Lab8. UART

실습 목표

* 주어진 코드를 바탕으로 UART의 동작을 이해한다.
* Asynchronous 통신의 동작을 이해한다.

실습 내용과 결과물

PC에서 terminal emulation program으로부터 입력된 character를 UART의 RX 모듈을 통해 수신하고, 이 character를 UART TX 모듈을 통하여 시리얼로 PC에 전송한다. 또한, 수신한 character를 7 Segments로 표시한다.

실습 준비물

vivado 2016.x 버전 (64bit 운영체제 필요)

NEXYS4 DDR Board

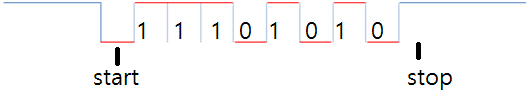
터미널 에뮬레이션 프로그램 (ex. teraterm)

* **선수지식**
* **시리얼 통신 포로토콜 - UART**

UART는 Universal asynchronous receiver/transmitter의 약어이다. UART는 시리얼 통신 혹은 직렬 통신의 하나로, 이는 병렬 통신과 반대되는 개념이다. 시리얼 통신은 한 비트씩 순차적으로 데이터를 보내는 방식이고 병렬 통신은 여러 비트를(대개 수 바이트) 한번에 보내는 방식이다. 시리얼 통신은 병렬 통신보다 전송이 느린 대신 구조가 간단하며, 가격이 저렴하고 만들기가 쉽다. 또한 통신 거리가 멀다. 시리얼 통신의 대표적인 프로토콜은 UART, I2Q(아이 스퀘어 씨), SPI 등이 있다. UART는 직렬 통신 중에서도 비 동기식이며, TTS 신호 level을 사용한다. 이는 노이즈에 취약하기 때문에 RS-232, RS-422, RS-485와 같은 통신 표준과 함께 사용한다. UART 통신 프로토콜은 Stop비트의 길이, parity 비트, 흐름제어, 전송률(baud rate)을 사전에 송/수신자 간 약속해야 한다.

* **UART 신호**

UART는 주로 1 character(1 byte)단위로 데이터를 전송한다.



(그림 8.1) UART 신호의 예

(그림 8.1)에서 1-1-1-0-1-0-1-0은 전송하는 데이터이다. 한 비트씩 순차적으로 한 바이트를 전송하는 것을 확인할 수 있다. 이같이 한 단위의 데이터를 보내는 과정을 한 프레임이라고 한다. 한 프레임의 구성은 다음과 같다.

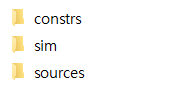
1. Start bit – 신호가 High to Low로 떨어짐
2. Data - 8비트의 데이터가 LSB부터 1비트씩 순차적으로 나타남
3. parity bit – (그림 8.1)에는 나와있지 않지만, error를 체크하기 위해 parity bit를 사용하는 경우가 있음. parity bit은 전송하는 데이터 중 ‘High’ bit의 개수가 홀수인지 짝수인지 표시함.
4. Stop bit – 한 프레임의 전송이 끝남을 알려주는 것으로, 1~2 bit width를 가진다. 전송자와 수신자 사이에 사전에 약속된 길이.

* **Baud rate**

Baud rate는 프랑스 한 전신 기술자의 이름을 딴 것인데, 일반적으로 bit율(BPS, bits/sec)과 같다. 데이터 전송 속도를 의미한다. baud rate가 9600이라면, 1초에 9600비트를 전송한다는 것이다.

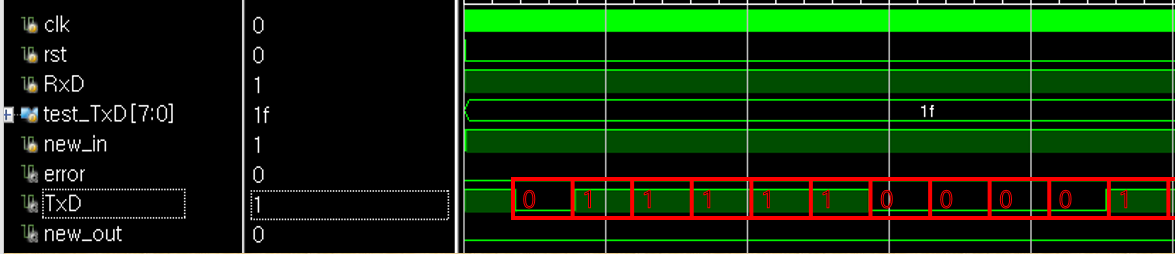
* **실습 따라하기**
* **UART\_top**

프로젝트를 생성하고 주어진 소스파일을 추가한다.



(그림 8.2) Given Files

소스파일을 추가한 뒤, <tb\_UART.v> 파일을 이용해 시뮬레이션을 진행한다.

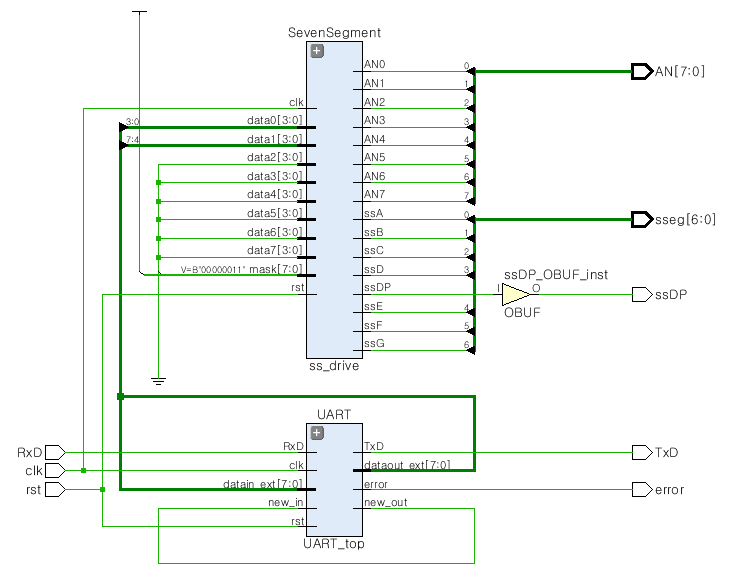


(그림 8.3) UART Wave Form

(그림 8.3)에서, 처음 1비트는 start bit로, High에서 Low로 처음 떨어진 신호이다. 10진수 31은 16진수로는 0x1F에 해당하는데, UART는 LSB를 먼저 전송하기 때문에, 전송하는 데이터의 sequence는 1 – 1 – 1 – 1 – 1 – 0 – 0 – 0 이어야 한다. (그림 8.3)에서, start bit이후로 위의 sequence가 정상적으로 나타나는 것을 확인할 수 있다. 마지막 2 비트는 parity bit와 stop bit이다. Parity bit는 위 design의 경우 내부적으로 odd parity로 되어있다. 전송하는 데이터 ‘d31에는 High가 총 5개 있다. 모듈 내에서 정한 Parity는 홀수로 High이므로, parity를 포함하면 총 6개의 High가 XOR된다. High가 짝수 만큼 있기 때문에 parity bit는 Low이다. Stop bit는 1bit이며, High이다.

* **UART\_example**

UART\_example은 UART모듈을 이용해 PC에서 전송받은 character를 7 Segments에 표시하고, 다시 PC로 전송(echo)하는 모듈이다. 아래 Schematic View를 참고해 UART\_example 모듈을 기술한다.

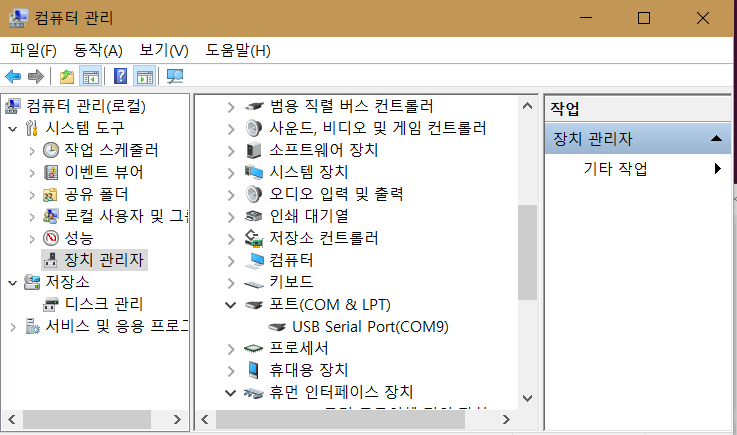


(그림 8.4) UART\_example Schematic View

(그림 8.4)의 UART 모듈의 입/출력을 살펴보면, RxD 입력을 통해 수신한 데이터를 dataout\_ext 출력으로 내보내고, 그 데이터를 다시 datain\_ext 로 되먹이는 것을 알 수 있다. 이는 echo의 기능을 구현한 것이다.

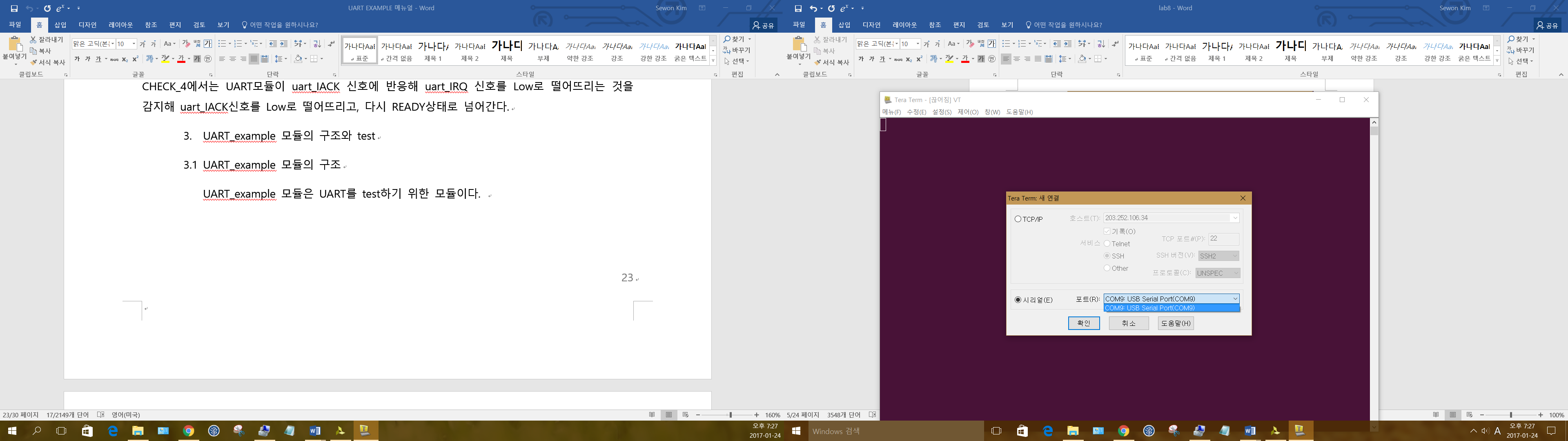
* **동작 확인**

UART\_example 기술을 완료한 뒤, <Synthesis>, <Run Implementation>, <Generate Bitstream> 과정을 거쳐 bit 파일을 생성한다. 생성한 bit 파일을 <Hardware Manager>를 통해 Nexys4 DDR 보드에 프로그램한 뒤 컴퓨터의 <장치관리자>를 실행한다.



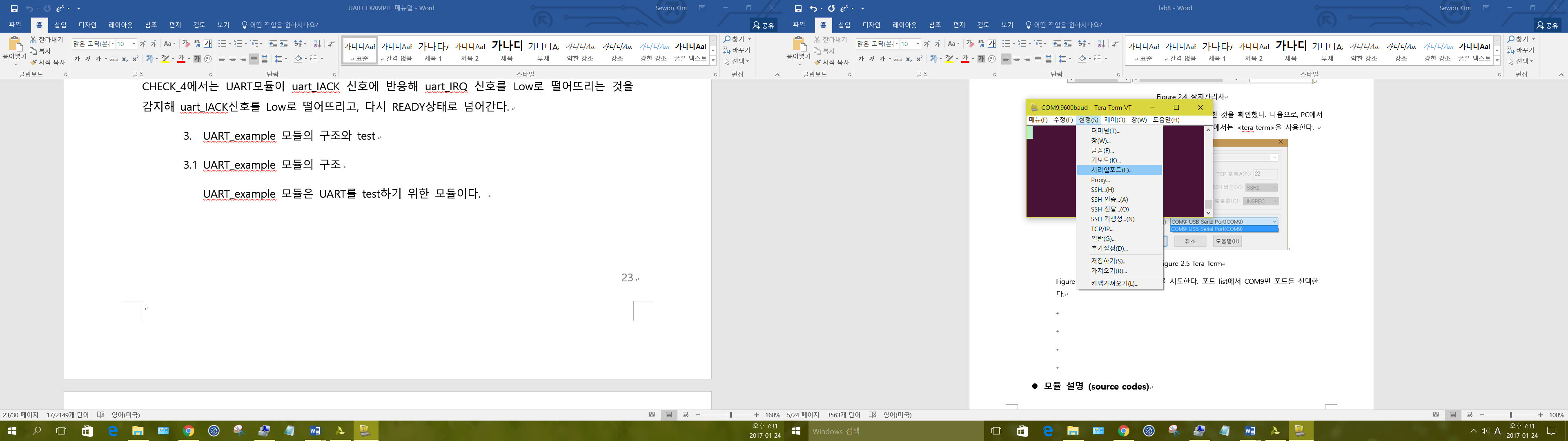
(그림 8.5) 장치관리자

(그림 8.5)에서 USB Serial Port가 COM9번으로 잡힌 것을 확인했다. 다음으로, PC에서 terminal emulator program을 실행한다. 본 실습서에서는 <tera term>을 사용한다.



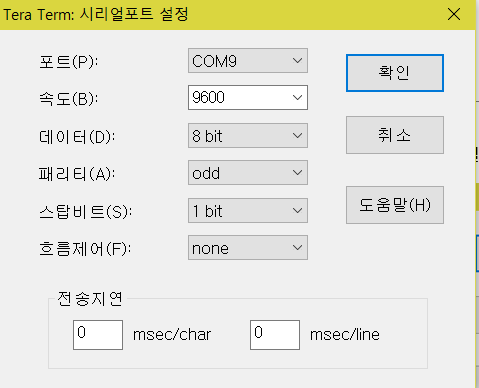
(그림 8.6) Tera Term 시작화면

(그림 8.6)에서, 시리얼 포트 연결을 시도한다. 포트 list에서 COM9번 포트를 선택한다.



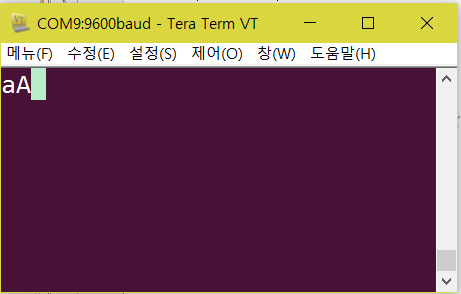
(그림 8.7) Tera Term 설정

(그림 8.7)과 같이 <설정>, <시리얼포트> 를 눌러 시리얼포트 설정 창을 연다.



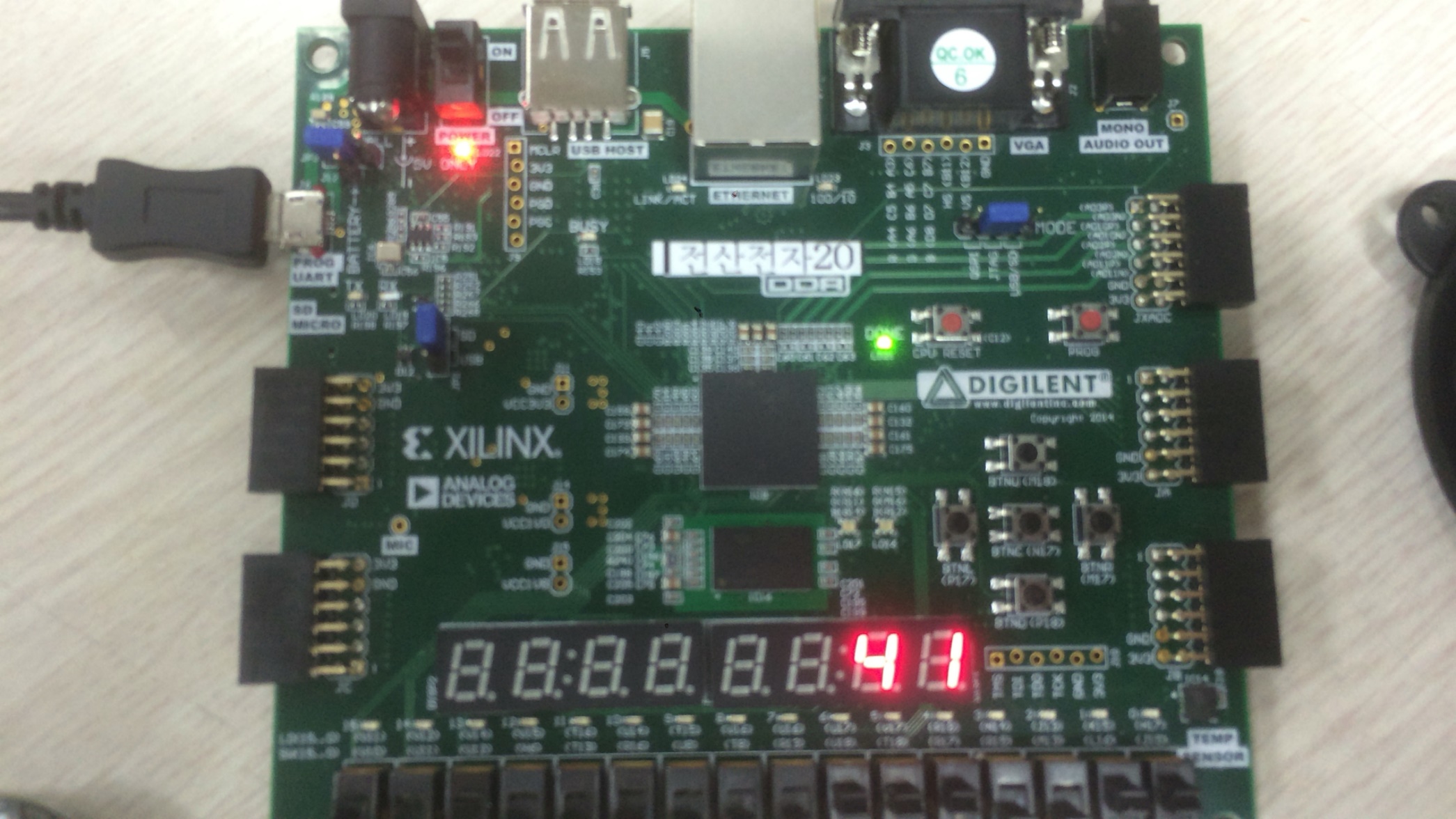
(그림 8.8) 시리얼포트 설정

(그림 8.8)과 같이 설정한다. 전송속도는 9600 bit/s이며, 한 프레임에 8비트의 데이터를 전송한다. 패러티 비트는 odd이며, stop 비트는 1bit으로 한다. 흐름제어는 하지 않는다.



(그림 8.9) 데이터 전송

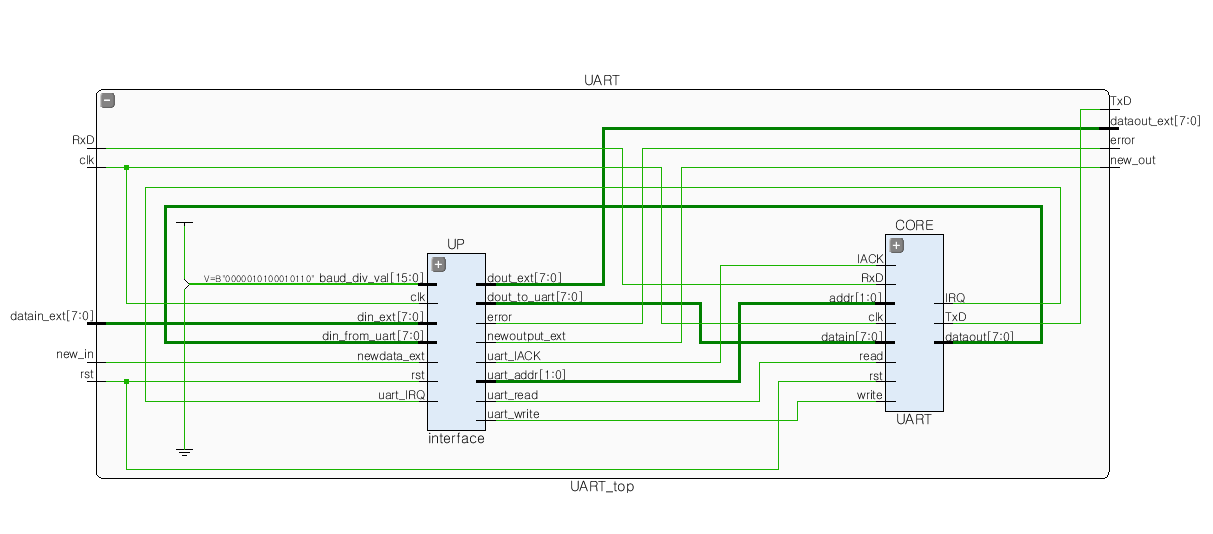
(그림 8.9)은 tera term 창에 ‘a’와 ‘A’를 차례로 입력한 것이다.



(그림 8.10) 7 Segments

(그림 8.10)는 ‘A’ 를 수신해 7 Segments에 표시한 것이다.

* **모듈 설명 (source codes)**
* **UART\_top**



(그림 8.11) UART\_top

(그림 8.11)은 UART\_top 모듈의 구조이다. UART\_top 모듈은 interface(UP)모듈(이하 UP)과 UART(CORE, 이하 UART)모듈(이하 UART)로 구성된다. UP는 UART와 UART를 사용하는 Master 모듈 간 interface를 제공하는 모듈이며, UART는 UART의 코어이다. 먼저 UART를 살펴보도록 한다.



(그림 8.12) UART 계층구조

(그림 8.12)에서 UART는 CLK 모듈과 RX, TX 모듈로 구성된 것을 확인할 수 있다. CLK 모듈은 UART를 baud rate에 데이터를 전송/수신 할 수 있도록 하는 일종의 클락 신호를 생성하는 모듈이다. RX는 신호를 받는 모듈이며, TX는 전송하는 모듈이다. UART의 코드를 살펴보도록 한다.

// addr = "00" => status (Read), control (Write)

// addr = "01" => RBR (Read), TBR (Write)

// addr = "10" => Clk Division Low (Write)

// addr = "11" => Clk Division High (Write)

`define TBRE status\_flags[7] // status: TBR Empty

`define RBRF status\_flags[6] // RBR Full

`define FE status\_flags[5] // Frame Error

`define OE status\_flags[4] // Overrun Error

`define PE status\_flags[3] // Parity Error

`define TC status\_flags[2] // transmit complete

`define status\_unused status\_flags[1:0] // unused status bits

`define TIE control\_flags[7] // control: interrup enable

`define RIE control\_flags[6] // control: interrup enable

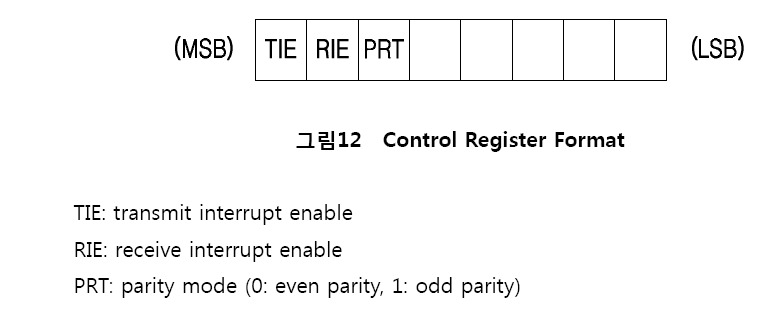
`define PRT control\_flags[5] // control: parity control (0:even, 1: odd)

위의 7개의 define문은 status register를 사용하기 위한 것이다. 그리고 아래 3개의 define문은 control register를 사용하기 위한 정의이다.



(그림 8.13) Status Register

(그림 8.13)은 UART의 다양한 상태를 register의 각 비트를 이용해 나타내는 Status Register의 구조를 보여준다. TE는 transmit buffer empty의 약어이며 transmit 버퍼에 데이터가 없는 경우를 의미한다. RE는 receiver buffer full의 약어이며, receiver 버퍼에 빈 공간이 없는 경우를 의미한다. FE는 frame error의 약어로, stop bit가 없는 경우를 의미한다. OE는 overrun error의 약어로, receiver 버퍼가 가득 찬 상태에서 새로운 데이터를 받은 경우를 의미한다. PE는 parity error의 약어로, parity bit가 맞지 않는 경우를 의미한다. TC는 transmit completed의 약어로, TSR을 통해서 전송이 끝난 경우를 의미한다.



(그림 8.14) Control Register

(그림 8.14)의 Control Register는 UART의 동작을 configuration하기 위한 것이다. TIE는 transmit interrupt enable의 약어이며, RIE는 receive interrupt enable의 약어이다. TIE는 앞서 status register에서 살펴본 TE flag에 대한 enable을 의미하며, TIE가 Low인 경우, TE에 의한 interrupt가 동작하지 않는다. 마찬가지로 RIE는 RE flag에 대한 enable이다. PRT는 parity mode의 약어로, Low이면 even parity를, High이면 odd parity를 사용한다.

module UART(

input clk,

input rst,

input read,

input write,

input [1:0] addr,

input RxD,

input IACK,

input [7:0] datain,

output reg [7:0] dataout,

output TxD,

output reg IRQ

);

UART 모듈의 입/출력 신호를 확인할 수 있다.

UART모듈 내부에서 사용하는 신호에 대한 선언이다

wire RxEn, TxEn;

wire setRBRF, setOE, setFE, setTBRE, setTC, setPE;

reg write\_TBR, read\_RBR, ld\_control,

read\_status, scale\_high\_ld, scale\_low\_ld;

wire [7:0] RBR;

localparam ADDR\_CTRL\_STAT = 2'b00, ADDR\_DATA\_BUF = 2'b01, ADDR\_SCALE\_LSB = 2'b10, ADDR\_SCALE\_MSB = 2'b11;

reg [7:0] status\_flags, control\_flags;

wire transmit\_int, receive\_int;

reg RxD\_d, RxD\_dd;

clkgen CLK (

.clk(clk),

.rst(rst),

.scale\_high\_ld(scale\_high\_ld),

.scale\_low\_ld(scale\_low\_ld),

.scale\_val(datain),

.RxEn(RxEn),

.TxEn(TxEn)

);reg RxD\_d, RxD\_dd;

clkgen 모듈을 instantiation한다. clkgen 모듈은 interface 모듈에서 전달한 정보로부터 baud rate에 맞는 클럭 신호를 생성한다. UART를 초기화할 때, UART의 baud rate에 대한 정보를 상위 8비트, 하위 8비트로 나눠서 clkgen 모듈에 전달하는 과정이 존재한다. UART를 초기화 하는 과정을 살펴본다.



(그림 8.15) UART Usage Procedure

(그림 8.15)에서 앞의 3단계는 UART를 사용하기 전에 초기화하는 단계이며, 초기화 단계에서 baud rate를 맞추기 위해 상위 8비트와 하위 8비트를 clkgen모듈에 저장하는 단계가 포함되는 것을 확인할 수 있다.

receiver RX (

.clk(clk),

.rst(rst),

.RxEn(RxEn),

.RxD(RxD\_dd),

.RBRF(`RBRF),

.PRT(`PRT),

.RBR(RBR),

.setRBRF(setRBRF),

.setOE(setOE),

.setFE(setFE),

.setPE(setPE)

);

transmit TX (

.clk(clk),

.rst(rst),

.TxEn(TxEn),

.TBRE(`TBRE),

.PRT(`PRT),

.ld\_tbr(write\_TBR),

.datain(datain),

.setTBRE(setTBRE),

.setTC(setTC),

.TxD(TxD)

);

Receiver 모듈과(이하 RX) Transmit 모듈을(이하 TX) instantiation하는 부분이다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

begin

RxD\_d <= 'b1;

RxD\_dd <= 'b1;

end

else if (RxEn == 'b1)

begin

RxD\_d <= RxD;

RxD\_dd <= RxD\_d;

end

RxD의 지연된 신호를 만든다. RxD는 데이터를 수신하는 port이다. 지연된 신호를 만드는 이유는, start bit을 detect할 때, RxD 신호와 지연된 RxD 신호를 비교해 RxD신호가 High to Low로 변하는 event를 detect해야하기 때문이다. 또한 비동기 신호를 시스템의 clk에 동기화할 수 있는 장점과 노이즈에 의한 영향을 줄이는 장점도 있다.

always @(addr, read, write)

begin

scale\_low\_ld = 'b0;

scale\_high\_ld = 'b0;

write\_TBR = 'b0;

ld\_control = 'b0;

read\_RBR = 'b0;

read\_status = 'b0;

if (write == 'b1)

case (addr)

ADDR\_CTRL\_STAT : ld\_control = 'b1;

ADDR\_DATA\_BUF : write\_TBR = 'b1;

ADDR\_SCALE\_LSB : scale\_low\_ld = 'b1;

ADDR\_SCALE\_MSB : scale\_high\_ld = 'b1;

default : ;

endcase

if (read == 'b1)

case (addr)

ADDR\_CTRL\_STAT : read\_status = 'b1;

ADDR\_DATA\_BUF : read\_RBR = 'b1;

default : ;

endcase

end

UART의 FSM이다. read와 write에 따라서, addr 신호를 바탕으로 UART를 control하는 신호가 생성된다. UART는 read, write, addr신호에 따라서 다음과 같은 동작을 한다.

<표 8.1> UART 제어신호

|  |  |  |
| --- | --- | --- |
| addr | WR = 1 (Write Operation) | RD = 1 (Read Operation) |
| 00 | Control Register | Status Register |
| 01 | Transmit Buffer | Receive Buffer |
| 10 | Clock Divider Value | - |
| 11 | Clock Divider Value | - |

always @(posedge clk, posedge rst)

if (rst == 'b1)

dataout <= 'bz;

else if (read\_RBR == 'b1)

dataout <= RBR;

else if (read\_status == 'b1)

dataout <= status\_flags;

출력신호 dataout에 대한 assignment를 나타낸다. Rst 신호가 High이면, High Z 값을 dataout에 assign하고, read\_RBR이 High이면, RBR에 저장된 값을 assign한다. Read\_status\_flags가 High이면, status\_flags를 assign한다. 그 밖의 경우에는, latch가 발생해 dataout신호가 바뀌지 않고 유지된다.

always @(posedge clk, posedge rst)

if (rst)

control\_flags <= 'b0;

else if (ld\_control == 'b1)

control\_flags <= datain;

초기화 과정에서, control\_flags를 update하는 block이다. reset을 할 때는 0으로 초기화되고, ld\_control신호가 High일 때는 datain을 통해 입력받은 값으로 update된다. 그 밖의 경우 latch가 발생해 값이 유지된다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

begin

`TBRE = 'b1; // TBRE

`RBRF = 'b0; // RBRF

`FE = 'b0; // FE

`OE = 'b0; // OE

`PE = 'b0; // PE

`TC = 'b0; // TC

`status\_unused = 2'b00; // status\_unused

end

else

begin

if (setTBRE == 'b1)

`TBRE = 'b1;

else if (write\_TBR == 'b1)

`TBRE = 'b0;

if (setRBRF == 'b1)

`RBRF = 'b1;

else if (read\_RBR == 'b1)

`RBRF = 'b0;

if (setFE == 'b1)

`FE = 'b1;

else if (read\_status == 'b1)

`FE = 'b0;

if (setOE == 'b1)

`OE = 'b1;

else if (read\_status == 'b1)

`OE = 'b0;

if (setPE == 'b1)

`PE = 'b1;

else if (read\_status == 'b1)

`PE = 'b0;

if (setTC == 'b1)

`TC = 'b1;

else if (read\_status == 'b1)

`TC = 'b0;

end

status register의 bit를 assign한다. reset시에는 TBRE로 정의한 bit를 제외하고 모두 Low이다.

interrupt신호에 대한 내용이다. Interrupt는 transmit\_int 와 receive\_int 두 종류이며, 인터럽트는 UART를 사용하는 Master모듈에 전해져 interrupt에 대한 처리를 요구한다. transmit\_int는 데이터를 전송할 때 버퍼가 비어 전송할 수 없는 경우 발생하며, receive\_int는 데이터를 수신할 때 버퍼가 가득 차 더 이상 데이터를 buffuring할 수 없는 경우 발생한다. 두 interrupt는 TIE와 RIE비트에 의해 각각 enable된다. IRQ신호는 두 인터럽트의 or-ing으로 assign하며, Master 모듈에 의해 IACK신호를 받으면(인터럽트 발생에 대한 응답) 다시 Low로 assign한다.

assign transmit\_int = `TIE & `TBRE; // TIE and TBRE

assign receive\_int = `RIE & `RBRF; // RIE and RBRF

always @(posedge clk, posedge rst)

if (rst)

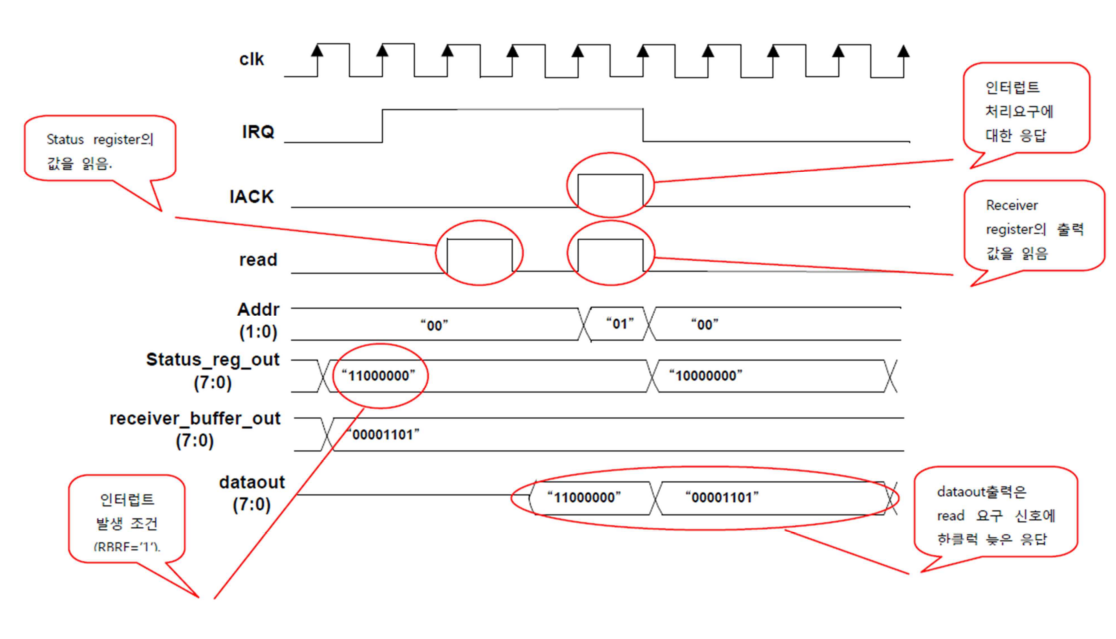
IRQ <= 'b0;

else if ((IACK == 'b1) && (IRQ == 'b1))

IRQ <= 'b0;

else if ((transmit\_int == 'b1) || (receive\_int == 'b1))

IRQ <= 'b1;



(그림 8.16) Interrupt 발생/처리의 예

다음으로, UART 하위 모듈인 clkgen 모듈의 코드를 살펴본다. clkgen 모듈은 UART의 baud rate generator라고도 부른다.

module clkgen(

input clk,

input rst,

input scale\_high\_ld,

input scale\_low\_ld,

input [7:0] scale\_val,

output RxEn,

output TxEn

);

reg [15:0] scale\_cnt\_rx;

reg [2:0] scale\_cnt\_tx;

reg [15:0] scale\_terminal\_val;

wire rx\_fin, tx\_fin;

clkgen 모듈의 입/출력 신호와 모듈 내부에서 사용하는 신호를 정의한다. scale\_high\_ld가 high일 때, scale\_value신호를 통해 clock 분주에 대한 denominator의 high 8 bits가 입력되며, scale\_low\_ld가 high일 때, scale\_value신호를 통해 low 8 bits가 입력된다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

scale\_terminal\_val <= 16'b0000000000000001;

else if (scale\_high\_ld == 'b1)

scale\_terminal\_val[15:8] <= scale\_val;

// scale\_terminal\_val[15:8] = scale\_terminal\_high

else if (scale\_low\_ld == 'b1)

scale\_terminal\_val[7:0] <= scale\_val;

// scale\_terminal\_val[7:0] = scale\_terminal\_low

clock 분주에 대한 denominator가 저장되는 scale\_terminal\_val에 상위, 하위 8bits씩 update하는 코드이다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

scale\_cnt\_rx <= 'b0;

else

begin

scale\_cnt\_rx <= scale\_cnt\_rx + 1;

if (rx\_fin == 'b1)

scale\_cnt\_rx <= 'b0;

end

assign rx\_fin = (scale\_cnt\_rx == scale\_terminal\_val)? 'b1 : 'b0;

scale\_cnt\_rx는 counter의 출력과 같이 동작한다. rx\_fin은 scale\_cnt\_rx가 scale\_terminal\_val에 저장된 값과 같아질 때만 high가 되는 pulse신호이다. Rx\_fin이 high일 때, scale\_cnt\_rx\_는 0으로 초기화된다. 따라서 scale\_cnt\_rx는 scale\_terminal\_val에 저장된 값을 최대값으로 가지는 counter의 출력과 같이 동작한다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

scale\_cnt\_tx <= 'b0;

else if (rx\_fin == 'b1)

begin

scale\_cnt\_tx <= scale\_cnt\_tx + 1;

if (tx\_fin == 'b1)

scale\_cnt\_tx <= 'b0;

end

assign tx\_fin = (scale\_cnt\_tx == 3'b111)? 'b1 : 'b0;

scale\_cnt\_tx는 rx\_fin을 동기신호로하는 counter의 출력과 같이 동작한다. tx\_fin은 scale\_cnt\_tx가 3’b111과 같아질 때만 high가 되는 pulse신호이다. tx\_fin이 high일 때, scale\_cnt\_rx\_는 0으로 초기화된다. 따라서 scale\_cnt\_rx는 scale\_terminal\_val에 저장된 값을 최대값으로 가지는 counter의 출력과 같이 동작하며, rx\_fin보다 8배 주기가 긴 pulse신호가 된다.

assign TxEn = tx\_fin & rx\_fin;

assign RxEn = rx\_fin;

TxEn과 RxEn은 UART의 Tx와 Rx의 동기신호로 사용된다.

clkgen모듈의 동작을 시뮬레이션을 통해 확인할 수 있도록 Test bench를 작성해 각 신호들을 확인해 보는 것을 권장한다.

다음으로, receiver 모듈의 코드를 살펴본다.

module receiver(

input clk,

input rst,

input RxEn,

input RxD,

input RBRF,

input PRT,

output [7:0] RBR,

output reg setRBRF,

output reg setOE,

output reg setFE,

output reg setPE

);

reg [8:0] RBReg; // recieve buffer

reg [8:0] RSR; // receive shift register

reg [3:0] BitCnt;

reg [2:0] SampleCnt;

reg [2:0] curr\_st, nx\_st;

localparam IDLE = 3'b000, START\_DETECT = 3'b001, SKIP\_INT = 3'b010, GET\_BIT = 3'b011, STOP\_BIT = 3'b100;

reg clr\_SampleCnt, inc\_SampleCnt, clr\_BitCnt, inc\_BitCnt;

wire Sample\_6, Sample\_3, BitCnt\_9;

wire detect\_falling\_edge;

reg shift\_RSR, ld\_RBR, RxD\_delayed;

wire parity;

먼저, receiver의 입/출력 신호를 선언하고 내부 신호를 선언한다. 입력인 RxEn은 Rx의 동기화 신호로, clkgen 모듈로부터 만들어진 신호이다. RxD는 RxEn을 지연시킨 신호이다. curr\_st와 nx\_st는 FSM의 각각 current state와 next state를 나타내는 신호이다. RSR는 1비트씩 전송받은 데이터를 저장하는 shift register이다. State는 총 5단계이며, IDLE, START\_DETECT, SKIP\_INT, GET\_BIT, STOP\_BIT로 24번 라인에서 localparameter로 그 이름을 정의하고있다.

assign RBR = RBReg[7:0]; // RBReg[8] is parity bit

assign parity = RSR[8] ^ RSR[7] ^ RSR[6] ^ RSR[5] ^ RSR[4] ^ RSR[3] ^ RSR[2] ^ RSR[1] ^ RSR[0];

parity를 구한다. 1프레임의 데이터(8bit+parity bit)를 모두 bit단위로 XOR시킨다. 즉, 8비트의 데이터와 parity bit에 High가 홀수만큼 있는지, 짝수만큼 있는지 확인하는 것이다. 총 9비트 중 High가 홀수만큼 있으면 parity는 High가 되고, 그 밖의 경우 Low가 된다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

RxD\_delayed <= 'b0;

else if (RxEn == 'b1)

RxD\_delayed <= RxD;

assign detect\_falling\_edge = (!RxD) & RxD\_delayed;

// start bit check

RxD\_delayed는 RxD신호를 RxEn이 High일 때 1 clk만큼 지연시킨 것이다. RxD를 negation시킨 것과 RxD\_delayed신호를 and함으로서, detect\_falling\_edge신호는 RxD가 High에서 Low로 떨어질 때, 1 clk동안 High가 유지되는 pulse 신호가 된다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

curr\_st <= IDLE;

else

curr\_st <= nx\_st;

receiver 모듈의 핵심을 이루는 FSM의 sequential machine part이다. Default 혹은 reset시에는 IDLE state를 유지한다.

always @\*

begin

clr\_SampleCnt = 'b0;

inc\_SampleCnt = 'b0;

clr\_BitCnt = 'b0;

inc\_BitCnt = 'b0;

ld\_RBR = 'b0;

shift\_RSR = 'b0;

setFE = 'b0;

setOE = 'b0;

setRBRF = 'b0;

setPE = 'b0;

receiver 모듈의 핵심을 이루는 FSM의 combinational machine part이다. Default 값을 나타낸다.

case (curr\_st)

IDLE :

begin

nx\_st = IDLE;

if (detect\_falling\_edge == 'b1)

begin

nx\_st = START\_DETECT;

clr\_SampleCnt = 'b1;

clr\_BitCnt = 'b1;

end

end

detect\_falling\_edge가 High가 되면, next state는 START\_DETECT가 된다. 또한, Clr\_SampleCnt와 Clkr\_BitCnt가 High가 된다. High가 되지 않으면, latch가 발생해 값이 state와 Clr\_SampleCnt와 Clkr\_BitCnt가 default값인 Low로 유지된다.

start bit을 detect했지만, 잡음에 의한 것인지를 판단하는 state이다. Start bit가 발생하고 RxEn의 첫 Pulse가 발생했을 때, RxD가 Low로 유지되고 있다면 잡음의 영향이 아니므로 next state는 SKIP\_INT가 된다.

SKIP\_INT :

begin

nx\_st = SKIP\_INT;

if (RxEn == 'b1)

begin

inc\_SampleCnt = 'b1;

if (Sample\_3 == 'b1)

nx\_st = GET\_BIT;

end

end

START\_DETECT :

begin

nx\_st = START\_DETECT

if (RxEn == 'b1)

begin

inc\_SampleCnt = 'b1;

if (RxD == 'b1) // NOISE

nx\_st = IDLE;

else if (Sample\_3 == 'b1)

nx\_st = SKIP\_INT;

end

end

RxEn이 High일 때 마다 inc\_SampleCnt가 High가 되어 SampleCnt신호가 1씩 증가하게 되며, SampleCnt가 3과 같아지면 Sample\_3신호가 High가 된다. 이에 관한 코드는 아래 첨부한다. Sample\_3가 High가 되면 next state는 GET\_BIT가 된다. SampleCnt가 3이 될 때까지 기다리는 이유는, UART가 비동기 신호이기 때문에, 강제적인 delay를 만들어 sampling하는 타이밍이 실제 signal의 중간쯤 위치하도록 하기 위함이다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

begin

BitCnt <= 'b0;

SampleCnt <= 'b0;

end

else

begin

if (clr\_BitCnt == 'b1)

BitCnt <= 'b0;

else if (inc\_BitCnt == 'b1)

BitCnt <= BitCnt + 1;

if (clr\_SampleCnt == 'b1)

SampleCnt <= 'b0;

else if (inc\_SampleCnt == 'b1)

SampleCnt <= SampleCnt + 1;

end

assign Sample\_3 = (SampleCnt == 'd3)? 'b1 : 'b0;

assign Sample\_6 = (SampleCnt == 'd6)? 'b1 : 'b0;

assign BitCnt\_9 = (BitCnt == 'd9)? 'b1 : 'b0;

clr\_BitCnt가 High이면 BitCnt는 0으로 초기화되며, inc\_BitCnt가 High이면 1씩 증가한다. Clr\_SampleCnt가 High이면 SampleCnt가 0으로 초기화되며, inc\_SampleCnt가 High이면 1씩 증가한다. Sample\_3와 Sample\_6, 그리고 BitCnt\_9은 모두 SampleCnt 혹은 Bitcnt가 특정 값을 가질 때마다 High가 되는 Pulse 신호이다.

GET\_BIT :

begin

nx\_st = GET\_BIT;

if (RxEn == 'b1)

begin

inc\_BitCnt = 'b1;

inc\_SampleCnt = 'b1;

if (BitCnt\_9 == 'b1)

begin

nx\_st = STOP\_BIT;

clr\_BitCnt = 'b1;

end

else

begin

shift\_RSR = 'b1;

nx\_st = SKIP\_INT;

end

end

end

RxEn이 High이 될 때마다, inc\_BitCnt를 High로 함으로써 BitCnt를 1씩 증가시킨다. 마찬가지로 , inc\_SampleCnt를 High로 함으로써 SampleCnt를 1씩 증가시킨다. shift\_RSR이 High이면, RSR의 MSB위치에서부터 RxD의 값을 채워넣는다. 관련 코드는 아래 첨부한다. BitCnt\_9가 High가 되는 것은, RxD의 값을 9번 sampling해서 저장했음을 의미하며, 이 때 next state는 STOP\_BIT가 되며 clk\_BitCnt가 High가 되어 BitCnt를 0으로 초기화한다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

begin

RSR <= 'b0;

RBReg <= 'b0;

end

else

begin

if (ld\_RBR == 'b1)

RBReg <= RSR;

if (shift\_RSR == 'b1)

RSR <= {RxD, RSR[8:1]};

end

shift\_RSR 신호와 ld\_RBR신호에 따라 RSR과 RBReg가 어떻게 assign되는지 확인할 수 있다

마지막 state인 STOP\_BIT state이다. RxEn이 High일 때, setFE, setOE, setPE신호를 assign한다. 먼저, RxD가 Low인 경우, 즉 stop bit가 감지되지 않는 경우 frame error인 FE를 set하며, read buffer가 High인 경우 overrun orror인 OE를 set한다. 또한, parity가 PRT와 다르면 parity error인 PE를 set한다. 그 밖의 경우, ld\_RBR을 set해서 수신한 data를 출력하도록 한다. 또한, RBRF를 set해서 수신할 데이터가 있음을 master module에 알린다.

STOP\_BIT :

begin

nx\_st = STOP\_BIT;

if (RxEn == 'b1)

begin

nx\_st = IDLE;

if (RxD == 'b0)

setFE = 'b1;

else if (RBRF == 'b1)

setOE = 'b1;

else if (parity != PRT)

setPE = 'b1;

else

begin

ld\_RBR = 'b1;

setRBRF = 'b1;

end

end

end

default : nx\_st = IDLE;

*receiver의 동작을 완전히 이해하면, 이를 바탕으로 transmitter 모듈의 코드를 이해할 수 있다.*

다음으로, Interface 모듈의 코드를 살펴본다. Interface 모듈은 UART를 사용하는 Master 모듈과 UART 모듈 사이에서 신호와 데이터의 처리를 담당한다.

`define TE status[7]

`define RF status[6]

`define FE status[5]

`define OVE status[4]

`define PE status[3]

state register의 각 비트에 대한 정의이다.

module interface(

input clk,

input rst,

input [15:0] baud\_div\_val,

/\* interface with external IO \*/

input newdata\_ext,

input [7:0] din\_ext, // extin->interface

output reg [7:0] dout\_ext, // interface -> extout

output reg newoutput\_ext,

/\* interface with UART \*/

output reg uart\_read,

output reg uart\_write,

output reg uart\_IACK,

input uart\_IRQ,

output reg [1:0] uart\_addr,

input [7:0] din\_from\_uart, // UART->interface

output [7:0] dout\_to\_uart, // interface->UART

output reg error

);

baud\_div\_val 은 master module로부터 입력 받는 baud rate에 대한 분주 dinominator이다. newdata\_ext 은 master module로부터 데이터를 입력 받을 때 high가 되는 일종의 write enable신호이다. din\_ext 는 master module로부터 데이터를 입력 받는 신호이다. newoutput\_ext 은 master module로 data를 출력할 때 high가 되는 일종의 read enable신호이다. dout\_ext 는 출력 데이터를 싣는 신호이다. uart\_read 부터 error 신호는 interface모듈과 UART 모듈 간의 제어신호이다. uart\_IRQ 와 uart\_IACK 은 UART로부터 interrupt 요청이 들어오고, 이에 대해 응답하는 신호로 pair이다. uart\_read 와 uart\_write는 UART신호에 대해 데이터를 읽거나 쓰겠다는 명령이다. uart\_addr는 <표 8.1>에 소개한 addr 신호이다. din\_from\_uart와 dout\_to\_uart 는UART와 interface간 주고 받는 데이터를 싣는 신호이다. error 은 master module에 error가 발생함을 알리는 신호이다.

reg [7:0] status; // status = { TE, RF, RE, OVE, PE, 3'b000 }

reg write\_data\_to\_uart, new\_input\_arrived, set\_error;

reg read\_data, read\_status, read\_data\_delayed\_command,

read\_status\_delayed\_command;

reg [7:0] ctrl\_data, data\_buf;

reg [2:0] curr\_st, nx\_st;

localparam INIT = 3'b000, INIT\_2 = 3'b001, INIT\_3 = 3'b010,

READY = 3'b011, CHECK\_STATUS = 3'b100,

CHECK\_2 = 3'b101, CHECK\_3 = 3'b110, CHECK\_4 = 3'b111;

모듈 내에서 사용하는 신호를 선언했다. localparam 정의를 통해 interface의 FSM이 총 8개 state개로 구성됨을 알 수 있다. INIT, INIT\_2, INIT\_3, INIT\_3은 UART를 초기화하는 단계의 state를 의미하며, READY는 일종의 idle state이며, CHECK\_STATUS, CHECK\_2, CHECK\_3, CHECK\_4 state는 상황에 맞게 UART를 제어하는 state이다.

Sequential machine part이다.

always @(posedge clk, posedge rst)

if (rst == 'b1)

curr\_st <= INIT;

else

curr\_st <= nx\_st;

always @(curr\_st,uart\_IRQ, new\_input\_arrived, status)

begin

uart\_write = 'b0;

uart\_read = 'b0;

uart\_IACK = 'b0;

uart\_addr = 2'b00;

write\_data\_to\_uart = 'b0;

read\_data = 'b0;

read\_status = 'b0;

ctrl\_data = 'b0;

set\_error = 'b0;

combinational machine part이다. 각 신호의 default 값을 나타낸다.

case (curr\_st)

INIT :

begin

uart\_write = 'b1;

uart\_addr = 2'b00;

ctrl\_data = 8'b01100000;

nx\_st = INIT\_2;

end

INIT\_2 :

begin

uart\_write = 'b1;

uart\_addr = 2'b10;

ctrl\_data = baud\_div\_val[7:0]; // load scale LSB

nx\_st = INIT\_3;

end

INIT\_3 :

begin

uart\_write = 'b1;

uart\_addr = 2'b11;

ctrl\_data = baud\_div\_val[15:8]; // load scale MSB

nx\_st = READY;

end

INIT state에서는 ctrl\_data를 UART의 control register에 쓴다. 63번 라인에서, control register에 쓰는 값은 receive interrupt를 활성화하고, odd parity를 사용한다는 것이다.

INIT\_2 state는 baud\_div\_val의 하위 8비트를 UART의 clkgen모듈에 쓰는 동작을 한다.

INIT\_3 state는 baud\_div\_val의 상위 8비트를 UART의 clkgen모듈에 쓰는 동작을 한다.

READY state에서는 uart\_IRQ 신호를 확인해 High이면 CHECK\_STATUS로 넘어간다. IRQ가 발생하지 않은 경우, master module로부터 새로운 데이터가 들어오면 UART에(transmit buffer에) 데이터를 쓴다.

CHECK\_STATUS :

begin

uart\_read = 'b1;

uart\_addr = 2'b00;

read\_status = 'b1;

nx\_st = CHECK\_2;

end

CHECK\_2 : nx\_st = CHECK\_3;

CHECK\_3 :

begin

nx\_st = CHECK\_4;

uart\_IACK = 'b1;

if (`RF == 'b1) // data ready in RBR

begin

uart\_read = 'b1;

uart\_addr = 2'b01;

read\_data = 'b1;

end

else if ((`TE == 'b1) && (new\_input\_arrived == 'b1))

begin

write\_data\_to\_uart = 'b1;

uart\_write = 'b1;

uart\_addr = 2'b01;

end

end

CHECK\_4 :

if (uart\_IRQ == 'b0)

begin

uart\_IACK = 'b0;

nx\_st = READY;

end

else

nx\_st = CHECK\_4;

default : nx\_st = READY;

READY :

begin

nx\_st = READY;

if (uart\_IRQ == 'b1)

nx\_st = CHECK\_STATUS;

else if (new\_input\_arrived == 'b1)

begin

write\_data\_to\_uart = 'b1;

uart\_write = 'b1;

uart\_addr = 2'b01;

end

end

CHECK\_STATUS에서는 status register의 값을 읽어 들인다. CHECK\_2에서는 1 클락 주기만큼 의도적인 delay 기능을 한다. 이는 UART로부터의 출력이 read 요구 신호에 1 클락만큼 늦게 응답하기 때문이다. CHECK\_3에서는 UART의 IRQ에 대한 uart\_IACK신호를 set하고, RF와 TE flag에 대한 처리를 한다. RF가 High인 경우, UART로부터 data를 읽는다. TE가 High이며, master module로부터 새로운 데이터가 들어온 경우, UART로 data를 쓴다. 다음, CHECK\_4 state로 넘어간다. CHECK\_4에서는 UART모듈이 uart\_IACK 신호에 반응해 uart\_IRQ 신호를 Low로 떨어뜨리는 것을 감지해 uart\_IACK신호를 Low로 떨어뜨리고, 다시 READY상태로 넘어간다.

interface UP (

.clk(clk),

.rst(rst),

.baud\_div\_val(16'b0000\_0101\_0001\_0110), //100MHZ / 9600 / 8

.newdata\_ext(new\_in),

.din\_ext(datain\_ext),

.dout\_ext(dataout\_ext),

.newoutput\_ext(new\_out),

.uart\_read(read),

.uart\_write(write),

.uart\_IACK(IACK),

.uart\_IRQ(IRQ),

.uart\_addr(addr),

.din\_from\_uart(dataout),

.dout\_to\_uart(datain),

.error(error)

);

UART\_top 모듈에서 interface 모듈을 instantiation하는 부분이다. 이 부분에서 baud rate를 정하는 부분을 확인할 수 있다. 현재 baud rate는 9600으로 정했다. baud rate 9600으로 정하기 위해서는 시스템 클럭을 분주할 factor가 필요하다. 이 factor에 해당하는 것이 baud\_div\_val 신호에 입력하는 값이다. 이 값은 시스템 클럭의 주파수인 100MHz를 원하는 baud rate 9600\*8으로 나눈 값이다. 나누는 값에 8이 곱해진 이유는, asynchronous 통신이기 때문이다. UART는 asynchronous 통신 프로토콜로, 송신자와 수신자가 동기화되지 않기 때문에 타이밍을 맞추기 위해서는 기준보다 더 빠른 신호가 필요하다.

* **과제**
* 위 실습 내용에서, 컴퓨터로부터 ENTER키( 0x0D )를 입력 받았을 때는 line feed와 carriage return을 출력하시오. (0x0A + 0x0D 혹은 0x0D + 0x0A)
* 그 외에는 입력받은 케릭터를 3번 연속으로 출력하시오.