Lab8. PS/2 키보드

실습 목표

* 주어진 코드를 바탕으로 PS/2 프로토콜을 이해한다.
* PS/2 를 이용해 키보드를 drive한다.

실습 내용과 결과물

NEXYS4 DDR보드에 키보드 장치를 연결해 사용한다. 키보드 장치에 키를 입력하면 NEXYS4 DDR 보드의 7 Segments에 해당하는 make code를 표시한다.

실습 준비물

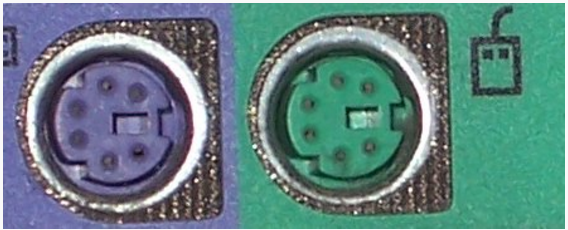
vivado 2016.x 버전 (64bit 운영체제 필요)

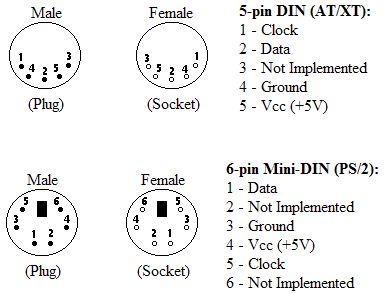
Nexys4 DDR Board

USB 키보드

* **선수지식**
* **시리얼 통신 protocol – PS/2**

PS/2는 직렬 통신의 하나로, 마우스나 키보드 등 user interface device에 사용되는 communication protocol이다. protocol로서는 직렬, 동기식, 양방향의 특성을 가지고 있다. 또한 device가 clock신호를 생성하는 특징을 가지고 있다. PS/2(피에스투)는 IBM에서 1987년에 도입한 Personal System/2의 약어이다. 초기의 PS/2의 connector는 동그란 모양에 6 pin을 가지는데, 최근에는 USB 포트를 이용해 대체하고있다.

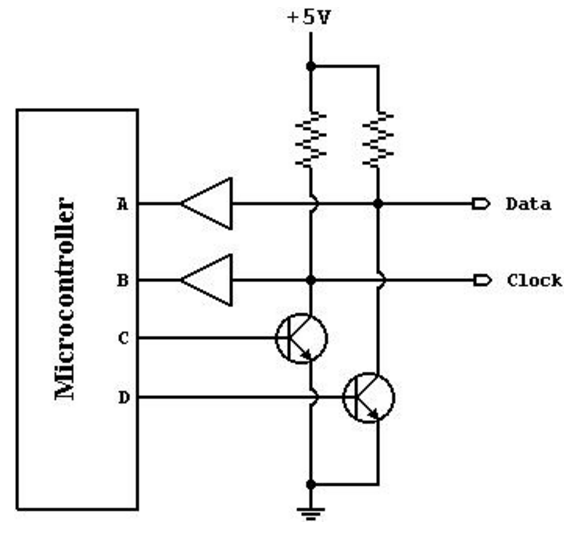


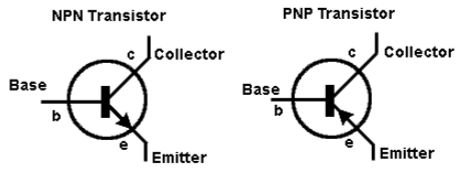


(그림 11.1) 위 : PS/2 port, 아래 : pin map

보라색 : 키보드, 초록색 : 마우스

UART와 비교할 때 다음과 같은 PS/2의 특징들을 주목해야 한다. 첫째, UART가 RX와 TX의 두 data line을 가진 것과 달리, PS/2는 data line이 single이다. 또한 clock신호를 Host가 아닌 device가 생성한다. PS/2의 clock과 data line은 open collector로 설계되어있다. Open collector 구조로 인해 Host는 데이터 전송에 대한 우선권을 가진다. Host가 데이터를 보낼 때는 clock line을 down시켜서 Low가 되게 하는데, device는 이를 detect하고 데이터를 수신할 준비를 한다.

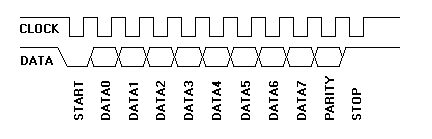




(그림 11.2) 위 : Open collector, 아래 : BJT transistor

(그림 11.2)(위)에서, Microcontroller는 Host를 의미한다. A, B는 각각 data line과 clock line에 연결된 port이다. C와 D는 각각 BJT Transistor의 Base에 연결되어있는데, BJT Transistor는 Base의 전압이 낮을 경우, Collector와 Emitter 사이가 open된다. 반대로 Base 전압이 높을 경우 Collector와 Emittor가 short된다. (그림 11.2)(위)의 도면에서 C와 D신호가 High가 되면 clock line과 data line가 모두 ground에 연결되어 Low가 됨을 확인할 수 있다. 이 같은 구조로 인해 data라인 하나로 양방향 통신을 구현할 수 있는데, 이에 대한 자세한 내용은 코드와 함께 살펴보도록 한다.

PS/2신호의 구성은 UART와 매우 유사하다. Start bit과 data, parity, stop으로 신호가 구성된다. Start bit은 항상 Low이며, data는 8bits으로 구성된다. data는 LSB부터 전송한다. Parity와 stop은 1bit이며, stop bit은 항상 High이다.



(그림 11.3) PS/2 신호

PS/2는 single data line으로 데이터의 송/수신을 모두 구현하기 때문에 Device -> Host, Host -> Device의 규약이 조금 다르다. 일례로, Host -> Device의 경우, Device는 Host의 data를 rising edge에서 읽는다. 반대로 Device -> Host의 경우, Host는 Device의 data를 falling edge에서 읽는다.

* Device -> Host

Read at falling edge

Device는 clock line을 Host가 잡고있는지 확인한다. Host가 clock line을 잡고 있다는 것은 Host가 clock line을 ground에 연결에 Low로 떨어뜨리고 있는 상태이다. 만약, clock line이 High로 50us이상 유지된다면 Host가 clock line을 잡고 있지 않는 상태이다.

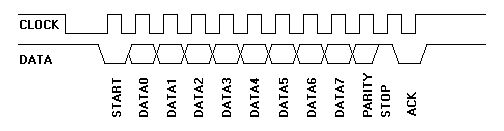
Device는 clock이 High일 때 data를 1 bit씩 LSB부터 전송한다.

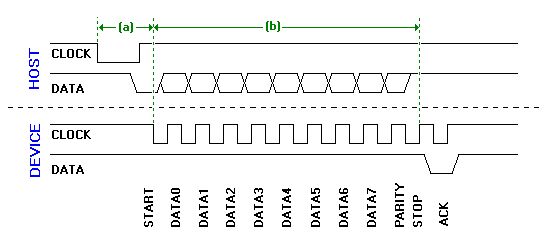
* Host -> Device

Read at rising edge

Host는 clock line을 Low로 떨어뜨려(최소 100us동안) 데이터 전송에 대한 request를 한다. 이후 clock을 release한다.

Host는 clock이 low일 때 데이터를 1bit씩 LSB부터 전송한다.





(그림 11.4) 위 : Host -> Device signal

아래 : Host / Device 의 입장

(그림 11.4)(위)는 clock line과 data line이 Host -> Device 송신시 어떻게 변하는지 보여준다. (그림 11.4)(아래)는 각각의 입장에서 능동적으로 생성하는 신호만을 보여준다. Host는 clock을 low로 지속한 뒤, release한다. Data line은 start bit을 시작으로 1비트씩 순차적으로 전송한다. 마지막에는 High로 stop비트를 전송한다. Device는 Host가 clock line을 low로 지속하면, clock을 생성하고, 데이터를 모두 받은 뒤 ack신호를 생성한다.

* **PS/2 Keyboard**

PS/2 Keyboard는 키보드로부터 PS/2 통신 프로토콜을 통해 신호를 입력받는 모듈이다. 먼저, 이 모듈의 기본적인 동작 과정을 확인한다.

1. 키보드의 키를 누르면 키보드는 Host에게 키에 해당하는 값을 전송한다. 이 값을 make codes라고 부른다. Make codes는 대개 한 바이트 이며, 경우에 따라 extension codes(E0)가 포함된 2 바이트의 make codes를 가지기도 한다.
2. 키보드의 같은 키가 계속 눌러지는 경우, 같은 make codes를 반복해서 전송한다. 이를 typematic state라고 하며, 정해진 typematic delay 마다 한번씩 make codes를 전송하게 된다.
3. 키보드의 키를 떼는 경우(Release), 해당하는 키의 break codes를 전송한다. Break codes는 0xF0 + make codes로 구성된다.

키보드의 각 키에 해당하는 make codes는 다음과 같다.

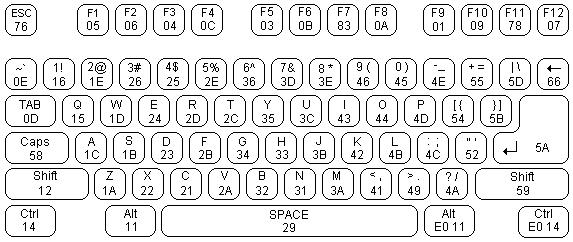
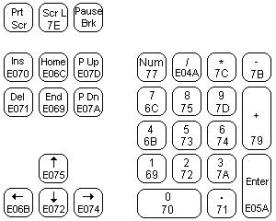


그림 11.5 키보드의 make codes 1

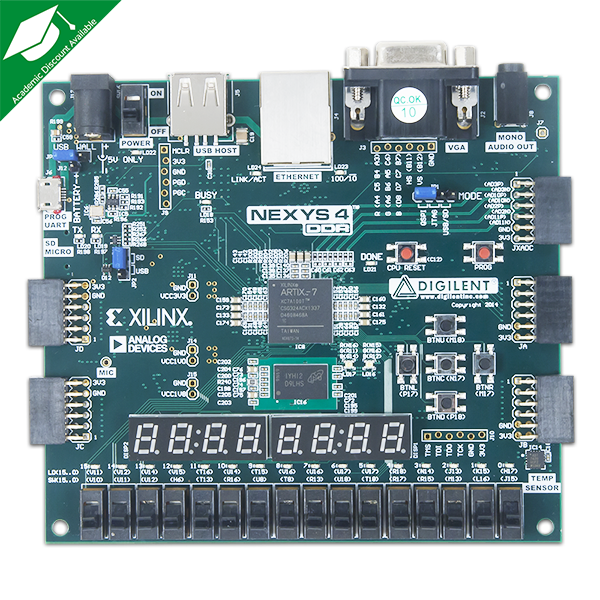


(그림 11.6) 키보드의 make codes 2

자세한 동작은 아래 모듈의 동작은 모듈 설명 파트에서 설명 하도록 한다.

* **USB HOST**

NEXYS4 DDR 보드에는 PS/2 포트가 없다. 대신 USB HOST 포트를 통해 키보드나 마우스같은 장치를 이용할 수 있다.



(그림 11.7) USB HOST PORT

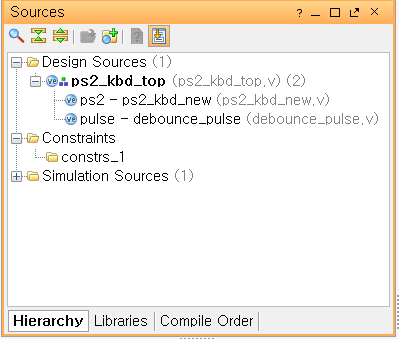
USB HOST PORT를 이용해서 마우스나 키보드 같은 디바이스를 drive하는 것 뿐만 아니라 USB 장치를 이용해 FPGA를 프로그래밍 할 수 있도록 해준다. NEXYS4 DDR 보드 후면에서 찾아볼 수 있는 Auxiliary Function microcontroller (PIC24FJ128)가 이와 같은 기능을 담당한다.



(그림 11.8) PIC24FJ128

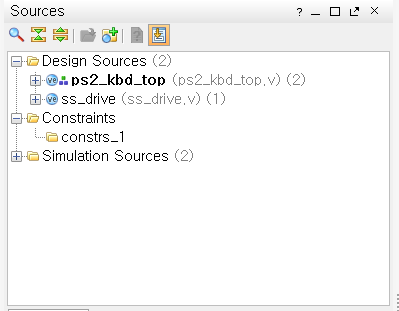
* **실습 따라하기**

먼저 새로운 프로젝트를 생성하고 주어진 소스를 추가한다.



(그림 11.9) Source Window 1

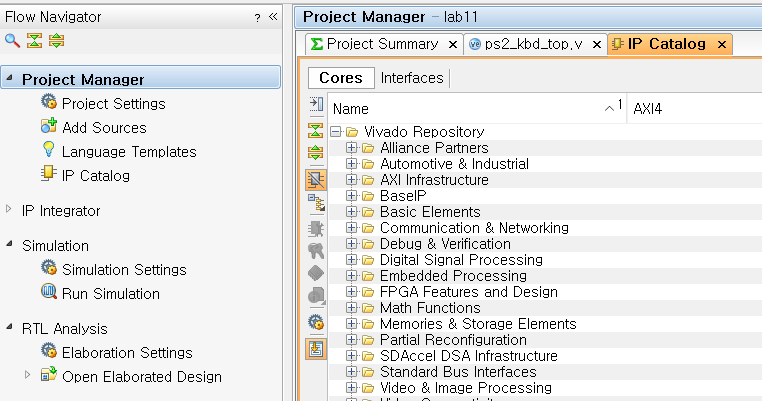
다음으로 Lab3에서 다룬 7 Segments 모듈을 추가한다.



(그림 11.10) Source Window 2

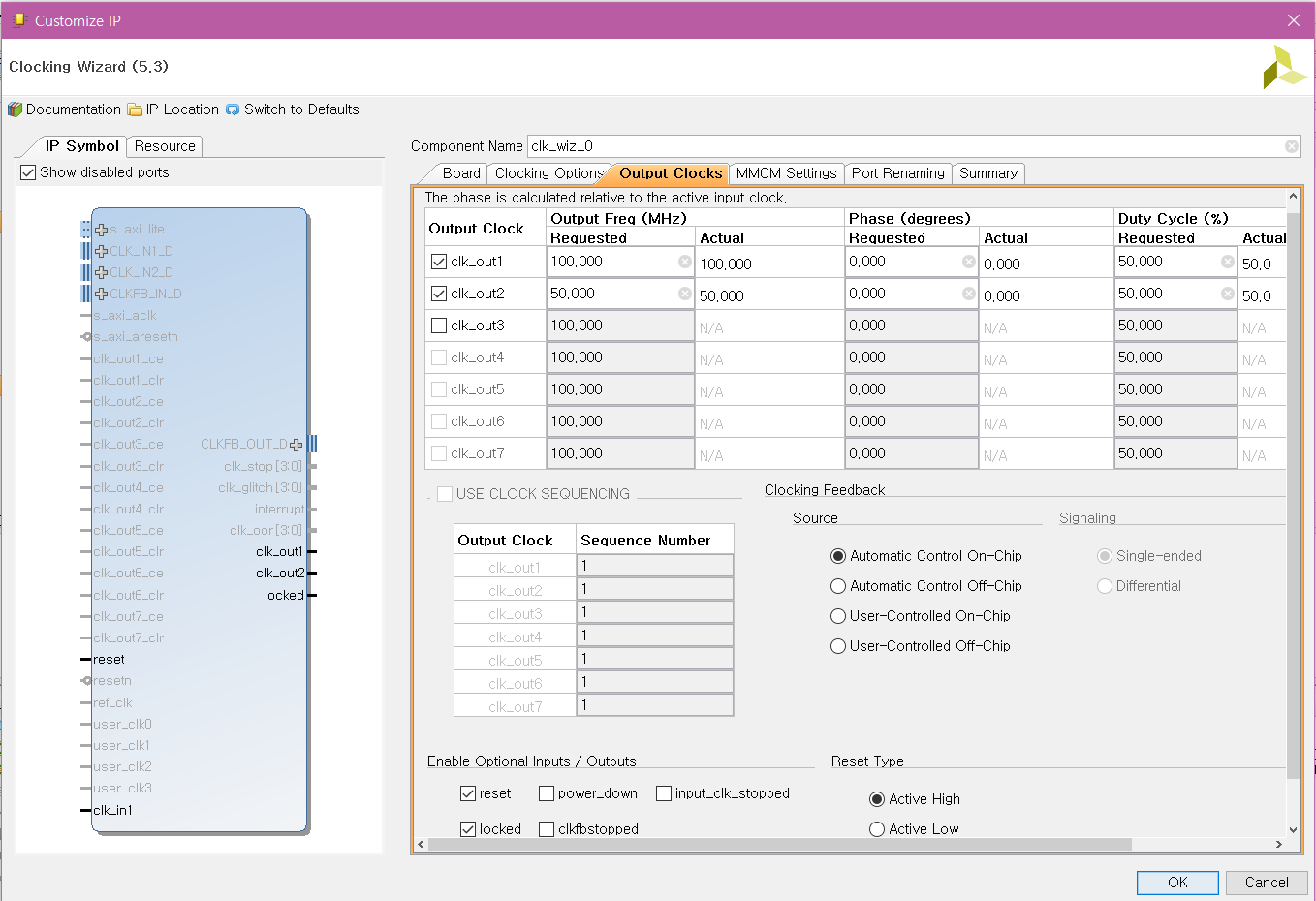
* **CLOCK CORE IP 생성**

본 실습에서 제공한 ps2\_kbd\_top 모듈은 50MHz 클락을 사용한다. NEXYS4 DDR 보드의 시스템 클락은 100MHz이므로, 클락을 분주할 필요가 있다. 클락을 분주할 때는 클락 분주용 IP를 사용해야한다. IP는 Intellectual Property의 약어로, 자주 사용되는 모듈을 라이브러리처럼 가져다 쓸 수 있도록 제공하는 것이다. 간단한 커스터마이징을 통해서 원하는 모듈을 쉽게 얻을 수 있다.



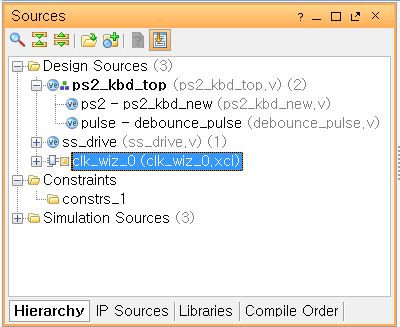
(그림 11.11) IP Catalog

<Flow Navigator>에서 <Project Manager> 하위 항목인 <IP Catalog>를 실행한 뒤, <FPGA Features and Design>을 선택한다. <FPGA Features and Design>하위 항목 <Clocking>을 열고, <Clocking Wizard>를 실행한다.



(그림 11.12) Customize IP

(그림 11.12)와 같이 Output Clocks의 설정을 마치고 <OK>를 누른다. 이후에 열리는 창에서도 <Generate>를 눌러 IP를 생성한다.



(그림 11.13) CLOCK CORE

(그림 11.13)와 같이 IP가 생성되었다.

* **ps2\_example**

새로운 소스파일을 만들고 파일 이름을 ps2\_example 로 한다. 아래 코드를 입력한다. ps2\_example 은 총 3개의 하위 모듈을 가진다. ps2\_kbd\_top 모듈과 ss\_drive 모듈, 그리고 clock 분주 IP이다.

module ps2\_example (

input clk,

input rst,

input ps2clk,

input ps2data,

output [6:0] sseg,

output ssB,

output ssDP,

output [7:0] AN

);

wire [7:0] scancode;

wire err\_ind;

wire clk\_50;

wire clk\_100;

ps2\_kbd\_top ps2\_kbd (

.clk(clk\_50),

.rst(rst),

.ps2clk(ps2clk),

.ps2data(ps2data),

.scancode(scancode),

.Released(Released),

.err\_ind(err\_ind)

);

ss\_drive ss\_drive (

.clk(clk\_100),

.rst(rst),

.data7(),

(중 략)

.data1(scancode[7:4]),

.data0(scancode[3:0]),

.mask(8'b0000\_0011),

.ssA(sseg[0]),

(중 략)

.ssG(sseg[6]),

.ssDP(ssDP),

.AN7(AN[7]),

.AN6(AN[6]),

.AN5(AN[5]),

.AN4(AN[4]),

.AN3(AN[3]),

.AN2(AN[2]),

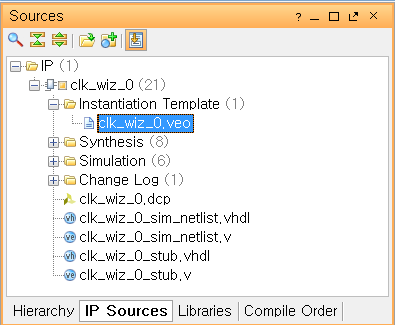
.AN1(AN[1]),

.AN0(AN[0])

);

endmodule

위 코드는 ps2\_kbd\_top 모듈과 ss\_drive 모듈을 instantiation한 것을 보여준다. Clock 분주 IP를 추가하기 위해서 <Source Window>에서 <IP Sources>를 선택한다.



(그림 11.14) IP Sources

(그림 11.14)과 같이 <Instantiation Template> 하위의 <clk\_wiz\_0.veo>를 더블 클릭한다. <clk\_wiz\_0.veo>파일의 72번 라인부터 81번 라인까지의 내용을 복사한다. 복사한 내용을 ps2\_example 내부에 붙여 넣고 다음과 같이 수정한다.

clk\_wiz\_0 clk\_core

(

// Clock in ports

.clk\_in1(clk), // input clk\_in1

// Clock out ports

.clk\_out1(clk\_100), // output clk\_out1

.clk\_out2(clk\_50), // output clk\_out2

// Status and control signals

.reset(rst), // input reset

.locked()); // output locked

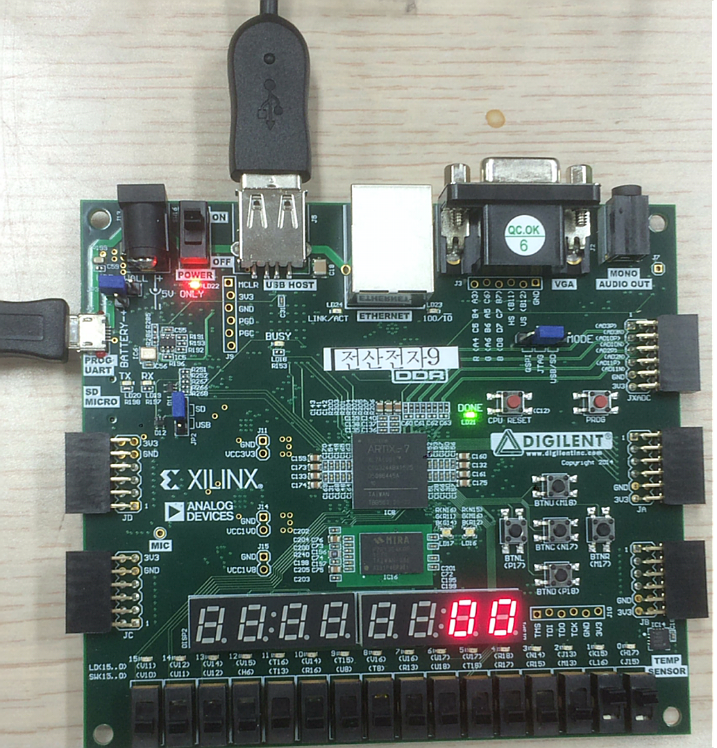
* **동작 확인**

constraints 파일을 생성해서 프로젝트에 맞게 작성한다. USB HOST의 ps/2 pin 이름은 아래 표를 참고한다.

<표 3.1> PS/2 Pin

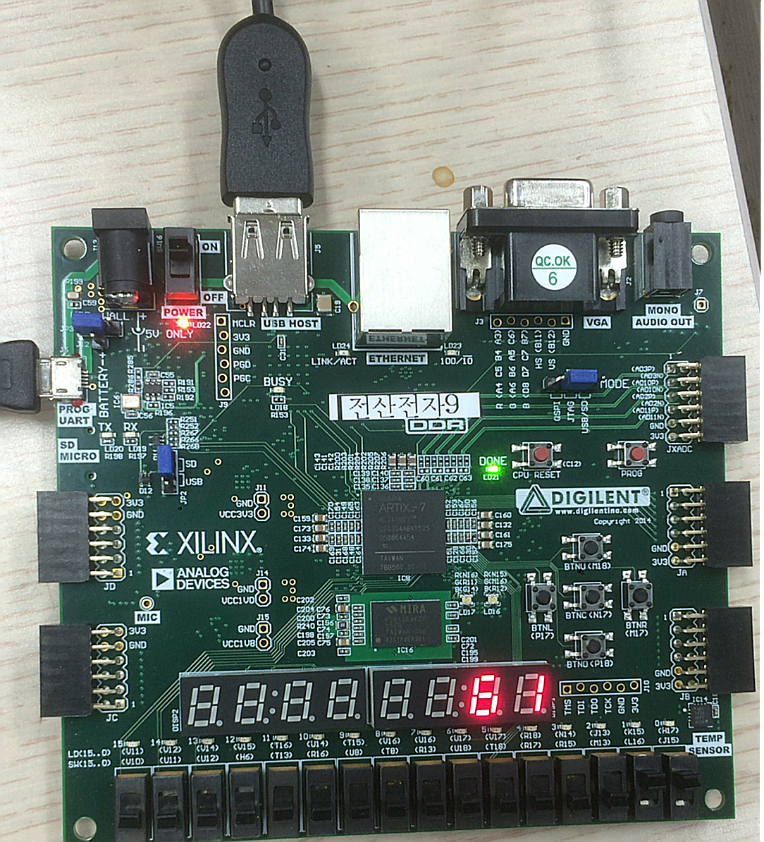
|  |  |
| --- | --- |
| Signal | Pin Name |
| PS/2 Clock | F4 |
| PS/2 Data | B2 |

Constraints 파일을 작성한 뒤 <Synthesis>, <Run Implementation>, <Generate Bitstream> 과정을 거쳐 bit 파일을 생성한다. 생성한 bit 파일을 <Hardware Manager>를 통해 Nexys4 DDR 보드에 프로그램한 뒤 키보드를 USB HOST 포트에 연결한다.

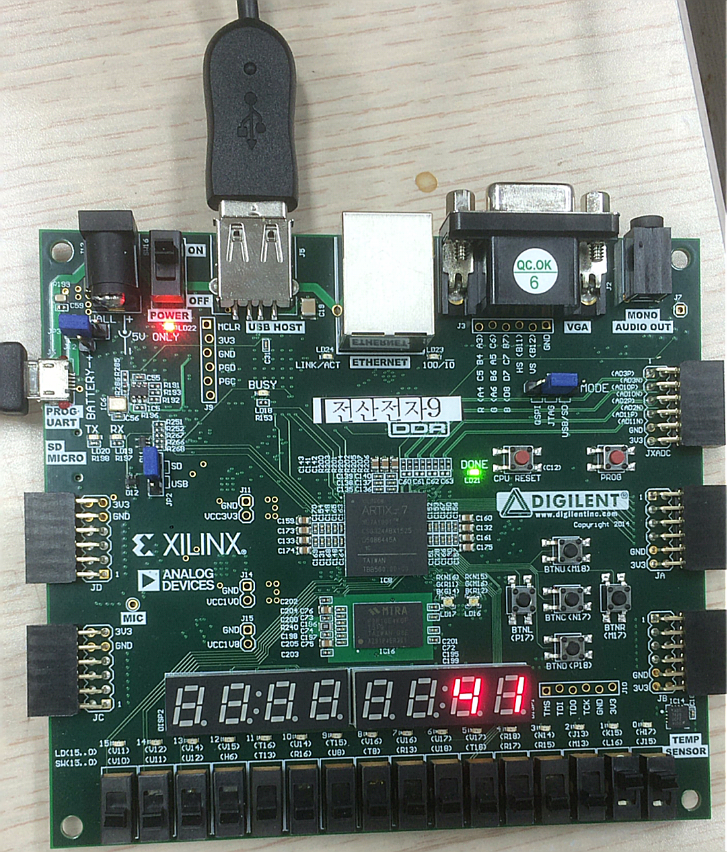


(그림 11.15) 셋팅된 보드

(그림 11.15)의 붉은색 박스로 표시한 부분이 USB HOST단자다. 준비된 키보드(USB port가 달린 것)를 USB HOST단자에 연결한다. 다음, 키보드의 아무 키나 눌러 값이 정확히 전달되는지 seven segments를 통해 확인한다.

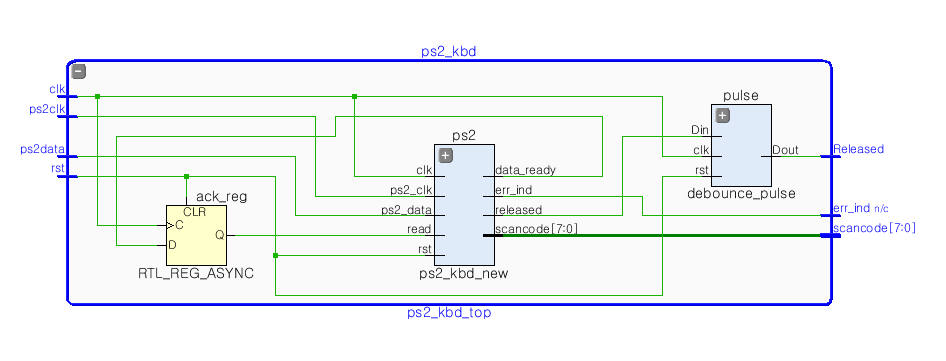


(그림 11.16) ‘a’를 눌렀을 때 보드



(그림 11.17) ‘shift’ + ‘a’를 눌렀을 때의 보드

* **모듈 설명 (source codes)**
* **ps2\_kbd\_top**



(그림 11.18) ps2\_kbd\_top

(그림 11.18)에서 ps2\_kbd\_top 의 핵심 모듈은 ps2\_kbd\_new 모듈이다. ps2\_kbd\_new 모듈의 코드를 살펴본다.

TIMERR\_120U\_BIT\_SIZE는 120us를 count하기 위해 필요한 counter register의 길이를 나타낸다. TIMER\_120U\_TERMINAL\_VAL은 120us가 되는 counter의 값을 나타낸다. ps2모듈에 공급하는 clock은 50MHz인데, 이 clock의 한 주기는 0.02us이다. 120us가 되기 위해서는 6000 period가 필요하다. 6000은 2진수로 나타내기 위해서 13bits가 필요하다. FRAME\_BIT\_NUM은 한 프레임을 구성하는 비트 수를 의미한다. ready\_st과 seady\_ack\_st는 FSM에서 사용하는 state의 이름이다. RELEASE\_CODE는 0xF0로 키를 눌렀다 뗄 때 전송하는 값이다. EXTENDED\_CODE는 extension시 사용하는 값 0xE0를 의미한다.

module ps2\_kbd\_new(

input wire clk,

input wire rst,

input wire ps2\_clk,

input wire ps2\_data,

input wire read,

output reg [7:0] scancode,

output wire data\_ready,

output reg released,

output reg err\_ind

);

localparam S\_H = 2'b00, S\_L = 2'b01, S\_L2H = 2'b11, S\_H2L = 2'b10;

`define TIMER\_120U\_BIT\_SIZE 13

`define FRAME\_BIT\_NUM 11

`define ready\_st 'b0

`define ready\_ack\_st 'b1

`define RELEASE\_CODE 'b11110000 // F0

`define EXTENDED\_CODE 'b11100000 // E0

`define TIMER\_120U\_TERMINAL\_VAL 6000

ps2\_kbd\_new 모듈의 입/출력 신호이다. 입력 신호는 ps2\_clk과 ps2\_data신호가 입력이라는 점이 주목할 만 하다. PS/2 protocol은 device에서 clock신호를 생성하기 때문에 ps2\_clk은 Host입장에서 입력 신호이다. ps2\_data는 양방향 신호이지만, 이 키보드 example에서는 키보드로부터 입력만 받도록 했다. 실제로는 키보드에 대한 setting (typemetic delay값 등)을 Host가 device에게 전송하는 경우도 있다. 입력 read 신호는 키보드로부터 받은 데이터를 Host가 받았음을 알리는 ack신호이다. 출력 [7:0] scancode 는 키보드로부터 받은 데이터이며, data\_ready는 host에게 데이터를 읽어갈 것을 요청하는 신호이다. 그리고 localparameter는 동기화된 ps2\_clk의 상태(state)를 나타내기 위한 것이다.

모듈에서 사용하는 레지스터와 와이어 변수들이다. 28, 29 라인은 state를 저장하기 위한 변수이다. 31번 라인은 ps2\_clk을 동기화하는 데 사용하는 변수이다. 32번 라인은 ps2\_clk의 transition(rising or falling)을 detect하는 데 사용하는 변수이다. 33번 라인의 rst\_timer는 timer counter를 reset시키는 데 사용하는 변수이다. shift\_done은 shift가 완료됨(1 frame을 모두 수신함)을 나타내는 데 사용하는 변수이다. 34번 라인은 수신한 data를 저장하는 shift register이다. 35번 라인은 shift를 하는 데 사용하는 변수이다. PS/2는 data를 1bit씩 송/수신하기 때문에 shift register를 이용해 수신한 데이터를 저장한다. 이에 대한 내용은 뒤에서 다룬다. bit\_cnt 변수는 몇 비트를 수신했는지 count하는 변수이다. 49번 라인의 shift\_flag는 키보드의 ‘shift’키가 눌러진 것을 나타내기 위해 사용되는 변수이다. 그 밖의 변수들은 다음 코드를 보면서 쓰임을 확인하도록 한다.

always @(posedge rst,posedge clk)

begin : sync\_reg

if(rst == 'b1)

begin

ps2\_clk\_d <= 'b1;

ps2\_data\_d <= 'b1;

ps2\_clk\_s<= 'b1;

ps2\_data\_s <= 'b1;

end

else

begin

ps2\_clk\_d <= ps2\_clk;

ps2\_data\_d <= ps2\_data;

ps2\_clk\_s <= ps2\_clk\_d;

ps2\_data\_s <= ps2\_data\_d;

end

end

28 reg [1:0] st,nx\_st;

29 reg [1:0] nx\_st2,st2;

30

31 reg ps2\_clk\_d, ps2\_clk\_s, ps2\_data\_d, ps2\_data\_s;

32 wire ps2\_clk\_rising\_edge, ps2\_clk\_falling\_edge;

33 wire rst\_timer, shift\_done;

34 reg [`FRAME\_BIT\_NUM - 1 : 0] q;

35 wire shift;

36 reg [3:0] bit\_cnt;

37 wire reset\_bit\_cnt;

38 wire timer\_timeout;

39 reg [`TIMER\_120U\_BIT\_SIZE-1:0] timer\_cnt;

40 wire got\_release;

41 wire output\_strobe;

42 reg hold\_release;

43 wire extended;

44 reg hold\_extended;

45 wire err;

46 reg parity\_err,ss\_bits\_err;

47 reg p;

48 reg valid;

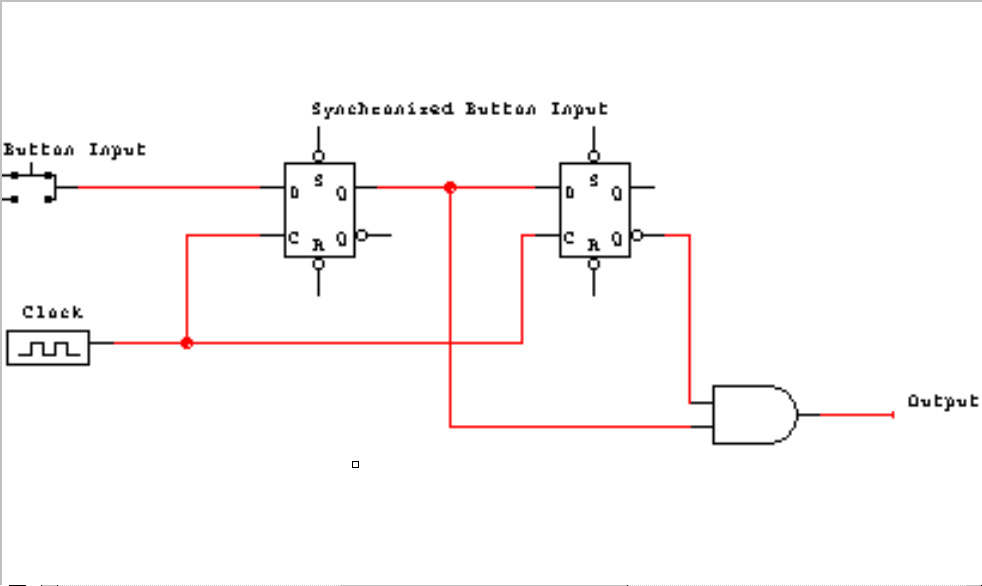
49 reg shift\_flag;

ps2\_data와 ps2\_clk신호를 Host의 clk신호에 맞게 동기화하는 코드이다. 결과적으로 ps2\_clk 신호와 ps2\_data 신호가 각각 D flip flop을 두 개씩 거치는 과정이 된다.

assign ps2\_clk\_rising\_edge = !ps2\_clk\_s & ps2\_clk\_d;

assign ps2\_clk\_falling\_edge = !ps2\_clk\_d & ps2\_clk\_s;

동기화한 신호를 이용해 debounsing pulse를 생성하는 코드이다.



(그림 11.19) debouncing or edge detecting

(그림 11.19)와 같이 D flip flop 두 개를 직렬로 연결한 뒤, 첫 번째 flip flop의 출력과 두 번째 flip flop 출력의 negation한 신호를 and-gating하면 debounce pulse신호를 얻을 수 있다.

always @(posedge clk)

begin : state\_reg

if(rst == 'b1)

st <= S\_H;

else

st <= nx\_st;

end

동기화한 ps2\_clk신호를 FSM으로 나타내 그 상태를 저장하는데, 위 코드는 FSM의 sequential machine part이다.

FSM의 combinational machine part이다.

assign shift = (st == S\_H2L) ? 'b1 : 'b0;

assign rst\_timer = (st == S\_H2L || st == S\_L2H ) ? 'b1 : 'b0;

always @(\*)

case (st)

S\_L : nx\_st = (ps2\_clk\_rising\_edge == 'b1) ? S\_L2H : S\_L;

S\_L2H : nx\_st = S\_H;

S\_H : nx\_st = (ps2\_clk\_falling\_edge == 'b1) ? S\_H2L : S\_H;

S\_H2L : nx\_st = S\_L;

default : nx\_st = S\_H;

endcase

shift는 ps2\_clk을 동기화 state가 S\_H2L일 때 High가 되는 신호이다. 한 비트를 수신할 때 마다 shift register를 shift right하며 데이터를 저장하는데, 이 떄 사용되는 신호다. rst\_timer는 ps2\_clk이 한 번 transition한 뒤의 시간을 측정할 때 사용된다. rst\_timer신호에 의해 timer counter가 초기화되는데, 만약 제 때 초기화되지 않아 timer counter가 일정 값을 넘어가면(120us, 6000) time out error가 발생한다.

always @(posedge clk)

begin

if((rst == 'b1) || (shift\_done == 'b1))

bit\_cnt <= 4'b0;

else if(reset\_bit\_cnt == 'b1)

bit\_cnt <= 4'b0;

else if(shift == 'b1)

bit\_cnt <= bit\_cnt + 'b1;

end

bit\_cnt는 shift가 발생하는 횟수를 count한다. reset\_bit\_cnt는 ps2\_clk신호가 일정 시간 이상(120us) High로 지속되는 경우 high가 되는데, 이 때는 bit\_cnt를 0으로 초기화한다.

assign timer\_timeout = (timer\_cnt == `TIMER\_120U\_TERMINAL\_VAL) ? 'b1 : 'b0;

assign reset\_bit\_cnt = (timer\_timeout == 'b1 && st == S\_H && ps2\_clk\_s == 'b1) ? 'b1 : 'b0;

// 120 us timer

always @(posedge clk)

begin : timer

if(rst\_timer == 'b1)

timer\_cnt <= 'b0;

else if(timer\_timeout == 'b0)

timer\_cnt <= timer\_cnt + 'b1;

end

// shift register for SIPO operation (11-bit length)

// ps2clk의 falling edge 때마다 ps2data를 한 bit씩 shift

always @(posedge clk)

begin : shift\_R

if(rst == 'b1)

q <= 'b0;

else if(shift == 'b1 )

q <= { ps2\_data\_s, q[`FRAME\_BIT\_NUM-1 : 1] };

end

timer\_timeout신호는 timer\_cnt가 제 때 초기화되지 않아 High가 되며, timer\_timeout신호는 앞서 언급한 reset\_bit\_cnt신호가 High가되는 조건으로 사용된다. 135번 라인부터 shift register의 동작에 대한 코드이다. Shift register인 q에 수신한 데이터를 1bit씩 shift right해서 저장한다. 1 frame을 모두 수신한 q는 다음과 같다.



(그림 11.20) Shift Register Q

assign shift\_done = (bit\_cnt == `FRAME\_BIT\_NUM) ? 'b1 : 'b0;

// got\_release: 키보드를 놓았으면 release code(F0)이 전송된다.

assign got\_release = (q[8:1] == `RELEASE\_CODE) && (shift\_done == 'b1) ? 'b1 : 'b0;

assign extended = (q[8:1] == `EXTENDED\_CODE) && (shift\_done == 'b1) ? 'b1 : 'b0;

// output\_strobe: 키보드를 계속 누르고 있는 상태이다.

assign output\_strobe = ((shift\_done == 'b1) && (got\_release == 'b0) && (extended == 'b0)) ? 'b1 : 'b0;

수신한 데이터(q[8:1])에 따라 got\_release, extended 신호를 발생시킨다. Shift\_done신호는 1frame을 모두 수신했다는 신호이다. Output\_strobe는 release나 extended가 아닌 상황에서 high가 된다.

hold\_release와 hold\_extended는 got\_release와 extended신호를 register에 저장한 것이다. 신호를 1 클럭 사이클 이상 유지하기 위해 사용한다.

always @(posedge clk)

begin

if(rst == 'b1)

st2 <= `ready\_ack\_st;

else

st2 <= nx\_st2;

end

always @(st2, output\_strobe, read)

begin

case (st2)

`ready\_ack\_st :

nx\_st2 = (output\_strobe=='b1)?`ready\_st:`ready\_ack\_st;

`ready\_st :

nx\_st2 = (read=='b1)?`ready\_ack\_st:`ready\_st;

default :

nx\_st2 = `ready\_ack\_st;

endcase

end

assign data\_ready = (st2 == `ready\_st) ? 'b1 : 'b0;

always @(posedge clk)

begin

if( rst == 'b1 || output\_strobe == 'b1)

hold\_release <= 'b0;

else if(got\_release == 'b1)

hold\_release <= 'b1;

end

always @(posedge clk)

begin

if( rst == 'b1 || output\_strobe == 'b1)

hold\_extended <= 'b0;

else if(extended == 'b1)

hold\_extended <= 'b1;

end

데이터 수신에 대한 request인 data\_ready 출력 신호와 이에 대한 ack신호인 read 입력 신호에 대한 처리를 담당하는 FSM이다. ready\_ack\_st신호는 일종의 idle state이다. ready\_st는 데이터 수신을 request하는 state이다.

이후의 코드는 일종의 memory이다. 수신한 신호에 해당하는 key의 ascii 값을 저장하고있다. memory에서 주의 깊게 다뤄야 하는 case는 ‘shift’키가 눌러진 상태, ‘extension’을 수신한 상태, 보통의 상태이다.

always @(posedge clk)

begin : send\_output

if(rst == 'b1)

begin

scancode = 'b0;

shift\_flag = 'b0;

released = 'b1;

err\_ind = 'b0;

end

메모리 파트의 도입부이다. Reset 값을 나타낸다.

else if(output\_strobe == 'b1) // keyborad가 눌려있음

begin

scancode = q[8:1];

released = hold\_release;

err\_ind = err;

키보드의 key가 눌러진 경우이다. Default 시 값을 나타낸다.

if(shift\_flag == 'b1) // shift + key

begin

valid = 'b1;

case (q[8:1])

'h1C : scancode = 'h41; // A

'h32 : scancode = 'h42; // B 'h21 : scancode = 'h43; // C

'h23 : scancode = 'h44; // D

'h24 : scancode = 'h45; // E

'h2B : scancode = 'h46; // F

'h34 : scancode = 'h47; // G

( 중 략 )

'h4C : scancode = 'h7A; // :

'h52 : scancode = 'h22; // "

'h41 : scancode = 'h3C; // <

'h49 : scancode = 'h3E; // >

'h4A : scancode = 'h3F; // ?

default :

begin

scancode = scancode;

valid = 'b0;

end

endcase

키보드의 ‘shift’키가 눌러지면 shift\_flag가 High가 된다. 즉, 위 코드는 Shift가 눌러진 상태에서 scancode에 대한 처리이다. Default 시 latch가 발생한다.

if(hold\_release == 'b1) // keyboard에서 손을 뗄 때

begin

valid = 'b1;

case (q[8:1])

'h12 :

begin

shift\_flag = 'b0;

scancode = scancode;

end

'h59 :

begin

shift\_flag = 'b0;

scancode = scancode;

end

default :

begin

scancode = scancode;

valid = 'b0;

end

endcase

end

‘shift’ 키가 눌러진 상태에서, hold\_release가 발생하면 shift\_flag를 clear한다. Scancode는 값을 유지시킨다.

else if (hold\_extended == 'b1)

begin

valid = 'b1;

case (q[8:1])

'h74 : scancode = 'h90; // right arrow

'h6B : scancode = 'h91; // left arrow

'h75 : scancode = 'h92; // up arrow

'h72 : scancode = 'h93; // down arrow

'h70 : scancode = 'h84; // ins

'h71 : scancode = 'h85; // del

'h6C : scancode = 'h86; // home

'h69 : scancode = 'h87; // end

'h7D : scancode = 'h88; // page up

'h7A : scancode = 'h89; // page down

'h4A : scancode = 'h2F; // /

'h14 : scancode = 'h80; // r\_control

'h11 : scancode = 'h81; // r\_alt

'h5A : scancode = 'h0D; // num\_pad ENTER

'h12 : scancode = 'h8A; // print screen

default :

begin

scancode = scancode;

valid = 'b0;

end

endcase

end

‘shift’키가 눌러지지 않은 경우 중에서, hold\_extended가 발생한 경우에 대한 처리이다.

‘shift’z키가 눌러지지 않았으며 hold\_extended도 발생하지 않은 경우로, 가장 보통의 경우에 대한 처리이다. Scancode가 h12이거나 h59이면 키보드의 ‘shift’키를 누른 것으로, shift\_flag를 High로 set한다.

else

begin

valid = 'b1;

case (q[8:1])

'h1C : scancode = 'h61; // a

'h32 : scancode = 'h62; // b

( 중 략 )

begin

scancode = scancode;

shift\_flag = 'b1;

end

'h59 : // shift\_on

begin

scancode = scancode;

shift\_flag = 'b1;

end

default :

begin

scancode = scancode;

valid = 'b0;

end

endcase

end

에러 신호를 발생키는 부분이다. P는 parity를 계산해 저장하는 변수이다. Start와 stop, parity bit까지 총 11 비트의 데이터를 모두 사용해 parity를 계산한다는 부분이 UART와 다르다. 11 비트를 각각 XORing하면 High가 짝수인 경우 parity는 low가 되고, 홀수인 경우 parity는 high가 된다. Parity\_err는 parity의 miss match에 의한 error이며, ss\_bits\_err는 start bit와 stop bit의 miss match에 의해 발생하는 error로 일종의 frame error이다. err 출력신호는 parity\_err와 ss\_bits\_err의 OR로 assign됨을 확인할 수 있다.

always @(q)

begin : err\_chk

p = q[0]^q[1]^q[2]^q[3]^q[4]^q[5]^q[6]^q[7]^q[8]^q[9]^q[10];

parity\_err = ( p == 'b1) ? 1'b0 : 1'b1;

ss\_bits\_err = ( q[0] == 'b1 || q[10] == 'b0) ? 1'b1 : 1'b0;

end

assign err = parity\_err || ss\_bits\_err;

* **과제**
* 보드에 연결된 PS/2 키보드로 받은 값을 UART를 이용해 PC에 전송한다.
* PC는 수신한 데이터를 화면에 출력한다.