Final Lab

Number Comparison game

20220302 김지현

20220375 윤서진

20220215 최정윤

2023.06.11.

디지털 시스템 설계 (CSED273)

1. 개요

본 실습은 sequential Circuit를 바탕으로 암호를 맞추는 게임을 구현하는 것을 목표로 한다. 플레이어는 FPGA의 버튼 입력과 함께 스위치 8개(키패드 역할, 이하 키패드라고 지칭)를 통해 숫자를 입력하여 무작위로 정해진 암호에 관한 정보를 얻을 수 있다. 즉 플레이어는 특정한 버튼 입력을 통해 입력 숫자와 암호의 차이를 판단할 정보를 선택할 수 있다. 게임을 구현하는 과정을 통해 7 segment display, BCD, sequence detector등의 개념을 더 깊이 이해하고 실제적으로 어떻게 사용될 수 있는지 익히고 다룰 수 있을 것이다. 또한 프로젝트를 통해 수업 시간에 배운 다양한 개념들을 최대한 활용하면서 동시에 창의적으로 구현하는 경험을 쌓고자 한다.

2. 이론적 배경

1) BCD (Binary-Coded Decimal)

BCD (Binary-Coded Decimal)은 10진수를 2진수로 표현하는 방법 중 하나로 각 자릿수를 4비트로 각각 표현한다. 예를 들어 12를 BCD로 표현한다면 0001 0010이 된다. 2진수가 아닌 BCD 코드를 사용하는 가장 큰 이유는 사용자와의 상호작용이 편리하다는 것이다.

본 프로젝트에서는 해당 BCD code의 원리를 조금 변형하여 사용한다. 본 프로젝트에서는 두 자리 숫자 2개(무작위 암호, 사용자 입력 수)를 사용하는데, 이때 사용되는 두 자리 숫자의 각 자릿수는 1~8의 범위를 가지므로 각 자릿수는 3비트로 표현될 수 있어 두 자리 숫자는 총 6비트로 표현된다. 이 과정에서 BCD의 방식을 응용하여 예컨대 011000가 저장되어 있다면

16+8=24가 아니라 011은 4, 000은 1이고, Little Endian 방식에 따라 14로 간주하여 사용된다. 이러한 방식으로 7 segment 입출력 등에 사용된다.

2) Sequential Circuit

Sequential Circuit는 출력이 현재의 입력과 이전의 논리회로 상태의 조합에 의해 결정되는 논리회로이다. 순차 회로는 시간적인 의존성을 가지므로, 시퀀스(Sequence)라고도 불리며, 이를 통해 복잡한 계산이나 제어 동작을 수행할 수 있다. 순차 회로는 주로 상태 기계(State Machine)를 구현하는 데 사용되며, 시퀀셜 논리 회로, 레지스터(Register), 카운터(Counter), 타이밍 회로(Timing Circuit) 등 다양한 응용 분야에서 사용하게 된다. 순차 회로의 설계와 분석은 상태 다이어그램(State Diagram), 상태표(State Table), 상태 전이 표(State Transition Table) 등의 도구를 사용하여 수행하게 된다. 이를 통해 원하는 동작을 기술하고 회로를 구현할 수 있다. 이번 프로젝트에서는 사용자가 2비트 정보를 확인하고자 할 때, 00, 01, 10, 11쌍을 찾을 때 이 논리회로를 사용할 것이다.

또한 두 가지 타입의 clocked sequential network (Mealy machine과 Moore machine)이 있는데 이번 프로젝트에서는 Moore machine의 형태로 구현할 것이다. Moore machine은 출력값이 현재 상태에만 의존하고, 현재의 입력에는 출력이 영향을 받지 않는 것을 말한다.

3) 7 segment display

7 segment display는 숫자와 문자를 표현하기 위해서 사용되는 디지털 디스플레이다. 이 프로젝트에서는 사용자와 '맞춰야 할 수'를 비교해 up down을 표현할 때와 현재 상태(1비트나 2비트 정보, 혹은 대소관계 정보를 확인하는 상태)를 나타내는 데에 사용된다.

4) sequence detector

시퀀스 디텍터는 순차 논리 회로의 한 유형으로, 특정한 입력 패턴을 감지해 결과값을 나타내는 역할을 한다. 이전에 정의된 패턴과 일치하였을 때 출력을 생성하거나 특정 동작을 수행할 수 있다. 이번 프로젝트에서는 지정된 암호에서 감지된 패턴의 수와 추측한 암호에서의 수를 비교해 출력할 것이다.

3. 실험 준비

ㄱ. State transition diagram

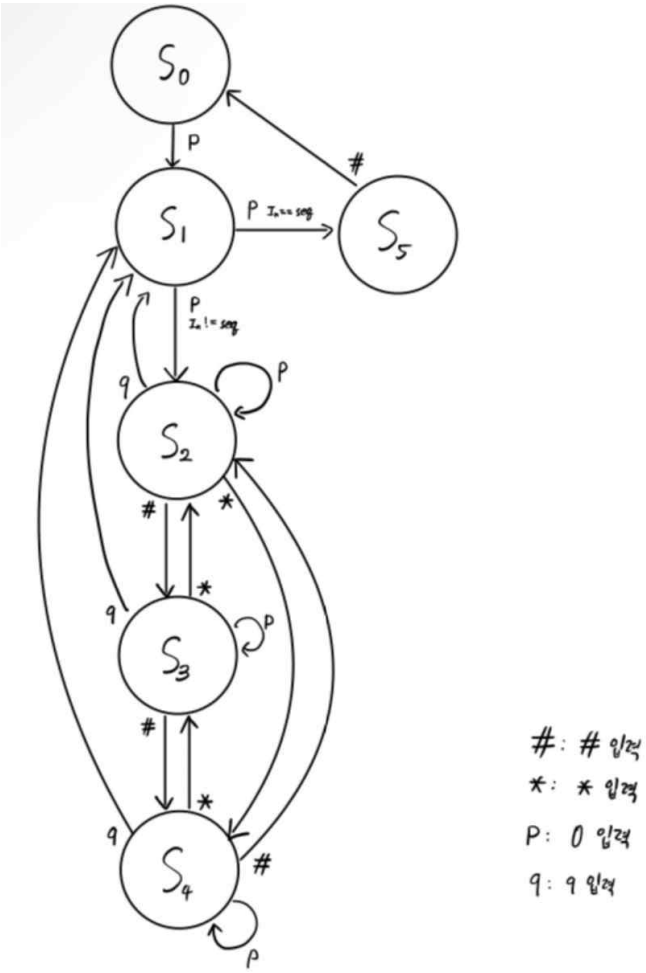


Figure State Diagram

위 diagram에서는 다음의 6가지의 상태를 가진다.

1. S0 : 맞춰야 할 수를 랜덤으로 생성하는 상태
2. S1 : 사용자의 입력을 받는 상태
3. S2 : 1비트 정보를 제공하는 상태
4. S3 : 2비트 정보를 제공하는 상태
5. S4 : 대소비교 정보를 제공하는 상태
6. S5 : 종료 상태

이러한 state 간의 변화는 모두 키패드 입력을 통해 이루어진다.

ㄴ. 전체적인 동작 설명

본 프로그램은 사용자가 키패드를 활용하여 랜덤하게 정해진 암호에 관한 정보를 얻어 그것을 맞추는 것을 목적으로 진행된다. 가장 먼저 랜덤하게

정해진 암호(targetNumber)가 결정되면, 사용자는 가운데 버튼(FPGA)을 눌러 그것의 정보를 얻어내는 과정을 시작할 수 있다. 사용자는 2자리 숫자(각 자리수의 숫자는 1~8의 범위, 이하 userInput)를 입력하고, 가운데 버튼(확인)을 누르는 과정을 통해 자신이 추측하는 암호를 입력한다. 만일 userInput이 암호(targetNumber)와 동일하면 종료 state(S5)로 이동한다.

사용자가 자신이 추측하는 암호를 입력하면, 1비트의 정보를 제공하는 state(S2)로 이동한다. 사용자가 왼쪽 버튼(FPGA)을 누르고 다시 가운데 버튼을 누르면 대소관계 정보를 제공하는 state(S4)로, 오른쪽 버튼(FPGA)을 누르고 가운데 버튼(FPGA)을 누르면 2비트의 정보를 제공하는 state(S3)로 이동한다. 사용자가 가운데 버튼(FPGA)을 누른 경우, 1비트의 정보를 제공받는 과정을 수행한다. 즉 사용자는 선택을 통해 0의 개수 혹은 1의 개수에 관한 userInput과 암호(targetNumber) 사이의 대소관계를 알 수 있다. 해당 정보가 7 segment를 통해 출력된 이후, 가운데 버튼을 다시 누르면 다시 1비트의 정보를 제공하는 state(S2)로 돌아가게 된다.

2비트의 정보를 제공하는 state(S3)로 이동한 경우에도, 왼쪽 버튼을 누르면 1비트의 정보를 제공하는 state(S2)로 이동, 오른쪽 버튼을 누르면 대소관계 정보를 제공하는 state(S4)로 이동하는 기능이 활성화된다. 그러한 입력 없이 가운데 버튼을 누르면 2비트의 정보를 제공받는 과정을 수행한다. 즉 사용자는 선택을 통해 userInput의 00, 01, 10, 11 개수에 관한 자신이 입력한 값과 지정된 암호 사이의 대소관계를 7 segment를 통해 제공 받는다. 제공받은 후 가운데 버튼을 다시 누르면, 다시 2비트의 정보를 제공하는 state(S3)로 돌아가게 된다.

대소관계 정보를 제공하는 state(S4)로 이동한 경우에도, 오른쪽 버튼을 누르면 2비트의 정보를 제공하는 state(S3)로 이동, 왼쪽 버튼을 누르면 1비트의 정보를 제공하는 state(S2)로 이동이 활성화된다. 만약 userInput과 암호의 대소관계를 알고 싶다면 다른 입력 없이 가운데 버튼을 누르면 된다. 해당 정보를 제공받은 후 다시 가운데 버튼을 입력하면, 다시 대소관계 정보를 제공하는 state(S4)로 이동한다.

그리고 위의 state들(S2, S3, S4)에서 아래쪽 버튼을 누른 후 가운데 버튼을 누른 경우, 다시 userInput를 입력하는 state(S1)로 이동한다.

종료 state(S5)로 이동한 경우, FPGA의 LED를 통해 정답임을 표시하고, 7 segment를 통해서도 End를 표시한다. 이후 가운데 버튼을 누르면 state(S0)로 돌아가 clock에 기반한 무작위 암호를 다시 설정하고 위 동작을 반복한다.

ㄷ. 입출력 방법 및 동작 설명

먼저 게임이 시작되면 처음의 가운데 버튼 입력 후, 사용자가 추측한 암호(userInput)를 키패드로 입력 받는다. 여기서의 사용자가 추측하는 암호는 1~8까지의 범위에서 2번 입력 받아와서 두 자릿수를 저장한다.

이는 각 한 자리의 숫자를 3비트로 표현하기 위함인데, 이러한 이유로 1을 000, 8을 111로 표현하는 방식으로 진행된다. (즉 0~7을 1씩 더해 1~8로 처리; 이는 구현의 편의성과 게임의 난이도를 높이기 위함) FPGA의 왼쪽 버튼, 가운데 버튼, 오른쪽 버튼, 아래쪽 버튼 입력은 state 이동에만 사용된다. 보다 자세하게 말하자면, 왼쪽 버튼은 이전 state로 이동, 가운데 버튼은 다른 입력에 따른 결정을 확정하는 역할로 주로 사용되고, 오른쪽 버튼은 다음 state(혹은 다음 게임)로 이동, 아래쪽 버튼은 재입력을 의미하여 S1 state로 이동하는 기능을 수행하도록 사용했다. 사용자 입력으로 userInput를 결정한 후, 가운데 버튼을 누르면 S1에서 userInput을 저장하고, 1비트 정보를 제공하는 S2로 이동한다.

1비트의 정보를 제공하는 상태인 S2에서

(1) 1을 입력 받은 경우(스위치 1을 누른 상태로 가운데 버튼 입력), 추측된 암호(userInput)에서의 0의 개수와 지정된 암호(targetNumber)에서의 0의 개수를 대소 비교한 결과를 7 segment에 출력한다. 같을 경우 SE, userInput의 0이 더 적을 경우 UP, userInput의 0이 더 많을 경우 dn(=Down)을 출력한다.
(2) 2를 입력 받은 경우, 위와 비슷하게 userInput 와 설정된 암호의 1의 개수를 비교한 결과를 출력한다.

위의 정보가 출력된 상태에서 가운데 버튼을 누르면 다시 1비트의 정보를 제공하는 S2로 돌아가고, 이 state에서는 7 segment에 1을 표시한다.

2비트의 정보를 제공하는 상태인 S3에서

(1) 1을 입력 받은 경우, 추측된 암호(userInput)에서의 00의 개수와 지정된 암호에서의 00의 개수를 비교한 결과를 7 segment에 출력한다. 같을 경우

SE, userInput의 00이 더 적을 경우 UP, userInput의 00이 더 많을 경우 dn을 출력한다.

(2) 2를 입력 받은 경우, 위와 비슷하게 userInput와 설정된 암호의 01의 개수를 비교한 결과를 출력한다. 단 이 경우 앞에서부터 '01'의 개수를 비교하는데, 예컨대 targetNumber이 78(111110)이고 userInput이 14(011000)인 경우 userInput의 01이 1개, targetNumber의 01이 0개이므로 dn을 출력한다. (현재 입력한 userInput보다 targetNumber는 01의 개수가 down임을 의미)

(3) 3을 입력 받은 경우, 위와 비슷하게 userInput 와 설정된 암호의 10의 개수를 비교한 결과를 출력한다. (2)와 동일하게 앞에서부터 개수를 읽음에 주의한다.

(4) 4를 입력 받은 경우, 위와 비슷하게 userInput 와 설정된 암호의 11의 개수를 비교한 결과를 출력한다.

위의 정보가 출력된 상태에서 가운데 버튼을 누르면 다시 2비트의 정보를 제공하는 S3로 돌아가고, 이 state에서는 7 segment에 2를 표시한다.

userInput과 지정된 암호 자체의 대소 관계 정보를 제공하는 S4에서는 7 segment로 4를 표시하여 현재 설정된 state가 무엇인지 보여준다. 이때 가운데 버튼을 누른 경우, 추측한 암호(userInput)와 지정된 암호(targetNumber)의 대소관계 정보를 제공한다.

(1) 만일 추측한 암호(userInput)가 지정된 암호보다 작다면, dn을 7 segment에 출력한다.

(2) 만일 추측한 암호(userInput)가 지정된 암호보다 크다면, UP을 7 segment에 출력한다. 예컨대 targetNumber = 78, userInput = 14이면 UP이다. 현재 입력한 userInput보다 targetNumber는 큰 수라는 의미이다.

위의 정보가 출력된 상태에서 가운데 버튼을 누르면 다시 대소관계 정보를 제공하는 S4로 돌아가고, 이 state에서는 7 segment에 4를 표시한다.

S1에서 입력된 userInput이 설정된 암호(targetNumber)와 동일할 경우 이동하는 S5(종료 상태)에서는 7 segment로 End를 표시하여 종료 상태에 있음을 보여준다. 다른 입력에는 반응하지 않고, 가운데 버튼을 누른 경우 다음 게임을 진행하기 위해 S0 state로 이동하여 무작위로 targetNumber를 재설정하며 작동을 반복한다.

ㄹ. 코드 작성 과정 및 고려 사항

먼저 이전의 lab 파일을 바탕으로 FPGA의 출력 설정 등의 요소들을 다루려 노력했고, if else 문이나 for문의 사용 제한 등도 없었기에 이를 바탕으로 익숙한 방식의 코딩을 먼저 시도했다.

1) 입력 관련

부품이 도착하기 전이므로 FPGA의 버튼과 스위치 입력을 전제하여 코드를 먼저 작성했다. FPGA의 가운데 버튼을 Action으로 두고, 잘못된 입력 등에 관해 잘못된 처리가 되지 않도록 일종의 clock처럼 쓰기 위해 Action에 관한 always를 통해서 대부분의 기능을 수행하도록 구현했다. 왼쪽 버튼의 입력을 받는 변수 p_Sel1이나 오른쪽 버튼의 입력을 받는 p_Sel2, 아래쪽 버튼의 입력을 받는 p_reInput은 매 입력마다 기존의 관련 변수 값(0 or 1)을 toggle시키는 방식으로 구현했다. 이는 각각의 입력에 따라 별도의 always문을 통해 state 변화를 진행하는 방식으로 발생하는 multi net 관련 에러를 피하고, 다른 입력 버튼을 추가하는 확장에 있어서도 일반적으로 처리할 수 있도록 하기 위함이다. 각 관련 변수 값(Sel1, Sel2, reinput)은 FPGA의 led를 통해 0인지 1인지를 보여주어 사용자가 해당 버튼이 현재 어떤 상태인지 확인할 수 있게 했다.

FPGA의 switch를 이용한 코드를 작성하는 과정에서도 constraints file 관련 등으로 문제가 많았지만, 그 외에도 고려해야 할 사항이 있었다. FPGA의 switch는 해당 입력을 hold할 수 있지만, 우리가 사용할 switch의 경우 값을 hold하지 않는 것임을 확인했다. 따라서 각각의 switch의 값에 대해서도 위와 동일하게 [7:0] p_sw를 입력 받아 [7:0] sw를 toggle시키는 방식으로 진행하였으나, 그 과정에서 반복되는 always 문이 코드의 가독성을 상당히 안 좋게 하였다. 관련하여 검색하여 찾아보던 중, always @(*)를 통해 간단히 sw<=p_sw;과 led[7:0] <= sw;로 해결될 수 있음을 이해할 수 있었다.

그러나 실제 부품을 사용했을 때 스위치가 눌렀을 때 led가 꺼지는 방식으로 작동하는 것으로 확인되어 sw<=!p_sw;로 그 값을 반전시켜 사용하려 했다. 그렇지만 이와 같은 변경 이후 led가 모두 켜진 상태에서 스위치를 누르면 더 밝게 해당 led가 빛나는 것으로만 표현되어 이에 관한 설명은 작동 전반에서 생략되게 되었다.

2) 변수 관련

본 코드에서 크게 다뤄지는 변수는 targetNumber과 userInput의 두 가지다. constraints file에서 input으로 받아온 clock의 posedge마다 6비트의

random의 값을 1씩 올리다 초기화하는 것을 반복하면서 targetNumber을 결정하는 단계에서 random의 값을 받아와 저장한다. 이 때문에 state 0에서는 처음에 COnE를 출력하는 과정이 존재한다. (COnE는 COnF(configuration)을 적으려 했으나 7 segment 출력에 사용할 요소가 너무 많아 F를 E로 대신 표기한 것이다.) userInput은 state 1에서의 사용자 입력을 통해 결정된다. 두 변수 모두 결정된 이후에는 먼저 두 변수 각각의 count0, count1, count00, count01, count10, count11의 값을 for문을 통해 결정한다. targetNumber의 경우에는 seq_count##, userInput의 경우에는 count##이다. 해당 for문은 간단한 sequence detector로 대신할 수 있을 것이라 생각했기에 사용했고, 게임의 난이도를 높이고자 앞에서부터 count를 하도록 했다. 예컨대 78는 110(7), 111(8)인데 111110으로 저장되고, 이 상태 그대로 count를 해서 count10 = 1, count01 = 0으로 처리했다.

3) state 관련

A. state 0

본래 기획과 달리 test code로 무작위로 정해진 targetNumber를 7 segment에 출력하도록 했으나, 해당 부분을 지우지 않았고 해당 출력을 제대로 보여주기 위해 counter를 사용했다. counter이 0일 때 COnE를 출력, 1일 때 targetNumber를 출력과 count 계산, 2일 때 state 1의 출력을 보여준다. (Action이 clock처럼 작동하기 때문에 1에서의 첫 입력은 state 1에서의 출력 이전에 진행되어야 하는 방식)

B. state 1

각 switch의 단일 입력에 따른 값으로 설정해야 하기에 초기에는 switch case 구문을 사용했으나, 다중 입력을 배제하고 |을 사용하여 or로 각 비트를 결정하여 userInput을 입력받도록 구현했다. 이 역시 두 자릿수이므로 counter가 0일 때 첫 자리 입력, 1일 때 두 번째 자리 입력과 count 계산, 2일 때 targetNumber와 같은지 비교하여 state 이동을 확인하도록 했다.

만일 재입력 기능이 없다면, state0에서 마지막에 설정해준 state1의 출력이 남아있다고 가정할 수 있지만, 재입력을 할 수 있기 때문에 state1인 경우 처음에 7 segment를 설정하는 부분이 필수적이었다.

C. state 2

먼저 state2에 맞는 display를 출력한 후, Sel1, Sel2, reInput의 입력이 있다면 이동을 진행하도록 한다. 여기서도 if를 간단히 사용했지만, 간단한 and를 통해 if문 없이 구현할 수 있을 것이다.

만일 Sel1, Sel2, reinput의 state 이동에 관한 변수가 모두 0이라면, sw에 따른 switch case 구문을 통해 count를 비교한 결과를 출력해준다.

처음 기획에는 예를 들어 targetNumber보다 userInput의 count#보다 작으면 dn(down)을 출력하도록 했으나, 사용자 입장에서 알고 있는 것은 userInput이므로 그것에 근거한 비교가 적절할 것 같아 "userInput보다 targetNumber의 count#은 ~하다"로 출력을 반대로 바꿨다.

D. state 3

먼저 state3을 의미하는 출력 이후, state2에서처럼 Sel1, Sel2, reinput의 변수가 0이라면 switch 입력에 따라 count 비교 출력을 제공한다.

단, state2에서처럼 count0의 비교만으로 count1이 확정되는 것이 아니기 때문에 각 count## 비교 정보마다 제대로 된 스위치 입력을 통한 출력인지 확인할 필요가 있다. 그래서 스위치 입력 없이 가운데 버튼을 누른 경우 state3을 의미하는 3은 남아있지만, 기존의 up, dn, SE의 출력은 지워지도록 할 필요가 있었다. 이를 바탕으로 각 count## 정보를 확인하고, 다른 count## 정보를 확인하기 전에는 한번 출력을 지우는 것이 확인하는 데에 큰 도움이 되었다.

E. state 4

state2, state3과 마찬가지로 출력과 state 이동에 관한 변수가 0인지 확인된 후에 targetNumber와 userInput의 대소 비교 정보를 제공한다.

단, targetNumber과 userInput이 같을 수는 없기 때문에 SE에 관한 출력은 고려하지 않는다. 만일 두 변수가 같다면 state1의 입력에서 state5로 이동하기 때문이다.

F. state 5

always Action 내부에서 state 5의 작동은 state=0; 뿐이다. state1에서 targetNumber와 userInput이 동일할 경우 state는 5로 변경되고, End를 이미 출력하기 때문에 그 상태에서 다시 가운데 버튼을 누른다면 state 0으로 초기화하여 또 다시 가운데 버튼을 누르면 COnE를 다시 출력하는 단계로 돌아간다.

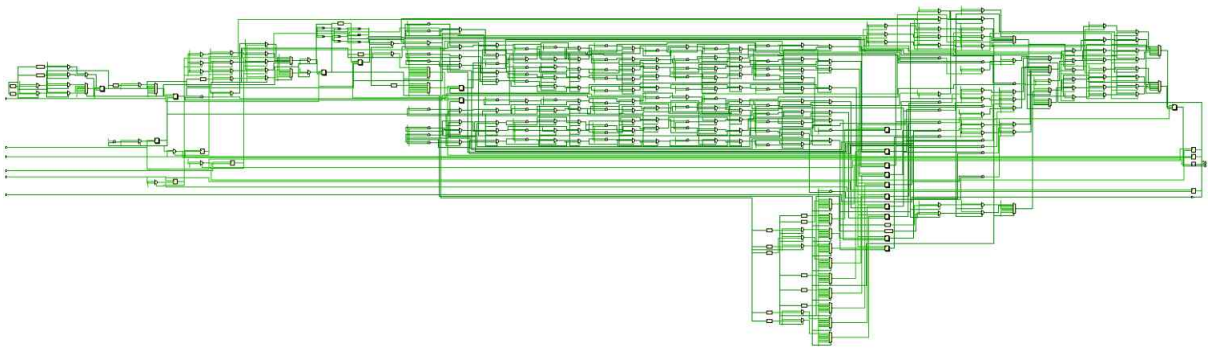
이처럼 Action을 clock처럼 state 이동을 처리하는 과정에서 state의 값을 바꿈과 동시에 해당 state의 출력을 수행하는 등의 고려가 포함되어야 했다.

그 외의 led_renderer module과 bcd_to_7seg module은 조교님의 lab 파일의 것을 변형해 사용하면서 최대한 에러가 발생하지 않도록 했다.

constraints file의 경우에도 조교님의 lab 파일을 참고했으나 "set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets Action_IBUF]"과 같은 코드를 추가하면서 관련 에러를 다뤘다.

4. 결과

먼저 작성한 코드의 schematic은 다음과 같다.



각 state에 대해서 별도의 module 설정을 하지 않았기 때문에 block으로 표현되지 않았고, if else, for, switch case 문법을 사용했기 때문에 더욱 복잡화되었다.

그리고 FPGA와 스위치를 빵판을 통해 연결하여 다음과 같이 정상적으로 작동함을 확인할 수 있었다.

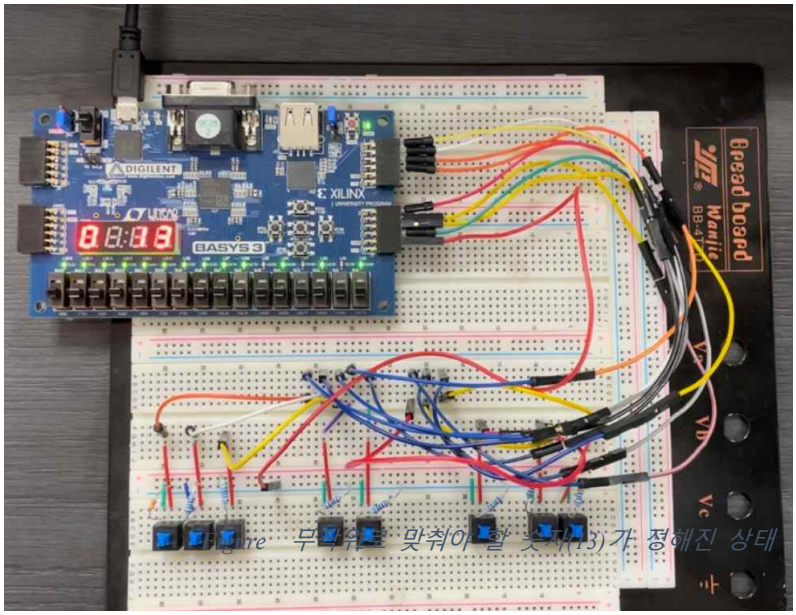


Figure 무늬원로 맞춰야할 것자(13)가 정해진 상태

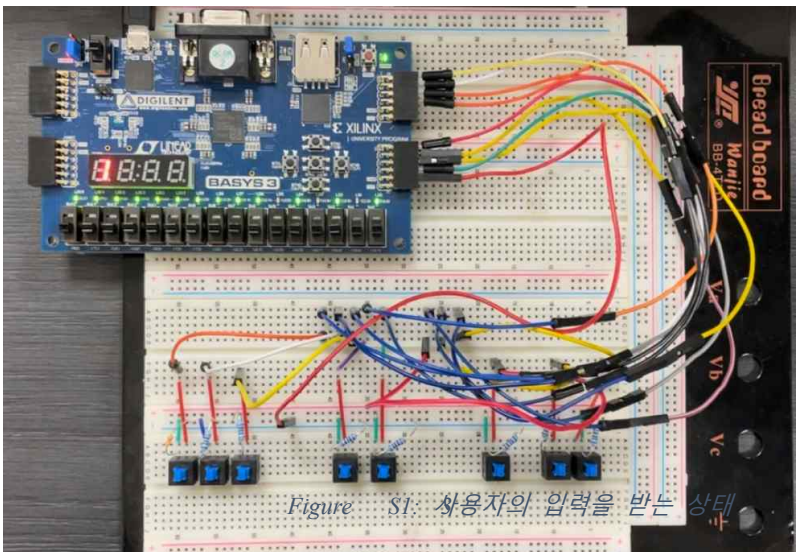


Figure S1. 사용자의 입력을 받는 상태

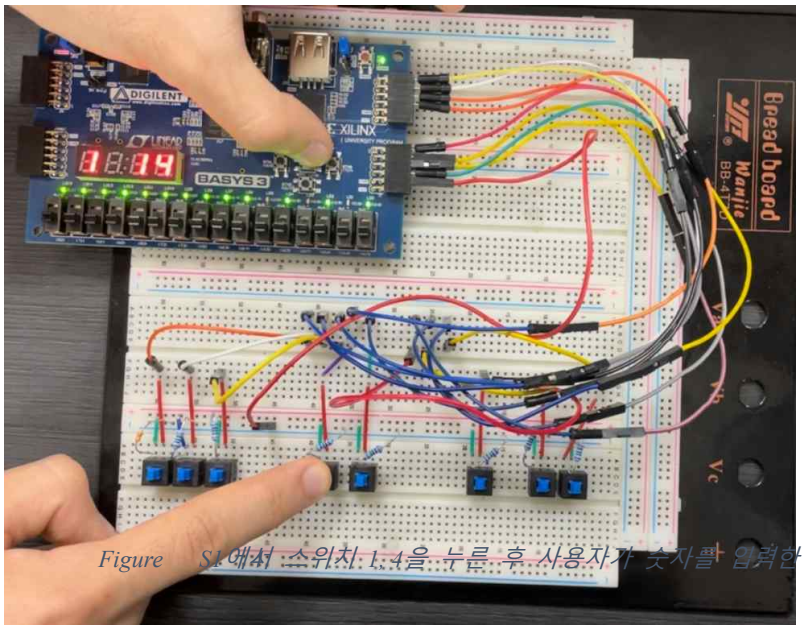


Figure S1에서 스위치 1,4을 누른 후 사용자가 숫자를 입력한 상태

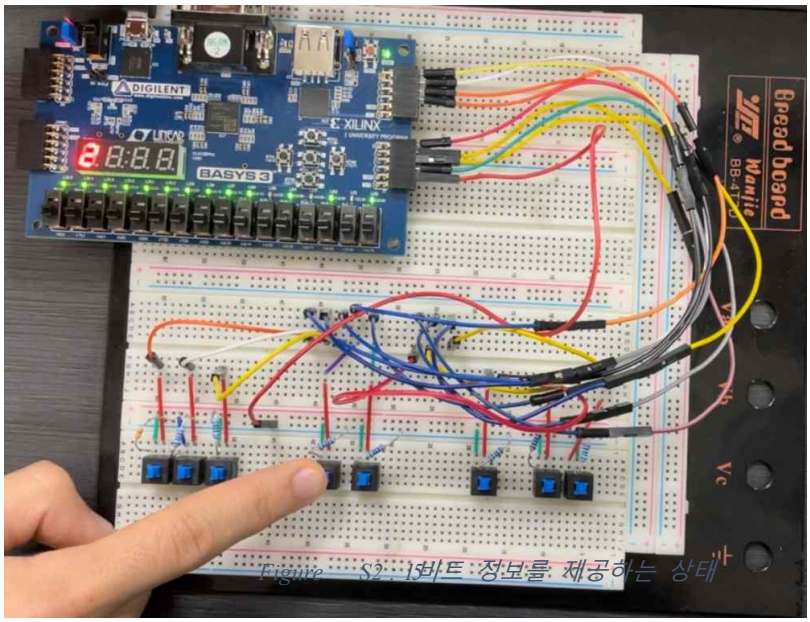
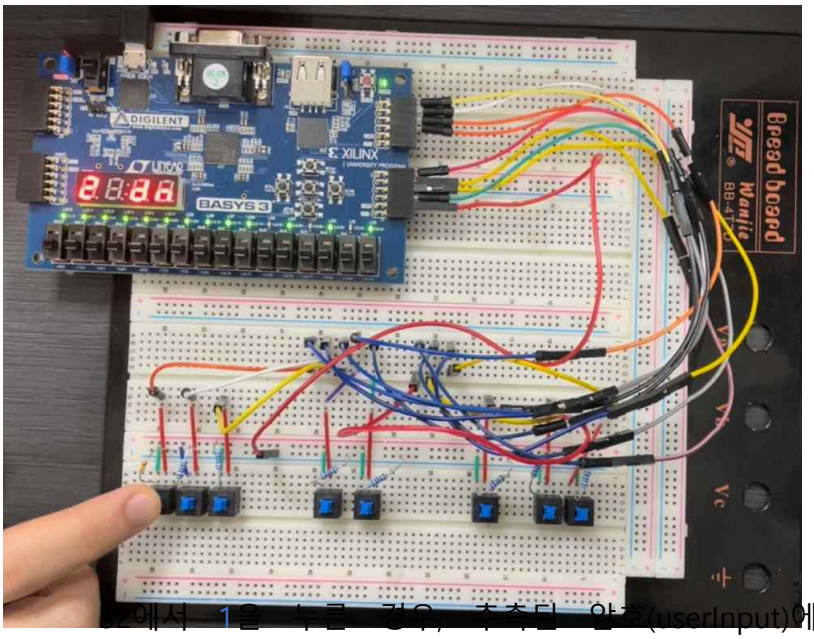
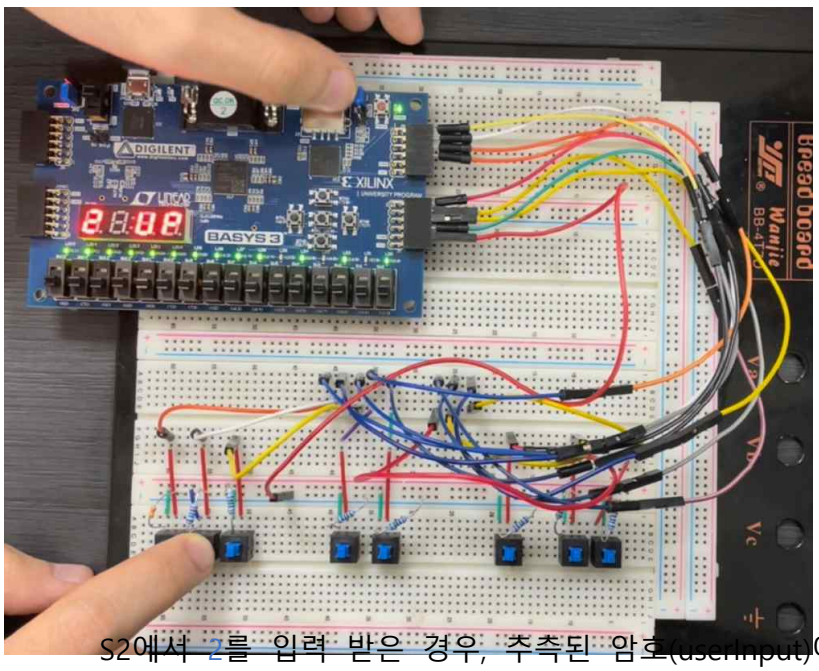


Figure S2 : 3바트 정보를 제공하는 상태



S2에서 1을 누를 경우, 추측된 암호(userInput)에서의 0의 개수와 지정된 암호(targetNumber)에서의 0의 개수를 대소 비교한 결과를 7 segment에 출력한다. 이때 targetNumber의 0이 더 적어 dn (=down)이 출력된다.



S2에서 2를 입력 받은 경우, 추측된 암호(userInput)에서의 0의 개수와 지정된 암호에서의 1의 개수를 대소 비교한 결과를 7 segment에 출력한다. 이때 userInput의 1 개수가 targetNumber보다 더 많아 UP을 출력한다.

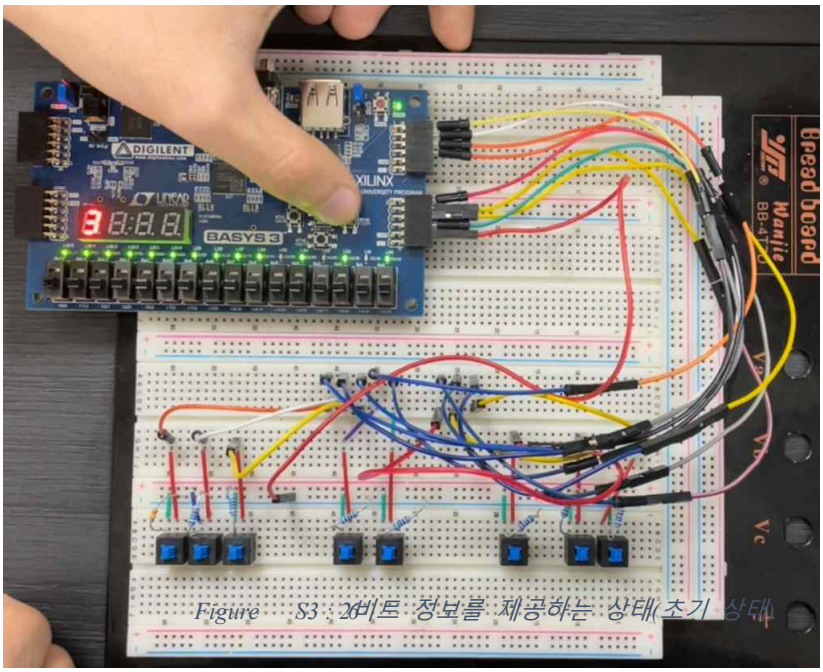


Figure 6 S3 : 2비트 정보를 제공하는 상태(초기 상태)

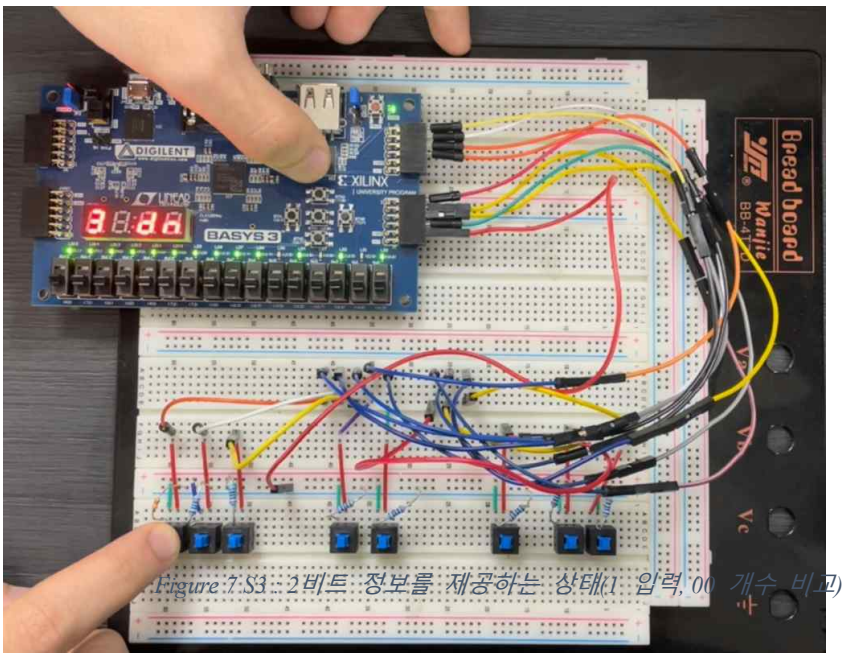


Figure 7 S3 : 2비트 정보를 제공하는 상태(1 입력, 00 개수 비교)

S3에서 1을 입력 받은 경우, 추측된 암호(userInput)에서의 00의 개수와 지정된 암호(targetNumber)에서의 00의 개수를 비교한 결과를 7 segment에 출력한다. 이때 targetNumber의 00이 더 적어 dn(=down)을 출력한다.

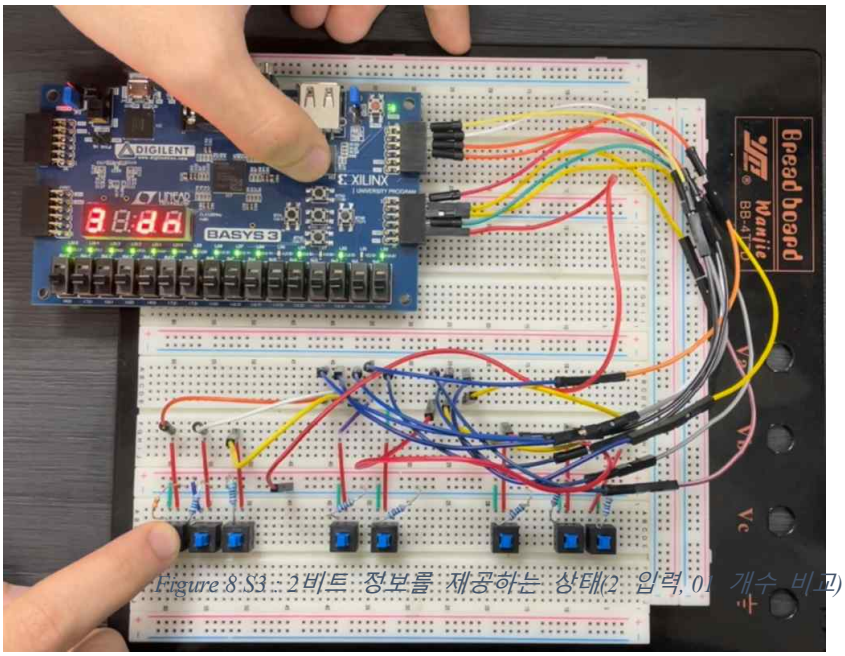


Figure 8 S3 : 2비트 정보를 제공하는 상태(2 입력, 01 개수 비교)

S3에서 2를 입력 받은 경우, 추측된 암호(userInput)에서의 01의 개수와 지정된 암호에서의 01의 개수를 비교한 결과를 7 segment에 출력한다. 이때 targetNumber의 01이 더 적어 dn(=down)을 출력한다.

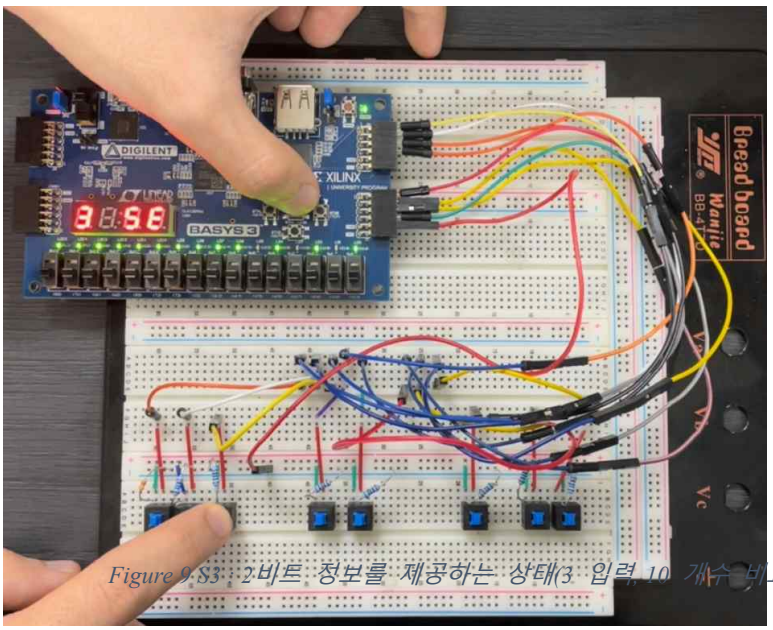


Figure 9 S3 : 2비트 정보를 제공하는 상태(3 입력, 10 개수 비교)

S3에서 3을 입력 받은 경우, 추측된 암호(userInput)에서의 10의 개수와 지정된 암호(targetNumber)에서의 10의 개수를 비교한 결과를 7 segment에 출력한다. 이때 userInput의 10이 targetNumber과 같아 SE(=same)을 출력한다.

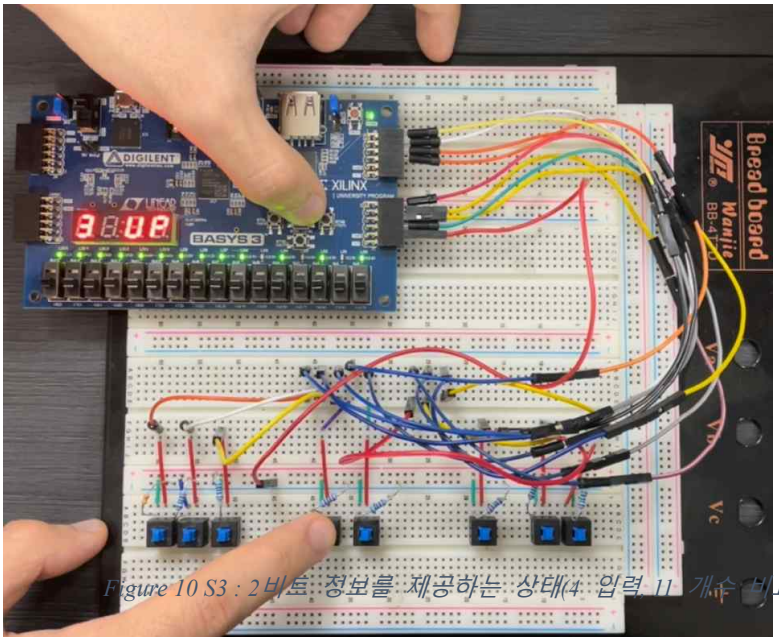


Figure 10 S3 : 2비트 정보를 제공하는 상태(4 입력 11 개수 비교)

S3에서 4을 입력 받은 경우, 추측된 암호(userInput)에서의 11의 개수와 지정된 암호(targetNumber)에서의 11의 개수를 비교한 결과를 7 segment에 출력한다. 이때 userInput의 11이 targetNumber보다 많아 UP을 출력한다.

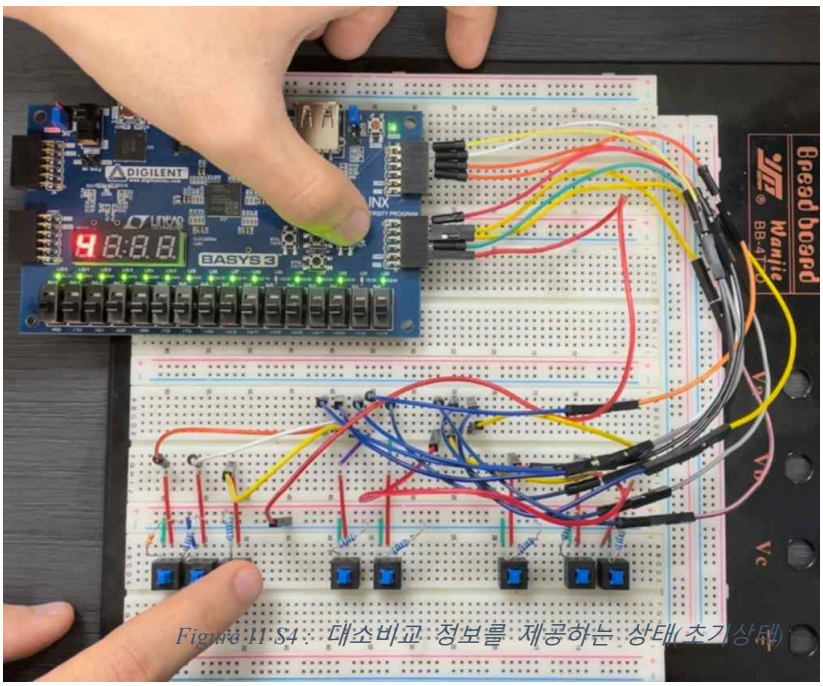


Figure 11 S4 : 대소비교 정보를 제공하는 상태(초기상태)

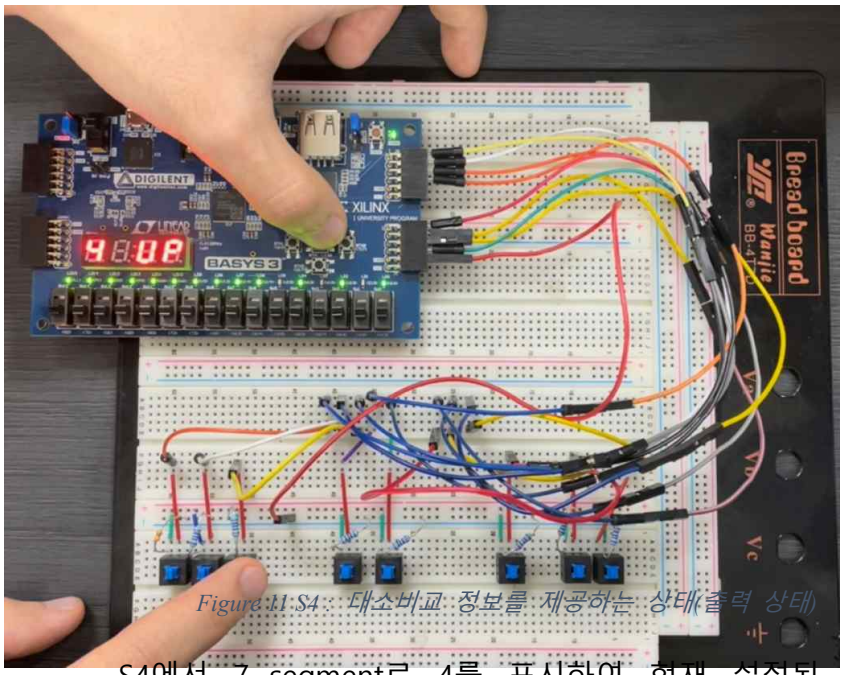


Figure 11 S4 : 대소비교 정보를 제공하는 상태(출력 상태)

S4에서 7 segment로 4를 표시하여 현재 설정된 state가 무엇인지 보여준다. 이때 가운데 버튼을 누른 경우, 추측한 암호(userInput)와 지정된 암호(targetNumber)의 대소관계 정보를 제공한다.

이때 지정된 암호(targetNumber)가 추측한 암호(userInput)보다 크다면 UP을 7 segment에 출력한다.

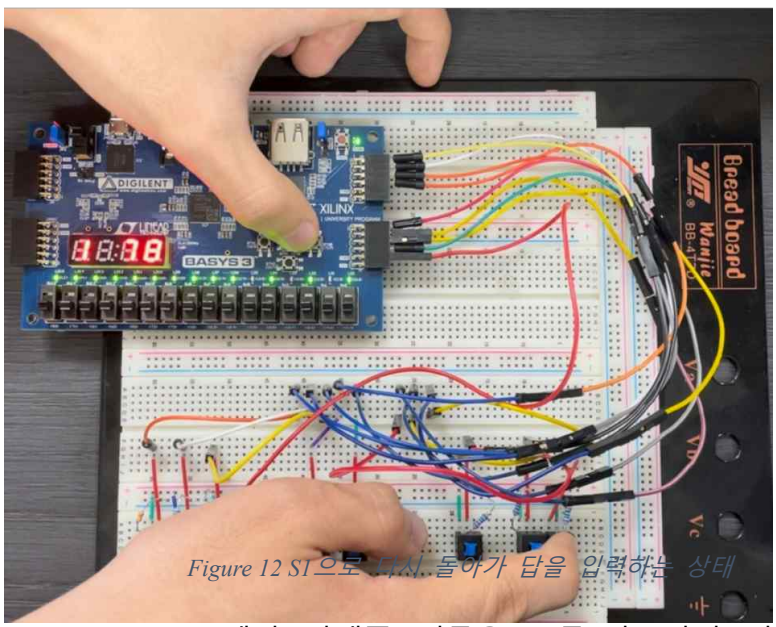


Figure 12 S1으로 다시 돌아가 답을 입력하는 상태

S2, S3, S4에서 아래쪽 버튼을 누른 후 다시 가운데 버튼을 누르면 userInput을 입력하는 S1으로 돌아간다.

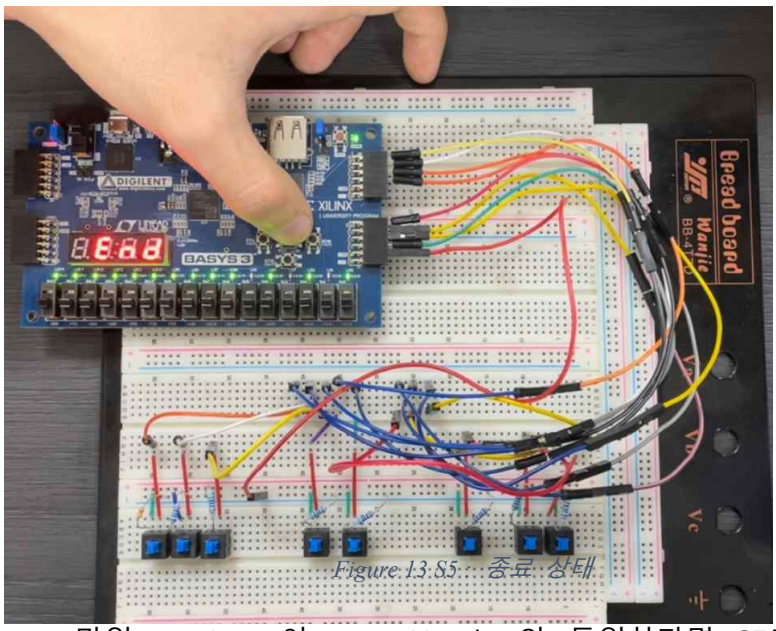


Figure 13 S5 : 종료 상태

만일 userInput이 targetNumber와 동일하다면 S5로 이동하여 End를 출력한다. 만일 다시 반복하고 싶다면 가운데 버튼을 눌러 S0로 이동할 수 있고, 이 경우 다시 clock을 기반으로 무작위 targetNumber를 재설정하여 반복한다.

5. 논의

1) 느낀점

파이널 프로젝트를 통해 수업 시간에 배운 다양한 개념을 실제로 구현하고 응용하는 경험을 쌓을 수 있었다. BCD code, 순차 회로 (sequential circuit), 7 segment display, sequence detector 등을 창의적으로 활용하여 자체 주제를 정하고 그 주제를 구현해가는 과정에서 배운 내용들을 복습할 수 있었습니다. 뿐만 아니라 이를 통해 보다 심화된 내용을 학습할 수 있었다.

또한, 프로그램 작성과 시뮬레이션 수행 및 동작을 확인, 그리고 원하는 방향으로 구현되지 않을 때 디버깅하는 과정에서 실전 경험을 쌓을 수 있었다. 프로젝트 진행 중에 다양한 어려움에 직면하였는데 문제의 원인을 찾고 해결하기 위해 여러 가지 접근 방법을 통해 많은 시도를 해보며 문제 해결 능력을 향상시킬 수 있었다. 마지막으로, 팀원들과 협동하여 프로젝트를 진행하면서 서로 토의하고 의견을 주고받을 수 있어 많은 도움이 되었다.

2) 어려웠던 점 및 해결방법

먼저, 키패드로 입력된 숫자와 암호를 비교하는 로직을 설계하는 과정에서 어려움을 겪었다. 특히, 여러 개의 키패드 입력과 각 자릿수를 비교하는 과정이 헷갈렸었다. 그러나, 위에서 설명한 상태 다이어그램(Fig. 1)을 활용하여 암호 비교 과정을 시각화하였고, 각 상태에서의 키패드 입력과 비교 조건을 명확하게 정의하는 방식으로 문제를 해결할 수 있었다.

또한, 프로젝트 코드 구현 중에 원인을 알 수 없는 버그와 오류가 발생하기도 하였다. 이러한 문제를 해결하기 위해서 코드 구현 방식을 변경하여 문제를 해결하였다. 먼저, FPGA 보드에 미리 코드 동작 수행을 확인한 후, 브레드보드에 키패드를 구현하여 발생한 오류가 코드의 문제인지, 회로의 문제인지를 파악하였다. 코드에 오류가 발생한 경우, 이전의 실습 코드를 바탕으로 디버깅하였고, 브레드보드에 문제가 생겼을 때는, 해당 키패드가 어디로 연결되었고, 코드상에서 어떻게 변화하는지를 되짚어 보며 어려움을 극복하였다

이러한 과정을 통해 키패드와 암호 비교를 수행하는 로직을 성공적으로 설계할 수 있었다. 이번 프로젝트 경험을 바탕으로 더 나은 문제 해결 능력을 갖출 수 있었으며, 앞으로의 프로젝트에서도 이번 기회를 토대로 어려움을 극복하며 발전해 나갈 자신감을 얻을 수 있었다.

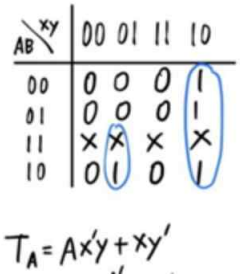
3) 코드 상의 한계, 제한점

특히 코드를 작성하는 과정에서 많은 어려움이 있었다. 익숙한 방식의 코딩을 먼저 시도하고 이후 해당 부분들은 lab에서 다룬 것처럼 gate modeling이나 assign을 통한 문법으로 바꿔보려고 했다. 하지만 작성한 코드에서 constraints file에서의 에러나 bitstream 과정의 에러 등 lab에서 다루지 않은 에러가 발생하는 일이 굉장히 잦았고, multi net과 같은 에러를 이해하고 그것을 피하기 위해 코드를 변형시키는 과정은 많은 시간을 소요하게 했다. 또한, <=와 =의 차이나, always 문 등에 관한 이해가 부족하기도 하여 명확하게 error로 표시되지 않는 것을 검색하면서 수정하거나 더 간결하게 작성하였는데, 그러한 코드 상의 변화 하나하나를 반영하여 시도할 때마다 6분 정도는 걸렸던 탓에 코드 디버깅 후 실행하는 시간만으로 10시간 정도는 보낸 것 같았다. 그 외에도 에러는 발생하지 않았음에도 FPGA에서 오작동하는 것을 보면서 구현의 실수를 이해하기 위해 실습실에서 10시간 정도를 또 보내면서 점차 if else, for문 등을 바꿔나갈 시간이 부족해져서 결국 그러한 제한이 없었으니까 이렇게 간단히 구현하는 것으로 만족하는 것을 선택하게 되었다.

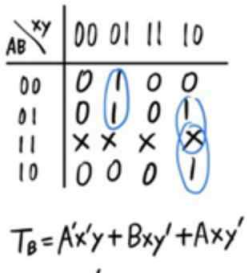
그러한 많은 시간이 소요된 것도 더 일반화되거나 구상하고 있던 추가 기능을 구현하지 못한 것에 아쉬움이 남지만, 특히 그 많은 시간 동안 다양한 에러를 접하면서 이해하지 못하고서 해결하지 못할 것 같은 불안감을 가지게 된 것이 굉장한 스트레스로 다가왔었다. 하루에도 수십 번씩 구현을 포기하는 것을 상상해보면서도 어떻게 최선의 구현을 위해 노력했지만, 예컨대 FPGA switch 입력을 통해서는 정상적으로 출력되던 led가 빵판에 연결한 switch의 입력으로는 값을 반전시켜줘도 전체 led가 다 빛나고 switch를 누르면 더 밝게 해당 led가 빛나는 등의 문제를 더 다루지 않은 것이 아쉽다. 결국 switch 입력을 보여주던 led에 관한 구상은 없던 것으로 하고서 마무리하는 등 아쉬운 것도 많지만, 나름대로 해결책을 떠올릴 수 있었던 것도 많았다. $random이 작동하지 않아 어떻게 무작위 수를 만들 수 있을지 고민하다가 clock 신호를 바탕으로 integer random을 6비트 counter처럼 사용해 해결하거나, multi net 에러를 접하면서 거의 전체 코드를 갈아엎고 Sel1, Sel2, reInput에 관한 always문을 없애 해당 에러를 해결하는 것 등이 그러했다.

시간 상의 이유도 있었고, state1의 입력(스위치 1~8, 그리고 다용도의 Action)을 처리하기 어려워서 state transition table 등도 생략하고, 관련 FF를 사용하는 등의 구현을 코드에는 반영하지 않다. 하지만 부분적으로 state 이동에 관한 if else문을 T FF의 사용 등으로 대체하려는 시도도 고안할 수 있었다. state1의 경우에는 스위치의 입력을 다루는 것이 애매하기 때문에 state2, state3, state4끼리의 이동만을 Sel1, Sel2에 대해서 다룬다면 다음과 같이 표현될 수 있다.

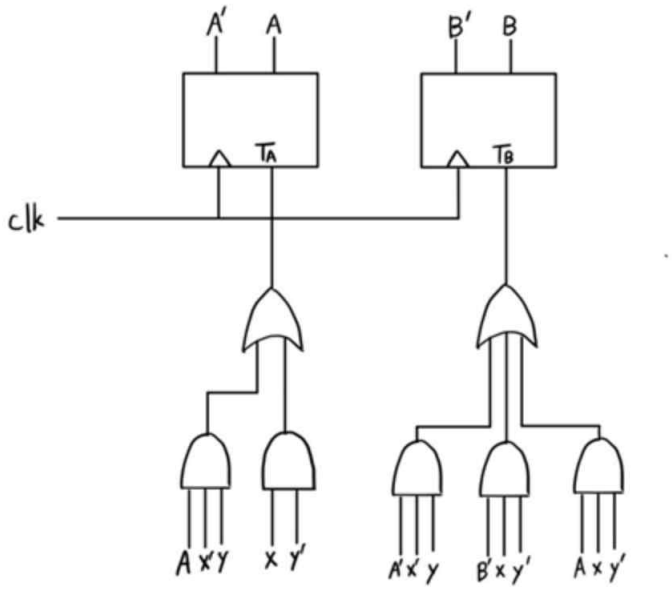
AB	x y	A^+ B^+	T_A T_B
0 0	0 0	0 0	0 0
	0 1	0 1	0 1
	1 0	1 0	1 0
	1 1	0 0	0 0
0 1	0 0	0 1	0 0
	0 1	1 0	1 1
	1 0	0 0	0 1
	1 1	0 1	0 0
1 0	0 0	1 0	0 0
	0 1	0 0	1 0
	1 0	0 1	1 1
	1 1	1 0	0 0



$T_A = Ax'y + xy'$



$T_B = A'x'y + Bxy' + Axy'$



이처럼 코드 상의 if else로 간단히 처리한 부분 역시도 시간만 조금 충분하게 주어진다면, 모두 이와 같이 수업 시간과 lab 시간에 다루었던 방식으로 구현할 수 있을 것이다.