_gate and_2(.a(and_1_output), .b(a), .out(out1));

        and_gate and_3(.a(s0), .b(not_2_output), .out(and_3_output));
        and_gate and_4(.a(and_3_output), .b(a), .out(out2));

        and_gate and_5(.a(not_1_output), .b(s1), .out(and_5_output));
        and_gate and_6(.a(and_5_output), .b(a), .out(out3));

        and_gate and_7(.a(s0), .b(s1), .out(and_7_output));   각 Gate 사이의 input과 Output Wire를 연결한다.
        and_gate and_8(.a(and_7_output), .b(a), .out(out4));  * Sample Code의 구성을 최대한 따르는 방향으로 Modeling을 진행했습니다.

endmodule
```

- Gate와 Gate 사이를 연결할 Wire를 정의한다.

- 각 Gate 사이의 input과 output 관계를 유념하며 wire로 연결한다. (상기 Gate 개요 참고)

- 이전 실험에서 사용하였던 AND Gate의 경우 허용 INPUT 수가 2개 이므로 2Level AND 회로로 구현한다.

2) Data Flow Modeling

* 설계 조건:
A hybrid implementation is not allowed.

* 설계 환경:

Model Sim-Intel FPGA Starter Edition/Verilog HDL

* 실험의 검증: Behavioral Modeling 기법으로 구현된 1-Bit Full Adder (sample code)와 Simulation 동작 결과를 대조하여 본다.

* 실험 세부:

(1) 1:4 DEMUX의 Truth Table과 Karnaugh Map을 작성하고, 설계에 적합한 Boolean Expression을 도입한다.

(2) Boolean Expression을 참고하여 코드의 기본 Logic을 설계한다.

```
//This is for the 1:4 demux module.

module one_to_four_demux_dataflow_module (a, s0, s1, out1, out2, out3, out4);
        input a;
        input s0, s1;
        output out1, out2, out3, out4;

        //Fill this out
        assign out1 = (a && !s0 &&!s1);
        assign out2 = (a && s0 &&!s1);
        assign out3 = (a && !s0 &&s1);
        assign out4 = (a && s0 &&s1);

endmodule
```

<Boolean Expressions>

out1 : $\overline{S0}$·$\overline{S1}$·a
out2 : S0·$\overline{S1}$·a
out3 : $\overline{S0}$·S1·a
out4 : S0·S1·a

- out1에 (a && !s0 && !s1)의 연산 결과를 할당한다.

- out2에 (a && s0 && !s1)의 연산 결과를 할당한다.

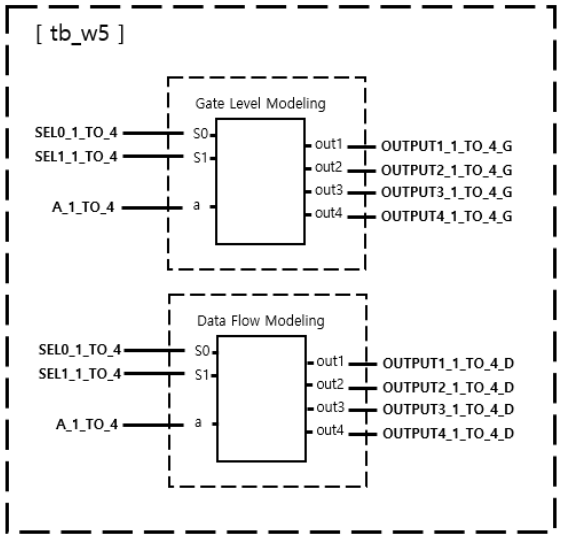
- out3에 (a && !s0 && s1)의 연산 결과를 할당한다.

- out4에 (a && s0 && s1)의 연산 결과를 할당한다.

3) Test Bench

* 검증 요소

(1) 입력 값이 Module에 정확히 할당되는가?
(2) Module 출력 값이 Truth Table과 일치하는가? (올바른 동작을 보이는가?)
(3) Test pattern이 output의 모든 경우를 발생시킬 수 있는가?




```
`timescale 1ns/1ns

module tb_w5;

        //2:1 MUX
        //input
        reg A_2_TO_1, B_2_TO_1;
        //select (input)
        reg SEL_2_TO_1;
        //output
        wire OUTPUT_2_TO_1_B; //behavioral modeling
        wire OUTPUT_2_TO_1_D; //dataflow modeling
        wire OUTPUT_2_TO_1_G; //gatelevel modeling

        ////////////////////////////////
        ////////////////////////////////
        ////////////////////////////////
        //1:2 DEMUX
        //input
        reg A_1_TO_2;

        //select (input)
        reg SEL_1_TO_2;
        wire OUTPUT1_1_TO_2_B, OUTPUT2_1_TO_2_B; //behavioral modeling
        wire OUTPUT1_1_TO_2_D, OUTPUT2_1_TO_2_D; //dataflow modeling
        wire OUTPUT1_1_TO_2_G, OUTPUT2_1_TO_2_G; //gatelevel modeling

        ////////////////////////////////
        ////////////////////////////////
        ////////////////////////////////
        //1:4 DEMUX
        //select (input)
        reg SEL1_1_TO_4, SEL0_1_TO_4;
        //input
        reg A_1_TO_4;
        //output
        wire OUTPUT1_1_TO_4_B, OUTPUT2_1_TO_4_B, OUTPUT3_1_TO_4_B, OUTPUT4_1_TO_4_B; //behavioral modeling
        wire OUTPUT1_1_TO_4_D, OUTPUT2_1_TO_4_D, OUTPUT3_1_TO_4_D, OUTPUT4_1_TO_4_D; //dataflow modeling
        wire OUTPUT1_1_TO_4_G, OUTPUT2_1_TO_4_G, OUTPUT3_1_TO_4_G, OUTPUT4_1_TO_4_G; //gatelevel modeling
        //temporal variable for loop
        integer count;
```

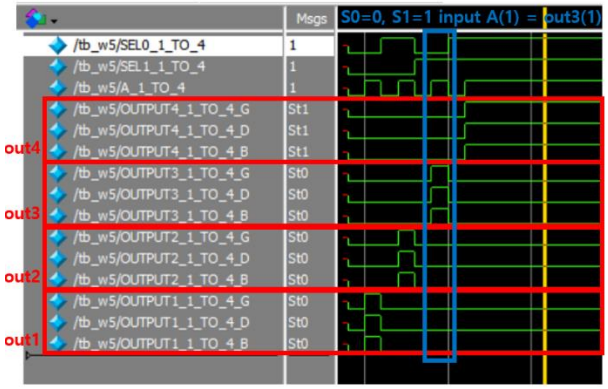
```
initial
begin
        A_2_TO_1 = 1'b0; B_2_TO_1 = 1'b1; SEL_2_TO_1 = 1'b0;
        A_1_TO_2 = 1'b0; SEL_1_TO_2 = 1'b0;
//      A_4_TO_1 = 1'b0; B_4_TO_1 = 1'b0; C_4_TO_1 = 1'b0; D_4_TO_1 = 1'b0;
end

initial
begin

        // Test pattern for 2:1 MUX
       A_2_TO_1 = 1'b0; B_2_TO_1 = 1'b0; SEL_2_TO_1 = 1'b0;
       #10 A_2_TO_1 = 1'b0; B_2_TO_1 = 1'b0; SEL_2_TO_1 = 1'b1;
       #10 A_2_TO_1 = 1'b0; B_2_TO_1 = 1'b1; SEL_2_TO_1 = 1'b0;
       #10 A_2_TO_1 = 1'b0; B_2_TO_1 = 1'b1; SEL_2_TO_1 = 1'b1;
       #10 A_2_TO_1 = 1'b1; B_2_TO_1 = 1'b0; SEL_2_TO_1 = 1'b0;
       #10 A_2_TO_1 = 1'b1; B_2_TO_1 = 1'b0; SEL_2_TO_1 = 1'b1;
       #10 A_2_TO_1 = 1'b1; B_2_TO_1 = 1'b1; SEL_2_TO_1 = 1'b0;
       #10 A_2_TO_1 = 1'b1; B_2_TO_1 = 1'b1; SEL_2_TO_1 = 1'b1;

       #90 //delay to border the test of 1:2 DEMUX
       #10 A_1_TO_2 = 1'b0; SEL_1_TO_2 = 1'b1;
       #10 A_1_TO_2 = 1'b1; SEL_1_TO_2 = 1'b0;
       #10 A_1_TO_2 = 1'b1; SEL_1_TO_2 = 1'b1;


       #90 //delay to border the test of 1:4 DEMUX
       for (count = 0; count <8; count = count +1)
              #10 {SEL1_1_TO_4, SEL0_1_TO_4, A_1_TO_4} = count;
end
```

- sample code에서 1:4 DEMUX의 test pattern은 for문을 사용해 구현했다 세 가지의 INPUT(S0, S1, a)이 000부터 시작하여 111까지 순차적으로 올라가고 이를 시뮬레이션 결과로 확인할 수 있다.

4. 실험 결과 (고찰)

[Experiment result]

(1) 1:4 DEMUX
[Data Flow Modeling/ Gate Level Modeling]



붉은 색으로 표시한 부분은 각각의 OUTPUT을 서로 다른 Modeling 기법으로 설계하여 시뮬레이션 한 결과이다. 주어진 sample code(Behavioral Modeling)와 더불어 총 3가지 방법으로 구현된 Module은 시뮬레이션 결과 동일한 파형으로 나타남을 알 수 있다. 또한 네 개의 out(out1, out2, out3, out4)에 표현될 수 있는 1(TRUE)과 0(FALSE)의 모든 경우를 표현할 수 있는 test pattern을 사용하였고 이를 시뮬레이션 결과로 확인할 수 있다.

파란색으로 표시한 부분은 주어진 각 Module이 설계 전 검증한 TRUTH TABLE과 Boolean expression 결과와 부합하는지 검증하기 위한 point이다. Select bit인 S0이 0이고, S1이 1인 경우 input에 할당된 값은 out3에 전달되어야 하는데 시뮬레이션 결과 합당한 결과가 도출되었음을 확인할 수 있다.

(2) 고찰

이번 실험에서는 Data Flow modeling 기법과 Gate-Level modeling 기법을 통해 설계를 진행하였고 해당 Module의 Truth Table과 Karnaugh Map을 이론적으로 점검하고 이를 Module 설계 시 어떻게 적용할지 고찰하였다. Conditional operator, Conditional statement에 대한 이해를 바탕으로 Sample code(test bench 및 4:1 DEMUX (Behavioral modeling)의 동작을 확인했고 각각의 Module이 합리적으로 test 될 수 있는지 검증하였다.

모든 Module의 구현 후에 각 Module을 정확하게 Test 할 수 있는 test bench code인지 고찰했고 발생할 수 있는 모든 경우의 수를 test하기 어려운 경우, 각 도출 값을 합리적인 형태로 일반화하여 정해진 Case 별로 Test 해야 함을 알 수 있었다. 또한 Test pattern 구현 시 검증해야 하는 test point를 명확하게 지정해야 하고 이는 모듈의 동작 검증에서 매우 중요함을 알 수 있었다.

5. 참 고

오윤호 (2021).*Logic Design Laboratory(ICE2005)* Week3.Sungkyunkwan Univ.