프로젝트 : 1 조

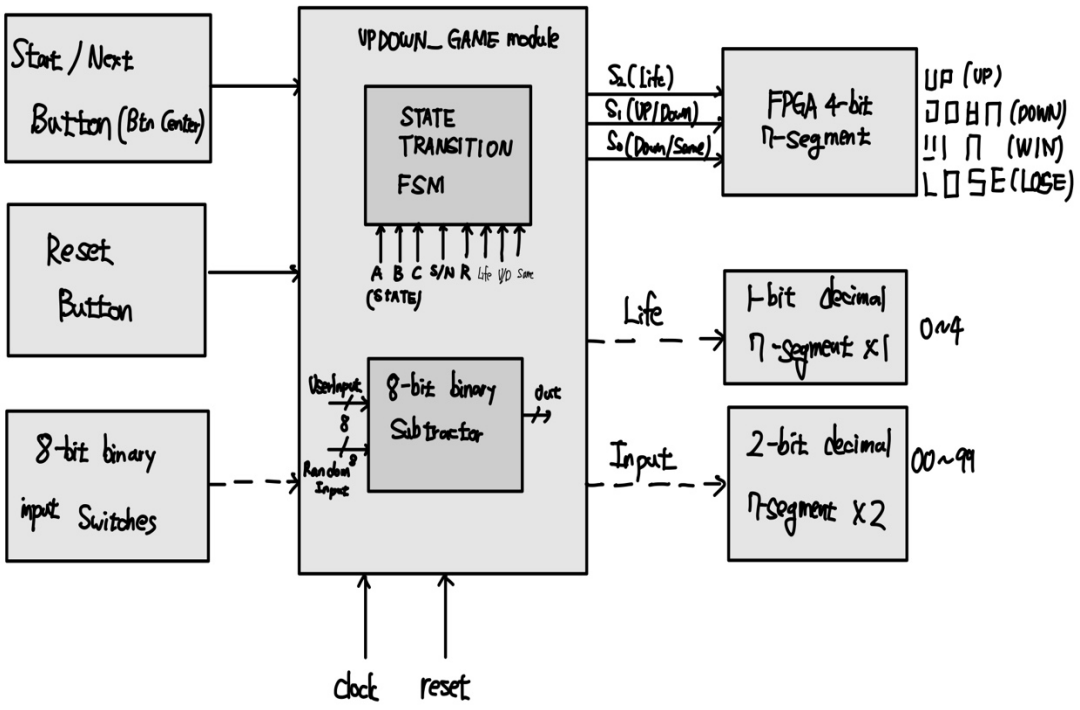
20210170 신재욱

20210661 오승준

20210815 정재현

1. 개요

본 프로젝트로 연구하고자 하는 주제는 '업-다운(Up-down) 게임'이다. 임의의 두 자리 난수를 생성하여 사용자가 숫자를 맞추는 것이다. 게임 시작 시 내부적으로 숫자를 임의로 선정한 뒤 사용자가 숫자를 입력하면 두 수의 대소 관계(up/down/win/lose)를 문자로 출력한다. 이러한 과정을 반복하여 난수가 무엇인지 특정 횟수(Life) 내에 맞히면 승리하는 게임이다. 많이 알려진 게임을 체계적인 회로로 설계하여 실용성을 높이고 흥미를 유발하는 것에 목적이 있다.



< BLOCK DIAGRAM >

2. 프로그램 작동 과정

프로그램의 작동 과정을 요약하면 아래와 같다.

- 값의 초기화

이 단계는 시작 혹은 리셋버튼을 통한 재시작 및 게임이 종료되어 다시 시작될 때 호출된다. life=4, 10 digit =0, 1 digit =0 으로 초기화 해주며 7-segment 로 해당하는 값을 나타내준다.

- 난수 생성

프로그램을 시작하면, 게임에 사용될 무작위 난수 하나를 생성한다. rand 함수를 사용하여 임의의 00~99 사이의 수를 생성하고, 라이프의 수를 4 로 초기화한다.(총 4 번의 입력기회가 주어진다.)

- 값 입력

난수가 생성되면, 멤브레인 3x4 키패드를 사용하여, 사용자가 10 의 자리 수와 1 의 자리 수를 입력하여 입력한 수를 완성한다. 10 의 자리 수 숫자에 10 을 곱하고, 1 의 자리 수와 더하여 00~99 사이의 수를 완성하고, 랜덤으로 생성한 수와 사용자가 입력한 수를 2 진수로 변환한다. 그리고 값을 해당하는 7-segment 에 출력해준다.

- 값 비교 및 반복

해당 단계 전에 life 수를 1 줄이고, 업데이트 된 life 수를 7-segment 에 나타내준다. 8-bit Subtractor 을 구현하여 기본 입력에 랜덤으로 생성한 수를 빼게 되고 나온 결과를 분석한다. 이때, 나온 결과를 8x1 mux 와 연결하게 되는데 이때, selection input 에 life 와 대소비교(up/down), 대소 비교(same/down)로 구성되어 있다. 먼저 아까 8-bit Subtractor 을 사용하여 나온 결과 중 MSB-LSB 순으로 A[7]~A[0]이라고 할 때, selection input 의 MSB 인 S2 는 Life 의 수를 할당한다. 이때 라이프가 1~4 인 경우 0 과 연결되고 0 인 경우 1 을 연결한다. 그리고 두번째 selection input 은 대소비교인데, A[7]이 selection input 의 S1 을 할당한다. 이때 A[7]이 1 인 경우 2 의 보수 연산에 따라 음수이기 때문에,up 을 나타내고, 0 인경우 양수이기 때문에 down 을 나타낸다. 그리고 S0 같은 경우 not 게이트와 연결하여, ~A[7]와 AND 게이트와 다음과 같이 묶이게 되는데, 이때 A[7]을 제외한 나머지가 or 게이트로 묶이게 된다. A[6]|A[5]|A[4]|A[3]|A[2]|A[1]|A[0]과 연결하게 된다. 이렇게 되면, 먼저 ~A[7]의 결과가 1 이고 나머지 A[6]|A[5]|A[4]|A[3]|A[2]|A[1]|A[0]가 1 일 때 A[7]~A[0]가 0 이 아니기 때문에 결국

down 을 나타내고, 0 일 경우 A[7]A[6]|A[5]|A[4]|A[3]|A[2]|A[1]|A[0]이 0 을 나타내므로 same 을 나타내게 된다. 그래서 8x1 의 결과 아래의 표처럼 나타낼 수 있다.

S2	S1	S0	Output	설명
1	1	X	Lose 를 나타냄	라이프가 0 인 경우 이기 때문에 lose 를 나타낸다.
1	0	1		
X	0	0	Win 을 나타냄	라이프가 존재, 입력 값과 랜덤 값의 뺄셈의 결과가 양수인데, S0 에 의해 입력 값과 랜덤 값이 같으므로, same 을 나타내기에 Win 을 나타낸다.
0	0	1	Down 을 나타냄	라이프가 존재, 입력 값과 랜덤 값의 뺄셈의 결과가 양수인데, S0 에 의해 입력 값과 랜덤 값이 다르므로, 입력 값>랜덤 값 이므로 down 을 나타낸다.
0	1	X	Up 을 나타냄	라이프가 존재, 입력 값과 랜덤 값의 뺄셈의 결과가 음수이므로, 입력 값보다 랜덤 값이 큰 걸 의미하므로 up 을 나타낸다.

이러한 과정을 거치고 나서, Up, Down 에 해당하는 state 에 도달하면 라이프의 값을 줄여준다. 이때, 7-segment 로 각 단어를 출력을 한다. 16x1 mux 을 사용하여 WIN/LOSE/UP/DOWN 을 나타내고, 그리고 사용자의 입력을 2 개의 7-segment 로 나타내고, 나머지 7-segment 하나로 라이프를 출력해준다. 이후, 다시 처음 상태로 돌아가 사용자의 입력을 받아온다.

- 종료 후 재시작

그리고 라이프가 0 (life = 1)이 되어 lose 가 된다면 특정 시간동안 딜레이를 걸어 출력하고 값을 초기화 하여 처음 상태로 돌아가 게임을 다시 시작한다. 그리고 win 의 경우에도 특정 시간동안 딜레이를 걸어 출력하고 값을 초기화 하여 처음 상태로 돌아가 게임을 다시 시작하게 된다. 이때 재시작 단계에서는 life=4, 10 digit =0, 1 digit =0 으로 초기화 해준다.

설명을 요약을 하자면 아래와 같다.

0-1) Life 의 수(7-segment 1)을 4 로 초기화 하고 7-segment 에 출력을 하고 10 의자리수 1 의 자리 수 모두 0 으로 초기화 하고 7-segment 에 출력해준다.

0-2) rand() 함수를 사용하여 두 자리 자연수를 생성한다.

1) Basis-3 의 스위치를 사용하여 10 의 자리 수와 1 의 자리 수를 입력 받는다. 이때, 2 진수의 입력으로 준다. 0000~1001 의 사이의 스위치 입력

2) 입력 받은 값과 랜덤으로 생성한 값을 2 진수로 변환하여 저장해준다.

3) 2 진수로 변환한 값을 7-bit 에 1-bit 부호를 추가한 8-bit 이므로, 8-bit Subtractor 를 이용하여 2 의 보수를 사용한 뺄셈을 한다.

4) 이에 나온 결과를 A[7]을 MSB 로 하여 A[7]~A[0]을 받는데 이것을 8x1 mux 와 연결하게 된다. 이때, S2 는 라이프의 수(1 이상=1,0=0)을 나타내고, S1 은 A[7]을 받아, 이때, A[7]이 1 인 경우 2 의 보수 연산에 따라 음수이기 때문에,up 을 나타내고, 0 인경우 양수이기 때문에 down 을 나타낸다. 그리고 S0 같은 경우 not 게이트와 연결하여, ~A[7]와 AND 게이트와 다음과 같이 묶이게 되는데, 이때 A[7]을 제외한 나머지가 or 게이트로 묶이게 된다. A[6]|A[5]|A[4]|A[3]|A[2]|A[1]|A[0]과 연결하게 된다. 이렇게 되면, 먼저 ~A[7]의 결과가 1 이고 나머지 A[6]|A[5]|A[4]|A[3]|A[2]|A[1]|A[0]가 1 일 때 A[7]~A[0]가 0 이 아니기 때문에 결국 down 을 나타내고, 0 일 경우 A[7]A[6]|A[5]|A[4]|A[3]|A[2]|A[1]|A[0]이 0 을 나타내므로 same 을 나타내게 된다. 그리고, 이 state 에서는 life counter 을 이용하여 life 값을 하나 감소 시키고, 7-segment 의 life 수를 업데이트 해준다.

5) 이때 8x1 mux 의 결과에 따라 selection input 이 0xx 인 경우 lose, 100 인 경우 win, 101 인 경우 up, 11x 의 경우 down 을 나타낸다. 이로 인해 각각에 알맞는 출력을 16x1 mux 을 이용하여, 딜레이를 이용하여 특정 시간 동안 UP,DOWN, 에 맞는 출력을 해준다. 이때 라이프가 1 이상일 경우 life=0)로 가서 다시 사용자의 입력을 받고 아닌 경우(life=1) WIN 에 경우 UP/DOWN 보다 길게 딜레이를 걸고 0)로 가서 초기화를 한다. 그리고 나머지 Life=1 인 경우 LOSE 을 출력하고 UP/DOWN 보다 길게 딜레이를 걸고 0)번째로 가서 초기화를 한다.

2. 이론적 배경 : 실습에 필요한 이론 설명

1) RAM

RAM 은 사용자가 자유롭게 내용을 읽고 쓰고 지울 수 있는 **기억장치이다**. 컴퓨터가 켜지는 순간부터 CPU 는 연산을 하고 동작에 필요한 모든 내용이 전원이 유지되는 내내 이 기억장치에 저장된다. 이번 프로젝트에서 RAM 은 랜덤 난수를 저장하고, 사용자가 입력하는 숫자를 저장하기 위해 사용된다.

2) Down Counter

Down Counter 는 Up Counter 와 다르게 수를 하방으로 셀 수 있는 Counter 이다. Counter 는 clock pulse 를 세어서 수치를 처리하기 위한 논리 회로(디지털 회로)이다. 계수기가 계수한 이진수나 이진화 10 진수가 decoder 를 통해서 7-segment 발광 다이오드에 표시되는 숫자로 변환하여 인간이 알아볼 수 있는 정보가 된다. 또, encoder 가 정보를 2 진수로 변환한 것을 계수기를 통해 계수 처리를 실시할 수 있다. 본 프로젝트에서 Down Counter 는 Up-Down Game 의 남은 life 횟수를 체크하는 데 사용될 것이다. Counter 와 7-Segment 를 연동하여 life 를 시각적으로 보여준다.

3) 7-segment display

7-Segment display 는 7 개의 선분(획)으로 구성되어 있으며, 위와 아래에 사각형 모양으로 두 개의 가로 획과 두 개의 세로 획이 배치되어 있는 디스플레이다. 7-segment display 는 일반적으로 숫자 또는 영문자를 표현하기 위해 주로 사용하는 display 이다. 본 프로젝트에서 7-Segment display 는 입력 받은 숫자 표현, 남은 life 표현, UP/DOWN 표현을 위해 사용된다.

4) 반가산기 (Half adder)

반가산기는 1 비트 이진수 두 개를 입력받아 합과 올림 을 출력한다. 이때 올림은 두 비트가 모두 1 일 때 1 이 되고, 합은 00,11 은 0 이고, 01,10 일 때 1 이다.

5) 전가산기 (Full adder)

전가산기는 두개의 반가산기를 이용하여 회로를 구성할 수 있는데, 차이점이라면

합과 올림을 구할 때 이진수 두 개 뿐만 아니라, 이전 가산기의 Carry 을 받아 연산을 한다는 점이다. 1-bit 이진수 두 개와 이전 가산기의 Carry in 을 입력 받아 합과 Carry out 을 출력한다.

6) N-bit 리플 가산기 / 감산기 (N-bit Ripple Adder / Subtractor)

N-bit 가산기는 N-bit 이진수 두 개를 더하여 N-bit 합 결과와 Carry out 을 출력한다. 여기서 N-bit Ripple Adder 을 구현할 수 있는데, N 개의 Full Adder 를 이어 구현할 수 있다. N 개의 Full Adder 기는 각자 N-bit 중 한 자릿수의 연산을 담당한다. 이때 k 번째 Full Adder 에서 나온 Carry out 은 다음 이어지는 Full Adder(k+1 번째) 의 Carry in 에 연결 되어 있다. 이렇듯 순차적인 회로의 구조로 인해 이전의 Full Adder 의 연산이 끝나야 다음 Full Adder 의 연산을 시작할 수 있다. 결국, 연산하려는 자릿수가 커지면 연산 소요 시간 또한, 늘어난다.

Subtractor 에서 A-B 꼴의 뺄셈은 A+(-B) 로 연산할 수 있다. 따라서 2 의 보수(1 의 보수 + 1)를 사용하여 N-bit 가산기를 사용해 감산기를 구현할 수 있다.

7) MxN 이진 곱셈기 (MxN Binary Multiplier)

MxN binary Multiplier 는 M-bit Multiplicand 와 N-bit Multiplier 을 사용하여 각 자릿수에서 부분 곱을 하여 나온 이진수 N 개를 더하면서 연산한다. 부분 곱은 bitwise AND 연산으로 만들 수 있고, Adder 로 부분 곱을 전부 합하여 결과를 구한다.

8) D-Flip-Flop / T-Flip-Flop

D Flip-Flop 은 clock 신호에 맞춰 입력 D 가 Q 에 반영되는 회로이고, T Flip-Flop 은 clock 신호에 맞춰 입력이 0->1 혹은 1->0 으로 변화할 경우 T 가 1 이 되고 나머지 변화가 없을 경우 0 을 나타내는 Flipflop 이다.

3. 실험 준비 : 회로를 Verilog 로 작성하기까지 과정에서 필요한 수식이나 회로도 첨부 및 설명

1) 입력

- control input은 3개(start, next, reset), output은 3개(S2(Life), S1, S0,)로 나타난다.

- Start/Next: Basis-3 보드의 스위치로 숫자를 최초 입력하거나 모든 입력을 완료하여 값을 보내겠다는 입력/숫자 Basis-3 보드의 하나의 숫자를 입력을 완료하여 다음 자릿수로 넘어가겠다는 입력
- reset: 사용자가 게임을 중단하고, 게임을 다시 시작하겠다는 입력

아래에 나타난 transition diagram 에서 각각의 transition 은 6-bit 의 binary number 로 표현되는데 왼쪽에서부터 start, next, reset, life, up-down, same 에 해당하는 값이 부여된다. 각각의 자릿수가 갖는 값이 무엇을 의미하는지는 다음과 같다.

- start: 전원이 켜지고 처음으로 수를 입력 받거나 입력이 완료된 경우 1, 이외의 경우 0
- next: 입력하는 자릿수가 십의 자리에서 일의 자리로 변경되었을 때 1, 이외의 경우 0
- reset: 게임이 끝나거나 사용자가 강제로 게임을 다시 시작할 때 1, 이외의 경우 0
- life: 라이프가 없을 때 1, 라이프가 존재하면 0
- up-down: 입력한 값이 난수보다 클 때 1, 작을 때 0
- same: 입력한 값이 난수보다 작을 때 1, 같을 때 0

이를 바탕으로 transition diagram 을 작성하면 아래 그림과 같다. 입력된 수와 난수를 대소 비교하여 결과를 보여준 뒤에 다음 state 로 transition 하는 것 사이에 delay 를 두어 사용자가 충분히 결과를 확인할 수 있도록 하였다.

<State Transition Table>

State	A	B	C	Start/next	reset	life	UpDown	Same	State+	A+	B+	C+	output
S0	0	0	0	1	X	0	X	X	S1	0	0	1	
S1	0	0	1	1	0	0	X	X	S2	0	1	0	
S1	0	0	1	X	1	X	X	X	S0	0	0	0	
S2	0	1	0	1	X	0	X	X	S3	0	1	1	
S2	0	1	0	X	1	X	X	X	S0	0	0	0	
S3	0	1	1	X	X	0	1	X	S4	1	0	0	
S3	0	1	1	X	X	0	0	1	S5	1	0	1	

S3	0	1	1	X	X	X	0	0	S6	1	1	0	
S3	0	1	1	X	X	1	0	1	S7	1	1	1	
S3	0	1	1	X	X	1	1	X	S7	1	1	1	
S4	1	0	0	X	X	X	X	X	S1	0	0	1	UP
S5	1	0	1	X	X	X	X	X	S1	0	0	1	DOWN
S6	1	1	0	X	X	X	X	X	S0	0	0	0	WIN
S7	1	1	1	X	X	X	X	X	S0	0	0	0	LOSE

각 state 에 대한 설명

S0 : ready state, life 의 수를 4 로 초기화 하고, 10 과 1 의 자리 수 모두 0 으로 초기화 하고 7-segment 에 각각 값을 출력해준다.

S1 : 10 의 자리 수 입력하고 왼쪽 7-segment 에 입력 수를 출력한다.

S2 : 1 의 자리 수 입력 오른쪽 7-segment 에 입력 수를 출력한다.

S3 : Compare 비교 단계 binary 8-bit 로 UserInput 과 RandomInput 을 만들어 8-bit Subtractor 에 넣어 비교를 한다. 여기서 부터는 life/UpDown/Same input 을 받아 결과에 따라 next state 로 이동한다. 그리고 life 의 값을 1 줄이고 업데이트된 life 를 life 를 나타내는 7-segment 에 다시 출력한다.

S4 : Up 을 나타내는 State 로 FPGA 의 기존 7-segment 4 개에 UP 를 출력한다. 그리고 S1 State 로 돌아가 다시 입력을 받도록 한다.

S5 : Down 을 나타내는 State 로 FPGA 의 기존 7-segment 4 개에 DOWN 을 출력한다. 그리고 S1 State 로 돌아가 다시 입력을 받도록 한다.

S6 : Win(Same)을 나타내는 State 로 FPGA 의 기존 7-segment 4 개에 WIN 을 출력한다. 그리고 S0 State 로 돌아가 초기화 하여 게임을 다시 시작하도록 한다.

S7 : LOSE 을 나타내는 state 로 FPGA 의 기존 7-segment 4 개에 LOSE 를 출력한다. 그리고 S0 State 로 돌아가 초기화 하여 게임을 다시 시작하도록 한다.

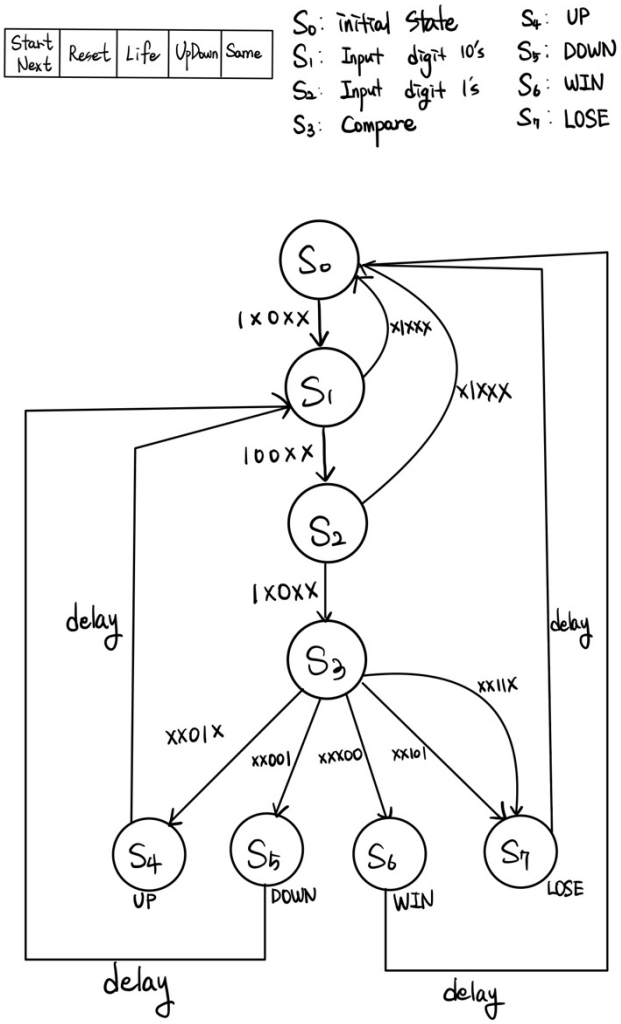


그림 Up-Down Game State Transition Diagram

먼저 state 가 8 개이므로 3 비트를 나타내기 위해 ABC 라는 변수를 사용하였고, 이 State Trasition Table 을 보고, 8 개의 variable (A, B, C, Start, Next, reset, life, Up/Down, Same)에 대하여 D FF 을 구현 할 시, ABC 라는 3 개의 state 에 대해서는 아래와 같은 결과를 얻을 수 있다.

Da 에 대해서 min-term 은 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127 이 나오고, Don't Care 는 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 21, 22, 23, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 52, 53, 54, 55, 64, 65, 66, 67, 68, 69, 70, 71, 84, 85, 86, 87 이 나오고 이를 Quine McCluskey 로 simplification 을 한 결과 DA = A'B'C'D'+A'B'C'F+A'D'E'+A'E'F,+A'BC 이 나오게 되었다.

DB 에 대해서 min-term 은 48, 49, 50, 51, 80, 81, 82, 83, 88, 89, 90, 91, 96, 100, 101, 102, 103, 104, 108, 109, 110, 111, 112, 116, 117, 118, 119, 120, 124, 125, 126, 127 이 나오고, Don't Care 는 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 21, 22, 23, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 52, 53, 54, 55, 64, 65, 66, 67, 68, 69, 70, 71, 84, 85, 86, 87 이 나오고 이를 Quine McCluskey 로 simplification 을 한 결과 A'BDF'G'H'+A'D'E'G'H'+A'CE'G'H'+A'BE'G'H'+A'BC'DF'+A'BCG'H'+A'C'D'E'+A'B'D'E'+A'B'C'D'+A'B'C'F+A'B'CE'+A'BC'E'+A'BCF+A'E'F 이 나오게 되었다.

DC 에 대해서 min-term 은 16, 17, 18, 19, 24, 25, 26, 27, 80, 81, 82, 83, 88, 89, 90, 91, 97, 101, 102, 103, 105, 109, 110, 111, 113, 117, 118, 119, 121, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191 이 나오고, Don't Care 는 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 21, 22, 23, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 52, 53, 54, 55, 64, 65, 66, 67, 68, 69, 70, 71, 84, 85, 86, 87 이 나오고 이를 Quine McCluskey 로 simplification 을 한 결과 DC = A'BDF'G'H+A'D'E'G'H+A'BE'G'H+A'BCG'H+A'BCFH+A'BCFG+A'C'DF'+A'E'FH+A'E'FG+B'D'E'+A'C'E'+B'E'F+B'C'+AB'이 나오게 되었다.

이어서 나머지 input 에 대해서도 구해보면 다음과 같다.

Start/Next = BtnCenter (버튼 Input)

reset = BtnBottom (버튼 Input)

life = dead 가 1 이 되는 경우 life 가 0 이 됨을 나타내고, dead=~life[2]&~life[1]&~life[0]일 경우 1 이 된다. (000 = 0)

아래의 경우 8-bit Subtractor 에서 나온 결과 output 을 wire [7:0] S_out 이라고 할 때,

updown = S_out[7] (이 값이 1 인 경우 최상위 비트가 1 이므로 음수를 나타냄->Up 그리고 0 인 경우 Down 을 나타내며, 단, 입력값-랜덤값이므로 입력값에 비해 랜덤값의 위치를 Up/Down 으로 나타낸다.)

same = (~S_out[7])&(S_out[7]+S_out[6]+S_out[5]+S_out[4]+S_out[3]+S_out[2]+S_out[1]+S_out[0]) 을 나타낸다. 해석하면 S_out[7]에 의해 음수(Up) 이아니고 나머지 &와 ()묶인 부분이 1 이 되는경우 입력값과 다르기 때문에 1 이 되면 UP 이 되고 0 이 되면 Same 이 되게 된다.

2) 출력

대소 비교를 완료하면 그 결과를 UP, DOWN, ... 등 영문으로 7-segment display 에 나타내기 위해서는 BCD-to-seven segment decoder 에 input 으로 들어올 값이 필요하다. 우리는 그것을 각각 I_1, I_0 로 표기하고 binary number 로 $0(00_2)$~$3(11_2)$가 할당되도록 하여 서로 다른 경우를 의미하도록 하였다. I_1I_0 의 값이 나타내는 의미는 다음과 같다.

- I_1I_0 = 00: (Nothing) Letter(Up-Down) 7-segment 에 빈 출력
- I_1I_0 = 01: (Up-down) Letter(Up-Down) 7-segment 에 "UP" or "DOWN" 출력
- I_1I_0 = 10: (Lose) Letter(Up-Down) 7-segment 에 "LOSE" 출력
- I_1I_0 = 11: (Win) Letter(Up-Down 7-segment 에 "WIN" 출력

또다른 7-segment display 에는 남은 시도(목숨)의 수를 출력한다. 이를 L 로 표기한다.

- L: (Life) 7-segment 에 남아 있는 Life 수를 출력 Life=0(1~4 개),Life=1(0 개)

3)구현한 모듈에 대한 설명

- 16-to-1 MUX

7-segment 가 표현하는 수가 숫자와 알파벳별로 각각 10 가지이기에 이를 출력하기 위해서 16x1 Mux 을 사용하여 출력하도록 한다. 이 장치는 입력 받은 숫자의 0~9 를 7-segment display 에 표시하기 위한 것과 DOWN, UP, WIN, LOSE 결과에 해당한 각 단어를 구성하는 알파벳을 7-segment display 에 표시하기 위한 것, 합하여 2 개가 필요하다.

- 8-to-1 MUX

Compare 단계에서 selection input 에 따라 다른 7-segment 의 출력(DOWN, UP, WIN, LOSE)을 위해서 8x1 Mux 을 구현하였다. 구체적으로 현재 남은 게임 횟수(Life), 입력받은

수와 난수의 일치 여부, 일치하지 않다면 대소 비교 결과의 3 가지를 selection input 으로 받는다. 그리고 data input 으로 4 가지 결과를 주어 그 중 하나를 선택하게 한다.

- Half Adder

Full Adder 을 구성하기 위해서 Half Adder 모듈을 구현하였다.

- Full Adder

Ripple Adder 를 구성하기 위해서 Full Adder 를 구현하였다.

- Ten Multiplier

이전에 구현한 HA, FA 을 사용하여 사용자로부터 입력을 받으면 10 의 자리 수와 1 의 자리 수로 구성되어 있는데, 이를 더하기 위해서 10 의 자리 수에 10 을 곱해주는 과정이 필요하다. 그래서 0~9 인 4bit 이진수를 00, 10, 20, ..., 80, 90 으로 만들어준다.

- Ripple adder

입력 받은 수와 난수 값을 생성하고자 할 때, 십의 자리에 10 을 곱한 값과 일의 자리 수를 더한 값을 할당하도록 한다.

- 8-bit Ripple Subtractor

입력 받은 수와 난수의 대소 관계를 파악하기 위하여 감산기를 사용한다. 이 때, 결과가 음수인 경우 down, 0 인 경우 same, 양수인 경우 up 이 된다.

- D Flip-Flop

각 state 에 대하여 값을 저장하기 위해 D Flip-Flop 을 사용한다.

- Digit to binary

난수와 입력 받은 수에 대하여 연산을 수행하기 위해 10 진수 값을 2 진수로 변환하여 준다.

- Up_segment

FPGA 에 있는 기본 7-segment 를 사용하여 UP 을 출력해주며, 7-segment 에서 U=1000001, P=0011000 라는 값을 할당하며 뒤의 두 개는 모두 표시를 하지 않기 때문에 1111111 로 나타낸다.

- Down_segment

FPGA 에 있는 기본 7-segment 를 사용하여 DOWN 을 출력해주며, 7-segment 에서 D=0000011, O=0000001, W=1000000, N=0001001 로 나타낸다.

- Win_segment

FPGA 에 있는 기본 7-segment 를 사용하여 WIN 을 출력해주며, 7-segment 에서 W=1010100, I=1111001, N=0001001 로 하고 뒤의 두 개는 모두 표시를 하지 않기 때문에 1111111 로 나타낸다.

- Lose_segment

FPGA 에 있는 기본 7-segment 를 사용하여 LOSE 을 출력해주며, 7-segment 에서 L=1110000, O=0000001, S=0100100, E=0110000 으로 나타낸다.

- edge_trigger_T_FF

각 state 에 대하여 값을 저장하기 위해 T Flip-Flop 을 사용한다.

- stateTransition module

우리가 본 회로를 구성하기 위하여 사용하는 각각의 state 를 3-bit 이진수로 나타낸다. 그리고 state 사이를 이동할 때 각각에 상응하는 transition 별로 input 과 state 를 구성한 3 개의 1-bit 이진수가 어떻게 변화하는지 Flip-flop 을 이용하여 구성한다. D Flip-flop 을 이용할 때 D 의 input 을 A, B, C(state 를 나타내는 3 자리 수) 및 start, reset, life, updown, same 의 5 가지 입력으로 나타낸다.

- Random module

본 회로에서 난수를 생성하기 위한 module 이다. 0~99 의 범위에서 무작위로 선정하여 변수 Random 에 저장하며 이 값과 입력된 값을 비교하면서 게임을 진행한다.

- outputTotal module

우리가 숫자를 입력한 다음 그 값과 자체적으로 생성된 난수와 비교(state S4~S7 에 해당)하여 나온 결과를 7-segment 에 나타내기 위한 통합적인 모듈이다. 이 모듈에서는 다시 compare 결과(UP, DOWN, WIN, LOSE)를 FPGA-board 에 자체적으로 장착된 7-segment display 에 출력하는 모듈을 호출한다. 또, 남은 life 수와 입력한 수 각각을 외부로 연결된 7-segment display 에 출력할 수 있는 모듈도 호출한다.

- compare_segment module

입력한 값과 난수 사이의 대소 비교에 따라 7-segment 에 표시되는 모양을 제어한다.

- life_segment module

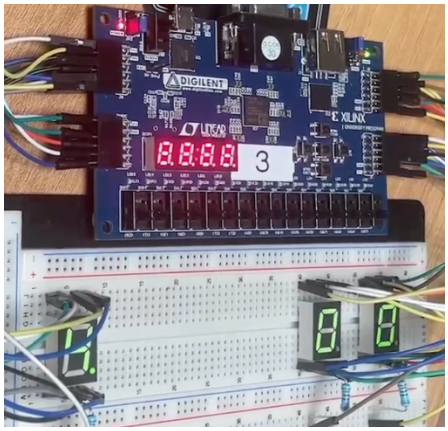
처음으로 4 의 값으로 주어진 life 를 게임 진행 결과에 따라 차감하며 그 값에 상응하는 7-segment 모양을 제어한다.

- number_segment module

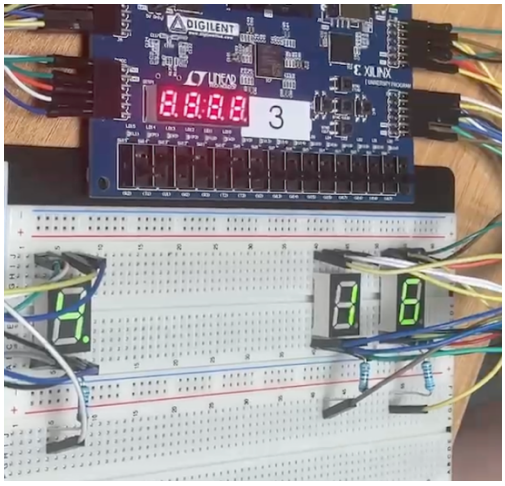
우리가 십의 자리와 일의 자리에 해당하는 숫자를 스위치로 입력하는데 그 값을 직접 7-segment display 에 표시하는 것과 관련된 모듈이다.

4. 결과 : Verilog 로 작성한 회로도와 시뮬레이션 결과, 구현 올바름 설명

- 모든 입출력은 Little-endian 형식으로 표현한다.
- Schematic 기능으로 생성한 회로도 캡처
- 시연하는 결과 출력

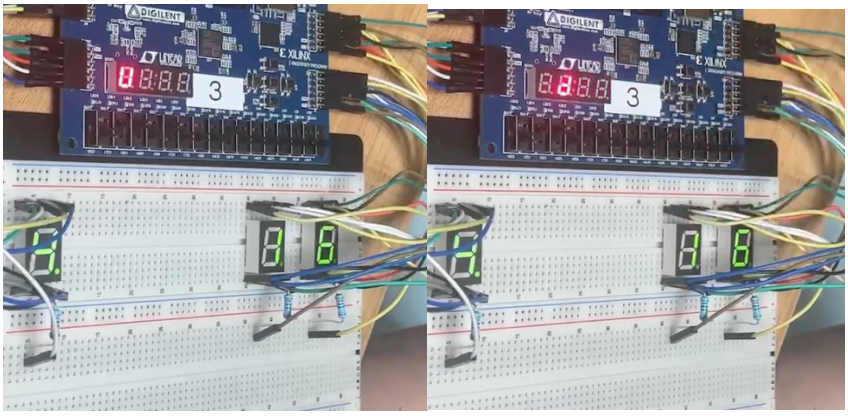


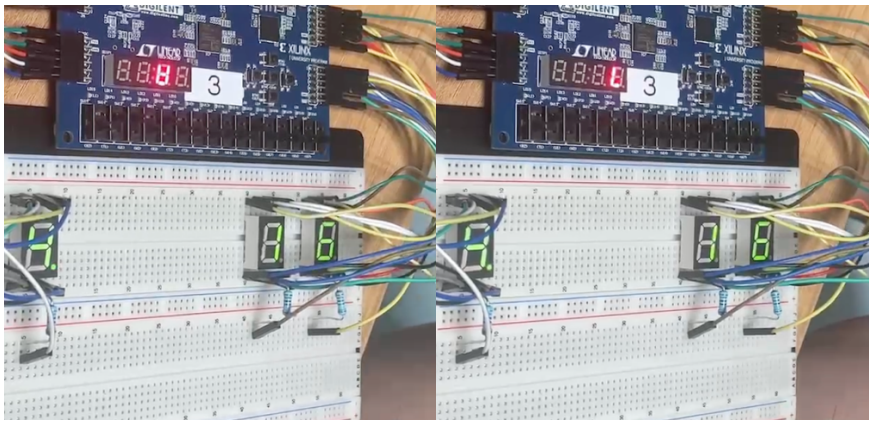
기본 실행화면 왼쪽 7-segment 가 라이프로 4 가 표시되어 있고, 그리고 오른쪽에 있는 것은 순서대로 10 의자리수 1 의자리수 입력이다. 이어서 스위치 왼쪽부터 4 개가 1 의 자리 수 입력, 그 뒤를 이어 또다른 4 개는 10 의 자리 수 입력으로 다음화면에서 정상적으로 숫자가 표시되는 것을 볼 수 있다.



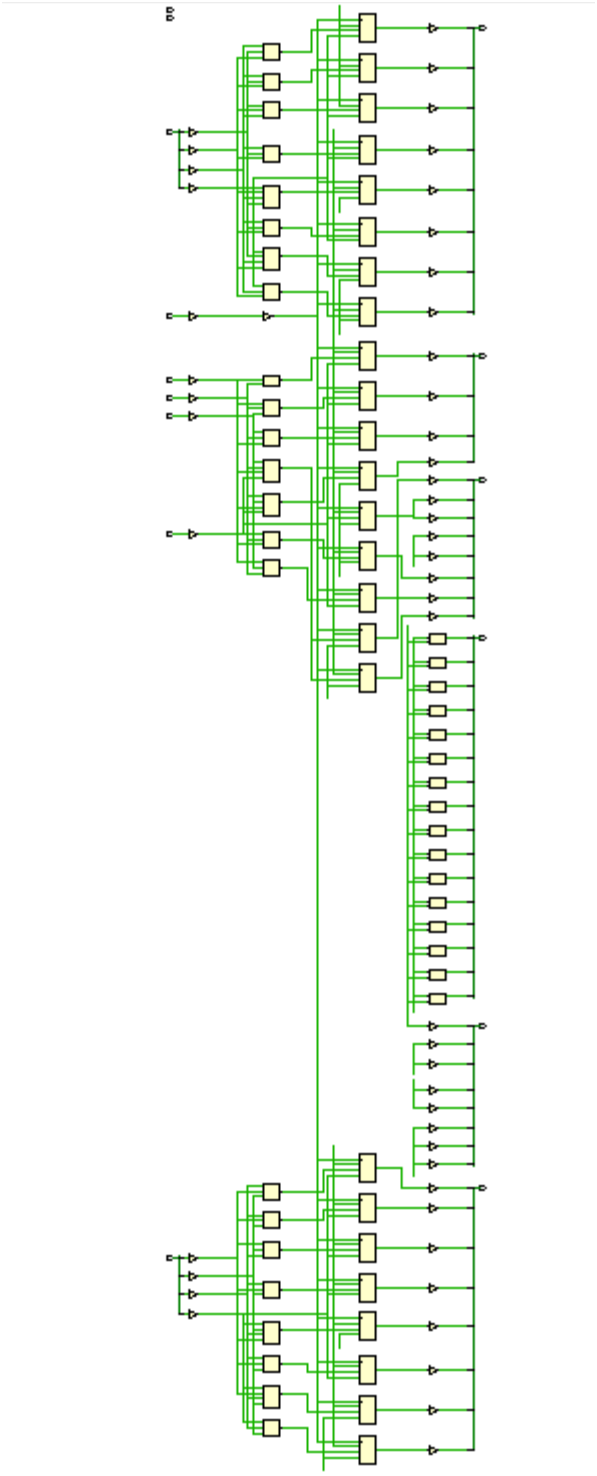
이어서 16 을 입력한 모습이다.

그다음 스위치를 이용하여 Up 의 이니셜 U, Down 의 이니셜 d, Win 의 이니셜 w, Lose 의 이니셜 L 이 정상적으로 출력 되는것을 나타낸 것이다.





그리고 Schematic 을 실행한 결과 다음과 같이 나오게 된다.



5. 변동사항

멤브레인 3x4 키패드를 스위치로 대체

멤브레인 3x4 키패드 형식이 7 개의 핀을 받아서 행과 열을 이용하여 3x4 행렬을 사용하여 input 을 받는데, 기존 basis-3 보드에 연결하면, 회로가 연결되지 않는다. 그래서 vcc 의 7 개의 핀 중 연결을 해보았지만, 해당 핀에 대한 입력이 정상적으로 되지 않아, 결국 스위치를 이용하여 입력을 받게 되었고, xdc 파일에서 스위치의 입력을 끌어서 메인 모듈에서 사용하도록 하였고, 기존 10 진수의 입력이 아닌, 2 진수 값의 입력으로 10 의 자리 수와 1 의 자리 수를 받게 되었다.

Input 에서 Start 와 Next 을 하나의 Input 로 통합

기존의 방법에서는 Start 와 Next 을 분리하여 구현하였지만, 버튼을 2 개 구현해야 한다는 번거로움과 기능이 동일하다는 점을 고려하여 Input 을 하나로 합쳐 Start/Next 로 만들었고, xdc 파일에서 BtnCenter 로 불러와 사용하도록 하였다.

Input 에서 여러 내용 1/0 값과 설명의 변경

기존의 방법에서는 life 의 설명이 게임의 승패가 결정되었을 때로 사용되었지만, 구현하다 보니 승패가 아닌 라이프의 수가 1~4 인 경우 0 을 출력하고, 0 인 경우 1 을 출력하도록 변경하였다. 이어 나머지 up-down/same 이 두개의 input 에 대해서는 비교하는 compare 단계에서 8-bit Subtractor 을 사용하다 보니, up/down 에서 입력 값보다 랜덤 값이 큰 경우 1 이고, 0 인 경우 down 을 나타낸다. 그리고 이어서 same/down 에 경우 0 인 경우 same 이고 1 인 경우 down 로 나타내고 정확한 설명은 앞부분에 나타냈다.

6. 논의 : 느낀 점, 결과가 잘못 나온 경우 원인 분석, 어려웠던 점 및 해결 방법

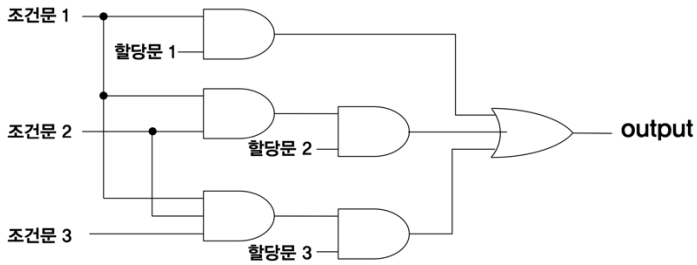
Counter 을 구현하는 과정에서 8 개의 variable 에 대하여 식을 simplification 하는 과정에서 어려움을 겪었다. 변수가 많기에 카노 맵 형식이 아닌 QM 방식을 사용하여야 했는데, 연산을 수행하는 과정이 매우 복잡하였다.

7-segment 를 FPGA 에 연결하는 과정에서, vcc 에 저항을 연결하던 과정에서 저항을 잘못 건드려 7-segment 에 과전류가 흐르게 되어 7-segment 내부 led 가 터지는 문제가 있었다. 하드웨어 회로를 설계할 때는 전류를 차단하고 진행해야 한다는 사실을 상기 할 수 있었다.

Keypad 을 FPGA 에 연결하는 과정에서, 초기에 단순하게 핀 7 개를 FPGA 에 직접 연결하여 입력을 FPGA 에 제공하는 것을 생각하였는데, 이렇게 연결을 할 시 키패드에 전력이 공급되지 않고 회로가 연결되지 않아 아무 기능이 없는 안테나를 연결한 것과 같은 상황이 됨을 알 수 있었다. Keypad 을 제대로 연결하기 위해서는 키패드의 행을 담당하는 핀에 vcc 와 GND 을 연결하고, 열을 담당하는 핀을 FPGA 입력 핀에 연결하여, 전류가 흐르는 행에서 버튼이 눌려 있는 열을 순차적으로 탐색하여 확인하는 방식을 사용해야함을 알 수 있었다.

비교기를 8 비트 감산기로 판단해서, Life/UPdown/downSame 을 판단하도록 만들어보았다. 2 의 보수를 기본으로 판단하기 때문에 기본 자릿 수는 7 비트 (0~99 만 사용)만 사용하지만 최상위 비트를 하나 추가하여 부호를 나타내도록 하였다. 그래서 8 비트 감산기의 결과 중 최상위 비트가 1 이면 음수고, 0 이면 양수이기 때문에 UP/DOWN 을 판단할 수 있고, Compare 단계에서 라이프를 줄여 라이프 수가 0 이라 변수 life 가 1 인 상태에서 비교시 Same 이 아닌 이상 무조건 Lose 가 되고 Same 인 경우 Win 이 되며, 반대로 라이프가 1~4 인 life=0 인 경우 same 의 경우 Win, 나머지 Up/Down 을 판별하도록 만들었다.

결론적으로 말하자면, 메인 모듈을 구현하지 못하였지만, 세부적인 모듈은 모두 구현을 하였고, 하드웨어 적인 오류들도 없었으며 7-segment 의 출력과 switch 와 button 의 사용 모두 문제가 없었다. 덧붙이자면, 기본적으로 모든 자료들과 모듈들을 구현하고 컴파일 까지 오류도 없고, 7-segment 와 switch, 버튼 하드웨어 상으로는 모두 정상적으로 동작함을 확인할 수 있었다, 하지만 메인 모듈에서 구문오류와 더불어 구현에 어려움이 있었다. Run Synthesis 와 Implementation, Generate Bitstream 이 세가지를 거치면서 컴파일을 하면서 고치느라 시간이 오래 걸렸으며, State 가 총 8 개 존재하고, Input5 개 존재하여 프로젝트 아이디어 자체가 어려운 내용이었을 뿐더러, 코드들도 너무 길고 구현하기 어려웠다.



일단 다음과 같이 mux 을 이용하여 If 문을 구현할 수 있으나 못했고, 만약에 If-else 문과 다른 문법들을 사용할 수 있었다 라면 완성을 할 수 있겠으나, if 문 조건도 매우 까다로웠고, 이를 Mux 로 구현하기가 어려웠다. 만약에 if-else 조건문을 사용했다면 조금 더 쉽게 구현할 수 있었을 것이고 시간도 절약했을 것 같았고 이번 과제를 통해 if-else 문이 얼마나 감사한 문법인지 깨닫게 되었다.

난수를 생성하는 과정이 단순히 $random 이라는 함수를 호출하여 생성하는 것이 힘듦을 깨닫고 특정 변수의 값이 변화하는 과정에서 새로운 수를 할당해야 하는 것이 어려웠다. 그리고 always... begin 구문을 D Flip-flip 과 같은 장치를 제외하고는 구성하지 못한다는 제한 하에서 코드를 작성하다 보니 특정 모듈을 state 가 변화할 때마다 호출해야 한다는 것이 어려움을 주었다. 특정 구문 안에서 다른 모듈을 호출하지 못한다는 점, 이미 할당된 변수의 값을 변화시키는 것도 다소 제한되어 있다는 것을 이번 과제를 수행하고 오류를 발견하는 과정에서 알게 되었다.

코드를 작성하는 과정에서 지금까지 다룬 다른 언어와 다른 특성을 갖는 verilog 라는 새로운 언어만이 갖는 문법을 알아가는 과정도 우리만의 과제였던 것으로 느껴졌다. 우리가 이번 학기에 디지털시스템설계 교과목을 수강하면서 작성한 Verilog 파일이 이번 프로젝트 과제보다 쉽게 느껴졌던 것은 아마도 사전에 쓰인 코드를 단순히 활용하는 것이 전부였기 때문이 아닐까 생각이 든다. Module 을 호출하고, module 에서의 Interface 에 필요한 변수에 값을 할당하는 것이 실습 과제의 대부분을 차지하였기에 오히려 우리가 자체적으로 주제를 선정하고 필요한 wire 만을 생성하는 것이 힘들게 느껴졌다고 본다. 한편으로는, 내가 이번 학기에 배운 combinational logic, sequential logic 을 활용하는 device 를 직접 구현하고 그에 상응하는 module 이 하나의 동작을 수행하는데 쓰였다는 점에서 개념을 실생활에 적용하는 능력을 발전시켜 왔다는 느낌을 받았다. 각각이 어떤 특징을 갖고 다른 module 을 정의하는 것에 있어서 어떻게

연관성이 있는지, 그리고 FSM 에서 어느 부분을 담당하게 하여야 할지 등을 우리 스스로 파악하였다는 점에서 이 프로젝트를 수행하며 유의미한 학습을 할 수 있었다.