VHDL 로 구현하는 비동기 직렬 통신 칩 UART

국 일 호
E-Mail : goodkook@csvlsi.kyunghee.ac.kr
Homepage : http://www.csvlsi.kyunghee.ac.kr

지난 몇 달간 본지의 연재를 통하여 어느 정도 VHDL 에 대한 기본적인 이해가 생겼을 것으로 본다. 컴퓨터를 이용하여 작업하는 모든 것이 그렇듯이 실제 예제에 의한 연습을 거치지 않는 한 죽은 지식에 불과하다. 머리 속에 대기중인 VHDL 을 살려내는 가장 좋은 방법중의 하나는 잘 짜여진, 그리고 어느 정도의 복잡성을 갖는 디자인 소스를 분석하고 설계하는 과정을 완전히 이해하는 것도 한 방법이라 하겠다. 이번에는 그 동안 익혀온 VHDL 을 토대로 비동기 직렬 통신 주변 장치인 UART(Universal Asynchronous Receiver-Transmitter) 설계를 분석해 보기로 한다. UART 를 분석해 봄으로서 양방향성 공통 데이터 버스로의 인터페이스 회로 설계와 시스템과 다른 클럭으로 작동하는 비동기 회로의 설계 등을 살펴 볼 수 있을 것이다. 이 글에서 다룰 내용은 UART 의 설계와 테스트 벤치의 작성, 합성, 그리고 기능 및 타이밍 시뮬레이션 등 VHDL 을 이용한 설계의 전과정에 대한 것이다. VHDL 을 이용하여 뭔가 쓸만한 것을 설계해보고자 한다면 이 글이 도움이 되길 바란다.

1. UART 의 구성

이번에 구현해 보고자 하는 UART 는 프레임 설정 및 통신속도 설정 등의 기능이 제외된 것이지만 그 외 비동기 통신 기능에 있어서는 실제로 사용하기에 손색이 없는 것이다. 직렬 송수신 데이터 비트의 프레임 구성은 시작 비트, 8 비트 데이터, 패리티 비트, 스톱비트로 고정되어 있고 패리티 에러(parity error) 및 프레이밍 에러(framing error) 검출 기능을 내장한다. 직렬 데이터 프레임의 구성은 그림 2 와 같다. 직렬 통신 속도는 UART 의 클럭을 16 분주하여 정한다.

1-1. UART 의 구성 및 입출력 핀 기능

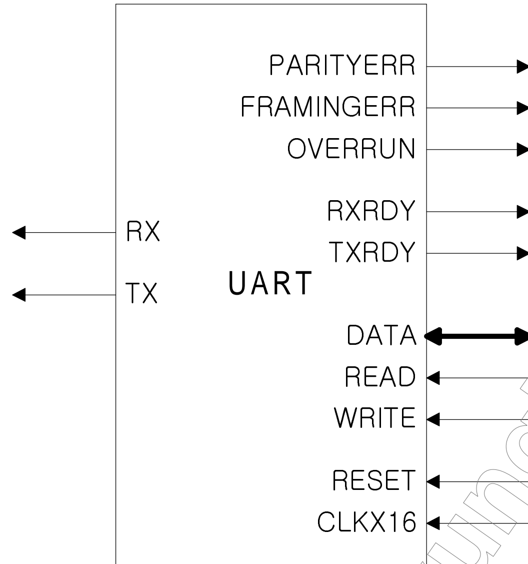
UART 의 외부 핀 구성(pin-out) 및 ENTITY 선언은 그림 1 과 같다. UART 각 핀의 기능은 표 1 에 나타내었다. 8 비트 양방향성 데이터 버스와 데이터 읽기 및 쓰기를 위한 제어 핀, 그리고 통신 에러를 나타내는 핀들로 구성된다. UART 의 내부 구성은 8 비트 데이터 입출력 인터페이스와 직렬 송신(TX unit) 및 직렬 수신부(RX unit)로 구성되며 그림 3 과 같다.

표. 1 UART 의 입출력 핀

핀 이름	비트폭	방향	기 능
RX	1	입력	직렬 수신
TX	1	출력	직렬 송신
RXRDY	1	출력	수신 완료
TXRDY	1	출력	송신 대기
READ	1	입력	병렬 데이터 읽기
WRITE	1	입력	병렬 데이터 쓰기

http://www.csvlsi.kyunghee.ac.kr

DATA	8	입출력	병렬 데이터
parityerr	1	출력	패리티 에러
framingerr	1	출력	프레이밍 에러
overrun	1	출력	오버런 에러
clkx16	1	입력	UART 동작 클럭




```
ENTITY uart IS
    PORT (clkx16 : IN    std_logic;        -- Input clock. 16x bit clock
            read : IN    std_logic;        -- Received data read strobe
           write : IN    std_logic;        -- Transmit data write strobe
              rx : IN    std_logic;        -- Receive  data line
           reset : IN    std_logic;        -- clear dependencies
              tx : OUT   std_logic;        -- Transmit data line
           rxrdy : OUT   std_logic;        -- Received data ready to be read
           txrdy : OUT   std_logic;        -- Transmitter ready for next byte
       parityerr : OUT   std_logic;        -- Receiver parity error
      framingerr : OUT   std_logic;        -- Receiver framing error
         overrun : OUT   std_logic;        -- Receiver overrun error
            data : INOUT std_logic_vector(0 TO 7)); -- Bidirectional data bus
END uart;
```

그림 1. UART의 입출력 핀 과 ENTITY

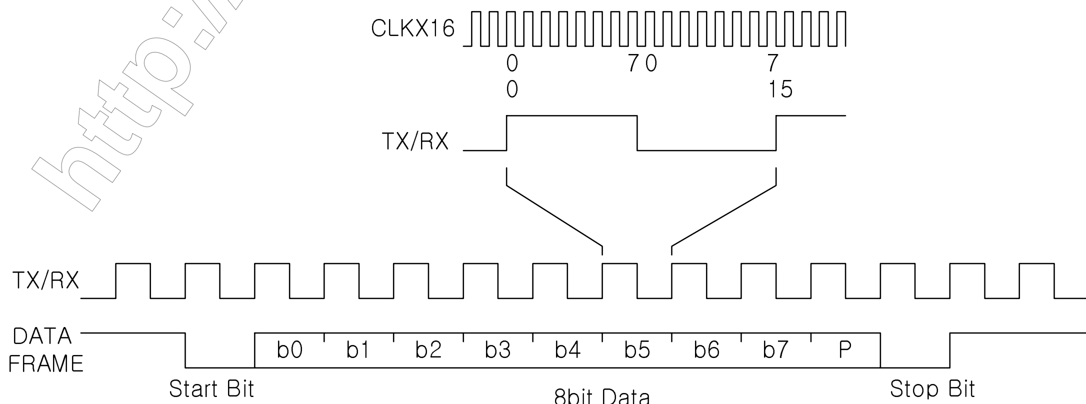


그림 2. 직렬 통신 데이터 프레임

http://www.cvlsi.kyunghee.ac.kr

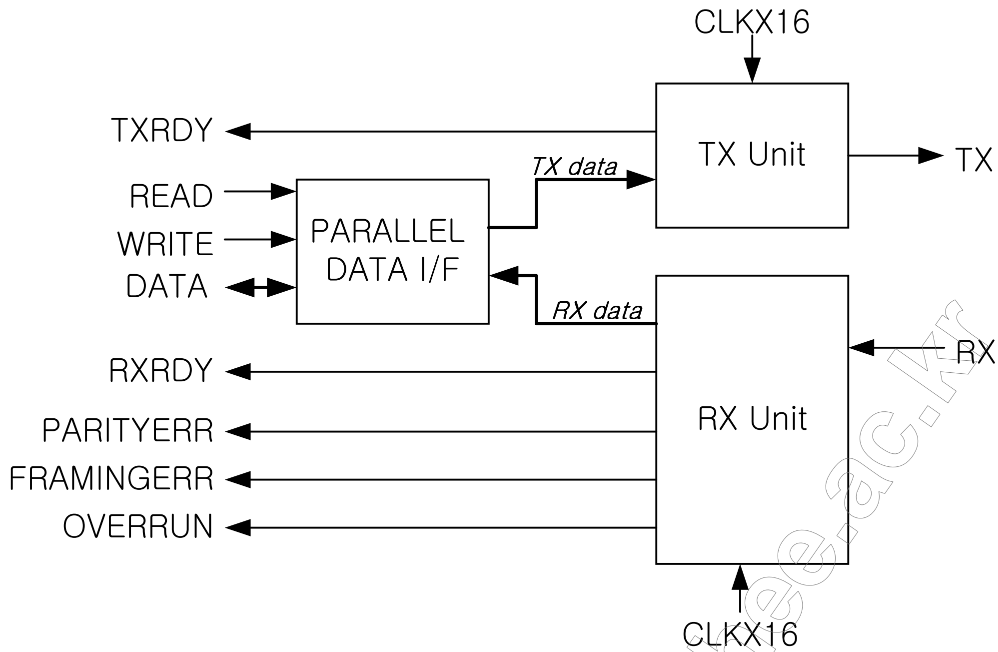


그림 3. UART의 내부 구성

1-2. 시스템 버스 인터페이스

주변 장치인 UART가 시스템 버스의 양방향성 공통 데이터 버스에 인터페이스 될 때 병렬 데이터 입출력 제어신호 관계는 그림 4와 같다. 송신부(TX Unit)로부터 송신용 쉬프트 버퍼에 더 이상 송신할 데이터가 없으면 송신준비신호(TXRDY)가 출력되며 이때 쓰기 스트로브(Write strobe)신호(WRITE)와 함께 8비트 데이터를 UART로 써넣을 수 있다. 수신은 수신부(RX unit)로부터 수신완료신호(RXRDY)가 출력되면 읽기 스트로브(read strobe)신호(READY)와 함께 8비트 데이터를 UART로부터 읽어 낸다. 이때 프레이밍(FRAMINGERR), 패리티(PARITYERR), 오버런(OVERRUN) 등의 에러 신호도 함께 출력 된다. 양방향 데이터 버스 "data"는 READ와 WRITE 제어선에 의하여 입출력 방향이 결정된다. UART는 시스템의 통제를 받는 일종의 주변 장치이다. 이런 경우 송수신 데이터의 읽기와 쓰기에 대한 인터페이스 타이밍과 양방향성 데이터 버스의 신호에 대하여 주의를 기울여야 한다. 그림 4에서 보듯이 시스템에서 송신할 데이터를 써넣기(WRITE) 는 UART로부터 송신 준비 신호(TXRDY)의 레벨 상태에서 가능하도록 설계되지만 수신 데이터의 읽기는 읽기 준비 신호(TXRDY)에 즉시 반응하도록 설계되어야 한다. 또한 외부 시스템과 공통 버스로 연결되는 양방향 버스를 다룰 때는 출력인 경우에 주의를 기울여야 한다. 외부시스템으로부터 읽기 요구가 없을 때는 반드시 하이 임피던스로 해 줌으로서 버스상의 충돌을 피하도록 한다. 결국 이와 같이 기술한 것은 그림 5에서 보는 것과 같이 양방향성 버스에 3-state 버퍼로 처리한 것이다. 양방향성 데이터 버스의 입출력 방향 제어를 기술하면 다음과 같이 된다

```
data <= rxhold WHEN read = '1' ELSE (OTHERS=>'Z');
txhold <= data WHEN write = '1' ELSE txhold;
```

http://www.cslsi.kyunghee.ac.kr

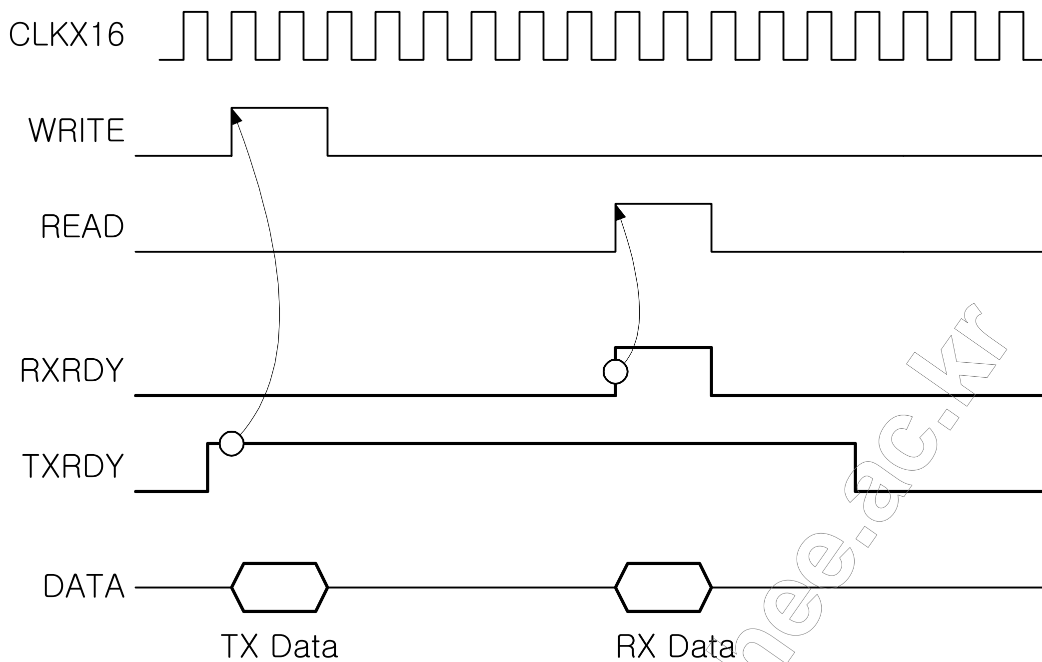


그림 4. 병렬 데이타 입출력 인터페이스

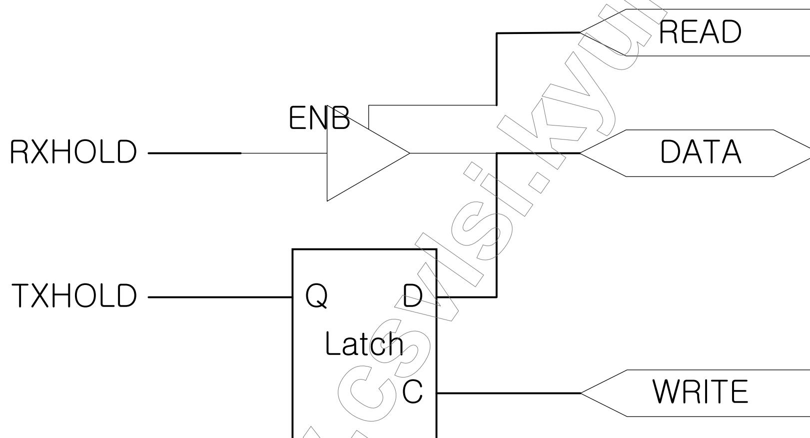


그림 5. 양방향 버스의 3-state 입출력

1-3. 송신부 (TX unit)

UART 의 설계에 있어서 송신부의 설계는 간단한 편이다. 본 설계에서는 통신 속도를 UART 의 마스터 클럭 clkx16 을 16 분주한 송신 클럭 발생기(make_txclk)와 패리티 발생기 회로 및 송신 종료 검출 회로를 갖춘 직렬 송신 쉬프트 레지스터(txshift), 외부 인터페이스를 위한 제어부(txio)로 구성된다.

- 송신 클럭 발생기 (make_txclk)

3 비트 카운터를 이용하여 송신 클럭 발생신호를 토글(toggle)시킴으로써 UART 의 마스터 클럭을 16 분주한다. 토글에 의하여 송신 클럭을 발생하기 위해서는 처음에 리셋일 때 카운터를 초기화 시켜야 한다. 송신 클럭 “txclk”은 직렬 송신 레지스터를 쉬프트 시키는데 이용된다.

http://www.cs.vlsi.kyunghee.ac.kr

```
PROCESS (reset, clkx16)
    VARIABLE cnt  : unsigned(2 DOWNTO 0);
BEGIN
    IF reset='1' THEN
        txclk <= '0';
        cnt := (OTHERS=>'0') ;
    ELSIF clkx16'event AND clkx16='1' THEN
        IF (cnt = "000") THEN
            txclk <= NOT txclk;
        END IF;
        cnt := cnt + "001";
    END IF;
END PROCESS;
```

- 직렬 송신 쉬프트 레지스터 (txshift)

외부로부터 WRITE 에 의하여 입력된 8 비트 병렬 데이터 "data"를 받아 래치된 "txhold"를 쉬프트 레지스터 "txreg"에 로드 한 후 송신용 클럭인 txclk 에 의하여 쉬프트 한다. 직렬 송신 쉬프트 레지스터의 VHDL 코드는 다음과 같다.

```
PROCESS (reset, txclk)
BEGIN
    IF reset='1' THEN
        txreg  <= (OTHERS=>'0') ;
        txtag1 <= '0' ;
        txtag2 <= '0' ;
        txparity <= '0' ;
        tx <= '0' ;
    ELSIF txclk'event AND txclk = '1' THEN
        IF (txdone AND txdatardy) = '1'  THEN
            -- 초기화 수행
            txreg    <= txhold; -- 새로 송신할 데이터
            txtag2   <= '1';  -- 송신 종료 검출용으로
            txtag1   <= '1';  -- 추가된 비트
            txparity <= '1';  -- 송신 패리티 검출용
            tx       <= '0';  -- 직렬 전송 포트
        ELSE
            -- 직렬 전송 레지스터의 쉬프트 동작
            txreg  <= txreg(1 TO 7) & txtag1;
            txtag1 <= txtag2;
            txtag2 <= '0';

            -- 송신중 출력 비트마다 패리티 계산
            txparity  <= txparity XOR txreg(0);
```

http://www.csvlsi.kyunghee.ac.kr

```
                -- Shift out data or parity or stop/idle bit
                IF txdone = '1' THEN
                    tx <= '1';          -- 전송 종료
                ELSIF paritycycle = '1' THEN
                    tx <= txparity;     -- 패리티 비트 출력
                ELSE
                    tx <= txreg(0); -- 직렬 전송 출력
                END IF;
            END IF ;
        END IF;
    END PROCESS;
```

송신 데이터 프레임에는 8 비트 데이터 이외에 시작 비트(start bit)와 패리티 비트(parity bit)가 첨가 된다. 따라서 총 송신할 비트 수는 8 비트 데이터 비트를 포함하여 10 비트이다. "txtag1" 과 "txtag2"는 10 비트 직렬 데이터 프레임을 만들고 송신 종료를 검지하기 위하여 사용된다. 직렬 송신 준비과정은 "txtag1"과 "txtag2"는 모두 '1'로 초기화 하고 쉬프트 레지스터 "txreg"는 래치된 8 비트 병렬 데이터 "txhold"를 로드한다. 피리티 비트는 odd parity 를 표시하므로 '1'로 초기화 하는 것이다.
송신할 8 비트 데이터 비트를 로드하기 위한 제어용 신호는 "txdone"과 "txdatardy" 이다. 송신 종료를 나타내는 "txdone"의 발생은 쉬프트 할 때 txtag2 에 '0'을 채우게 되므로 txreg 를 비롯하여 txtag1, txtag2 모두 '0'일 때 송신 종료이다. 송신 종료 검출은 다음과 같다.

```
txdone <= NOT (txtag2 OR txtag1 OR
               txreg(7) OR txreg(6) OR txreg(5) OR txreg(4) OR
               txreg(3) OR txreg(2) OR txreg(1) OR txreg(0));
```

송신 쉬프트 레지스터의 기술은 간단하다. 다만, 쉬프트 레지스터의 기술이 PROCESS 블록 내에 기술되어 있으므로 할당 구문의 순서에 주의 한다. 8 비트 송신데이터 쉬프트 레지스터는 다음과 같이 기술 할 수도 있으나 위의 예와 같이 간략하게 표현 하도록 하자.

```
    -- txreg  <= txreg(1 TO 7) & txtag1;
    txreg(0)  <= txreg(1);
    txreg(1)  <= txreg(2);
    txreg(2)  <= txreg(3);
    txreg(3)  <= txreg(4);
    txreg(4)  <= txreg(5);
    txreg(5)  <= txreg(6);
    txreg(6)  <= txreg(7);
    txreg(7)  <= txtag1;
```

송신할 8 비트 데이터가 쉬프트 레지스터(txreg)에 로드 된 후 직렬 전송이 시작 되면서 패리티 비트의 계산이 이루어 진다. 직렬 송신 쉬프트 레지스터와 패리티 계산을 기술하면 다음과 같다.

http://www.cs.kyungnam.ac.kr

```
    txparity <= txparity XOR txreg(0);
```

송신 종료 “txdone”는 8 비트 데이터와 시작비트, 패리티 비트 등을 모두 전송하였음을 나타낸다. 따라서 패리티 비트를 송신 비트 프레임에 삽입하려면 송신 종료 이전에 패리티 비트 삽입 싸이클을 감지하기 위한 회로 가 필요하다. 패리티 비트 삽입 사이클 감지 회로는 송신 데이터의 초기화 때 “txtag1” 비트가 쉬프트 되어 txreg(1)에 도달한 때이므로 다음과 같단 하게 기술할 수 있다.

```
paritycycle <= txreg(1) AND NOT ( txtag2 OR txtag1 OR
                                  txreg(7) OR txreg(6) OR txreg(5) OR
                                  txreg(4) OR txreg(3) OR txreg(2));
```

직렬 전송 출력되는 경우는 송신 종료, 패리티 삽입, 데이터 전송일 때 각각 다르다. 송신 종료 시 출력은 항상 ‘1’ 이어야 하며, 패리티 검출 싸이클 때는 그때까지의 패리티 비트 레지스터 값을 출력하고, 그 외 데이터 전송일 경우에는 전송 데이터의 직렬 출력이다. 그림 6 은 직렬 송신 쉬프트 레지스터와 패리티 비트 생성 기능의 구조를 나타내었다. txtag1 과 txtag2 에 있던 비트 값 ‘1’이 txreg (0)과 txreg(1)에 도달하면 패리티 계산은 종료하고 그 다음 송신될 비트에 패리티 값을 MUX 를 통하여 직렬 출력하게 되는 것이다.

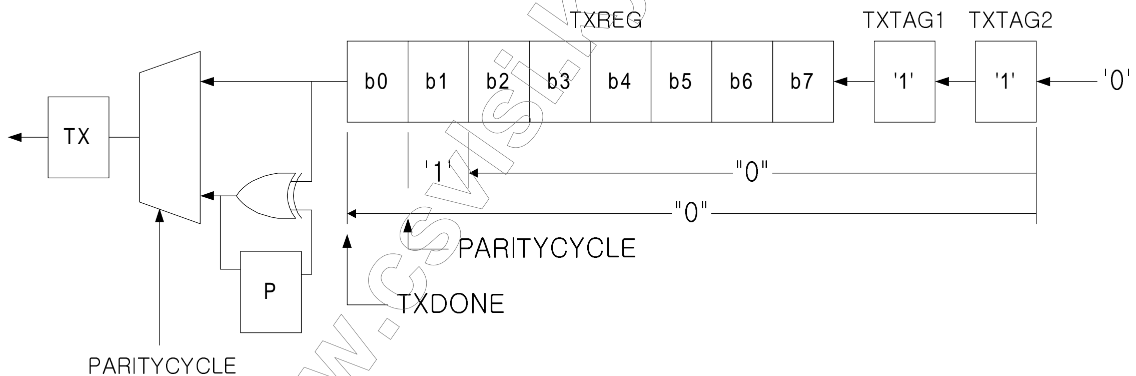


그림 6. 직렬 송신 쉬프트 레지스터와 패리티 비트 생성 구조

- 시스템 인터페이스(txio)

직렬 전송 쉬프트 레지스터(txreg)에 전송할 데이터를 초기화 시킬 때 사용했던 제어 신호인 “txdatardy”는 외부로부터 입력되는 8 비트 송신할 데이터가 준비되었음을 나타낸다. 외부 시스템으로부터 송신 데이터의 준비는 WRITE 신호와 함께 이루어진다. 이때 외부 시스템과의 인터페이스 이므로 UART 의 송신 클럭(txclk)과는 무관하게 이루어 지며 시스템 클럭인 clkx16 에 의하여 시스템 인터페이스 회로가 작동하게 된다. WRITE=’1’인 동안 시스템 버스로부터 래치 되는 송신 데이터 “txhold”는 WRITE=’0’이 된 후 비로서 직렬 송신 쉬프트 버퍼에 로드 되어야 한다. 따라서 시스템 인터페이스의 중요한 사항은 시스템의 쓰기 제어신호를 감지하여 송신 데이터를 래치 한 후 직렬 송신 레지스터를 초기화 시킬수 있도록 제어신호 txdatardy 를 생성하는 것이다. 다음은 외부로부터 병렬 데이터와 쓰기 과정의 제어를 기술한 것이다. 주된 내용은 UART 의

http://www.cvsl.kyunghee.ac.kr

송신 클럭과 무관하게 주어지는 WRITE 의 상승과 하강을 감지하여 직렬 송신 시작을 제어하게 될 제어신호 txdatardy 의 발생 기능이다.

```
PROCESS (reset, clkx16)
VARIABLE wr1,wr2: std_logic;
VARIABLE txdone1: std_logic;
BEGIN
    IF reset='1' THEN
        txdatardy <= '0' ;
        wr1 := '0' ;
        wr2 := '0' ;
        txdone1 := '0' ;
    ELSIF clkx16'event AND clkx16 = '1' THEN
        IF wr1 = '0' AND wr2= '1' THEN
            txdatardy  <= '1';
        ELSIF txdone = '0' AND txdone1 = '1' THEN
            txdatardy  <= '0';
        END IF;

        wr2 := wr1;
        wr1 := write;

        txdone1 := txdone;

    END IF ;
END PROCESS;
```

회로를 설계할 때 클럭 이외의 제어신호를 직접 트리거로 사용하는 것은 옳지 않다. UART 의 모든 내부 동작은 상승 에지(rising-edge)에서 이루어지도록 설계되었다. 그러나 외부로부터 주어지는 제어신호 “write”의 트리거 에지는 상승 혹은 하강 어느것도 보장할 수 없다. 따라서 쓰기 제어신호 “write”의 상승과 하강의 감지를 위해서 2 비트 쉬프트 레지스터를 사용 함으로서 clkx16 클럭 신호와 동기를 맞추도록 한다. 그림 7 은 2 비트 쉬프트 레지스터를 이용하여 외부 제어신호 “write”의 하강을 감지한 후 내부 클럭 clkx16 의 상승 에지에 맞춰 내부 제어신호인 “txdatardy”를 생성하는 회로와 타이밍이다. 이러한 방법은 비동기 신호의 에지를 검출하여 동기 제어신호로 처리하는데 자주 이용된다. 나중에 다루겠지만 UART 에서 수신 신호의 시작 비트 검출에도 같은 방법이 이용될 것이다. VHDL 로 표현하면 다음과 같다.

```
IF clkx16'event AND clkx16 = '1' THEN
    -- detect falling edge “WRITE”
    IF wr1 = '0' AND wr2= '1' THEN
        txdatardy  <= '1';
    END IF;
    -- Shift Register
    wr2 := wr1;
```

http://www.csvlsi.kyunghee.ac.kr

```
        wr1 := write;
    END IF ;
```

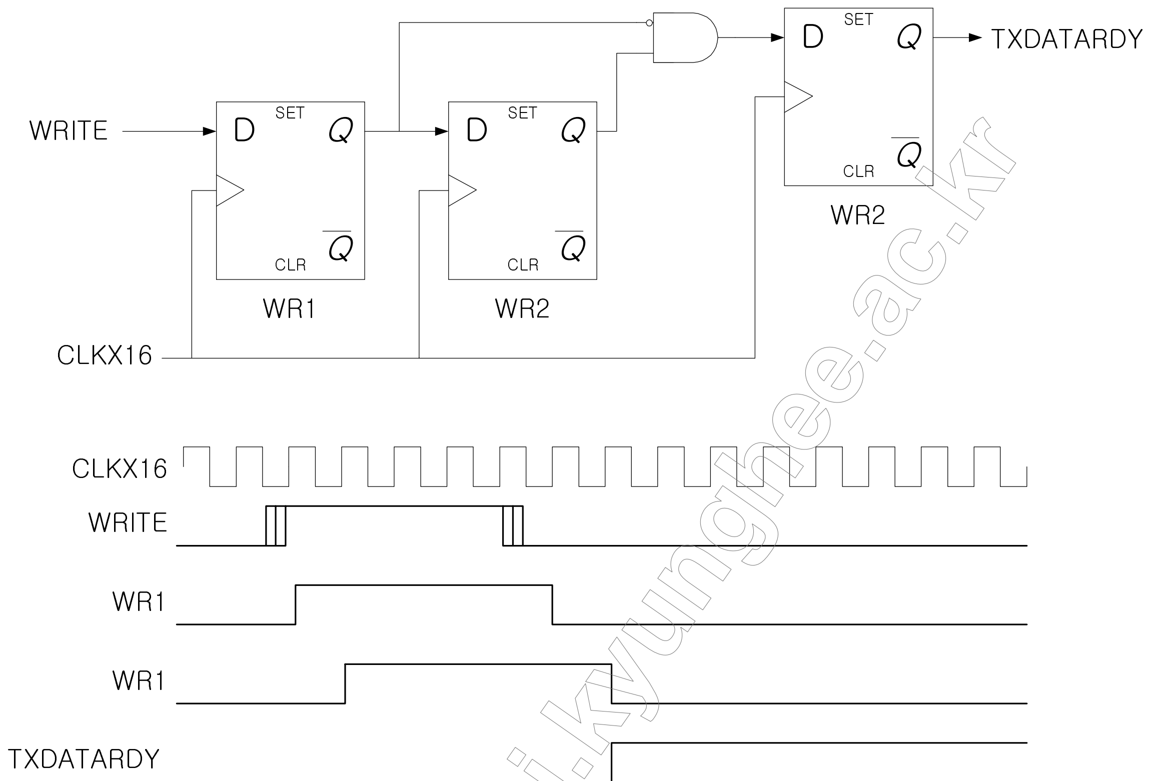


그림 7. 2비트 쉬프트 레지스터를 이용한 외부 제어신호 "write"의 하강 감지

직렬 송신부(TX unit)와 외부의 데이터 인터페이스와의 제어 동작 타이밍은 그림 8과 같다.

10비트 프레임 직렬송신이 완료되면 모든 송신 버퍼 레지스터와 txtag1, txtag2가 '0'이 되므로 txdone(txrdy)='1'. txrdy='1'이면 외부로부터 병렬 데이터를 받아들일 수 있다.
txdone은 txdone1으로 래치
외부로부터 data가 준비되면 write='1'. 이때 data는 txhold에 래치
write='1'이 wr1 과 wr2로 shift 되면서 write의 falling edge가 검출되면 txdatardy가 set.
txdone='1', txdatardy='1'인 상태에서 송신 레지스터 txreg에 txhold가 로드되고 txtag1과 txtag2가 set으로 초기화. 초기화 과정에 의하여 txdone='0'
txdone='0'과 txdone1='1'일때 txdatardy='0' 래치
txdatardy='0'인 구간에서는 직렬 송신 시작 (shift)

그림 9는 VHDL로 기술한 비동기 직렬 송신부와 외부 시스템 인터페이스 회로를 합성한 회로의 일부분이다. 외부의 제어신호 "write"를 감지하기 위한 부분과 송신종료를 표시하는 txdone, txdone1 을 주의 깊게 살펴 보기 바란다.

http://www.csvlsi.kyunghee.ac.kr

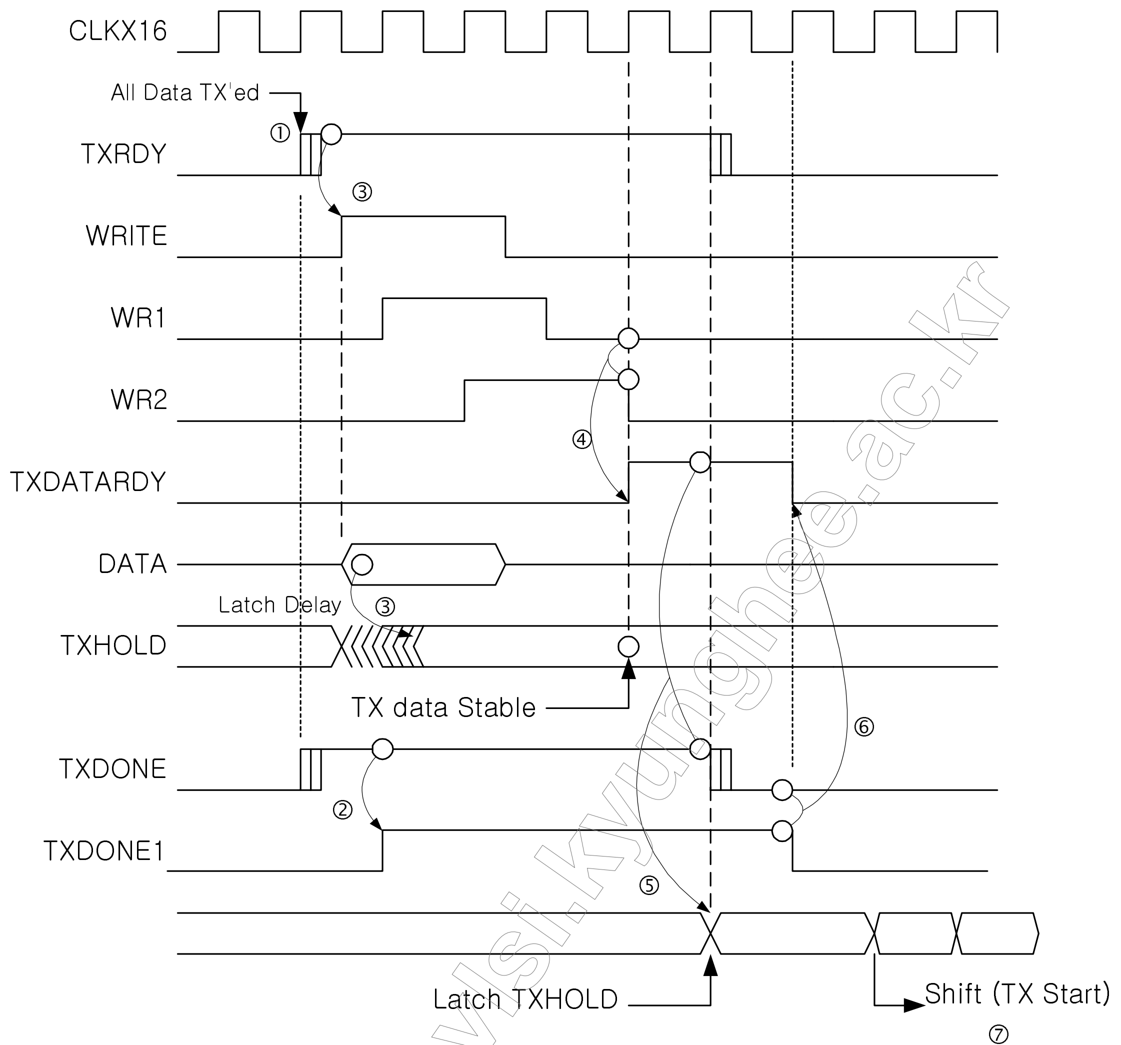


그림 8. 비동기 직렬 통신부와 외부 병렬 데이터 WRITE 제어 타이밍

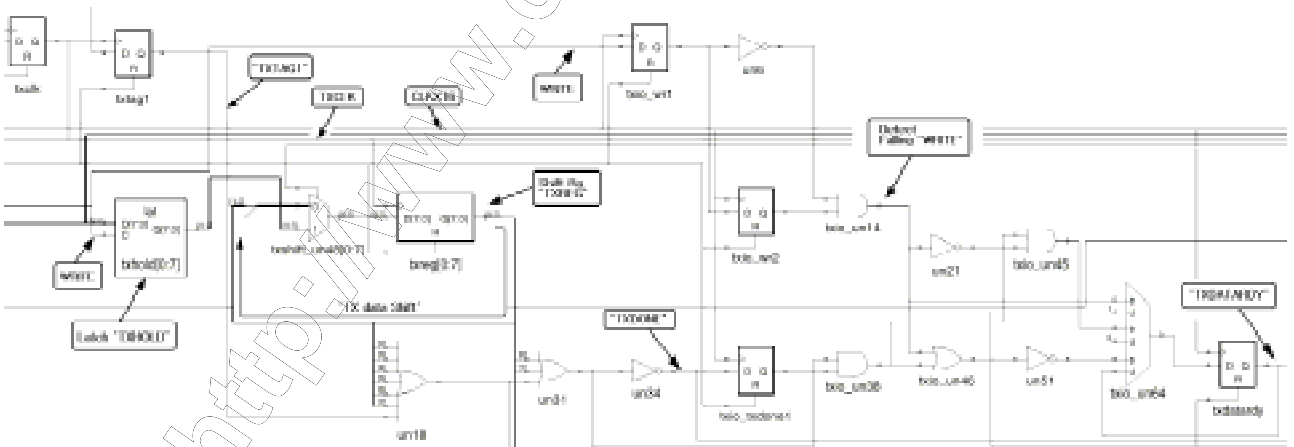


그림 9. VHDL 로부터 합성된 비동기 직렬 통신부와 외부 병렬 데이터 WRITE 제어회로

http://www.csvlsi.kyunghee.ac.kr

1-4. 수신부 (RX unit)

비동기 직렬 통신 수신부를 설계한다. 주요기능은 수신 클럭 발생(make_rxclk), 직렬 데이터 수신(rx_proc), 시스템 인터페이스(rxio)로 구성된다. 직렬 데이

터수신은 송신에 비하여 다소 복잡하다. 대기 상태(idle)에서 직렬 데이터 프레임의 시작 비트를 검출하고 데이터 수신이 완료되면 패리티, 프레이밍, 오버런 등의 에러를 검사한다.

- 수신 클럭 발생기 (make_rxclk)

수신 클럭은 수신 대기상태(idle)에서 시작 비트의 검출이 이루어진 후 송신 때와 마찬가지로 시스템 클럭 clkx16 을 16 분주하여 발생 시킨다. 수신 클럭 발생기(make_rxclk)의 VHDL 은 다음과 같다.

```
PROCESS  (reset, clkx16)
    VARIABLE rxcnt : unsigned(0 TO 3);
    VARIABLE rx1   : std_logic;
    VARIABLE hunt  : boolean;
BEGIN
    IF reset='1' THEN
        hunt := FALSE ;
        rxcnt := (OTHERS=>'0') ;
        rx1 := '0' ;
        rxclk <= '0' ;
    ELSIF clkx16'EVENT AND clkx16 = '1' THEN

        rxclk <= rxcnt(0);

        IF (rxidle = '1' AND rx = '0' AND rx1 = '1') THEN
            hunt := TRUE;
        END IF ;

        IF rxidle = '0' THEN
            hunt := FALSE;
        END IF;

        rx1 := rx;

        IF (rxidle = '0' OR hunt) THEN
            rxcnt := rxcnt + "0001";
        ELSE
            rxcnt := "0001";
        END IF;

    END IF ;
END PROCESS;
```

수신 클럭을 발생시키는 분주회로는 송신 때의 경우와는 달리 4 비트 카운터를 이용하여 MSB 를 클럭 신호로 사용한다. 직렬 수신에서 시작 비트가 검출되지 않은 대기 상태에서는 분주용 카운터가 정지 상태에 있게 되며 카운터는 초기

http://www.csvlsi.kyunghee.ac.kr

값을 "0001"을 갖는다. 이는 시작 비트의 검출을 위해서 수신 입력에 1비트 레지스터를 이용하였기 때문이다. 시작 비트가 검출된 이후에는 4비트 카운터를 이용하여 clkx16을 16분주한 수신 클럭 "rxclk"이 연속적으로 발생 된다. 수신 클럭은 10비트 직렬 데이터 프레임이 모두 수신 되었을 때 rxidle 에 의하여 정지 된다. 그림 10은 수신 클럭 발생기 회로와 타이밍은 그림 10과 같다.

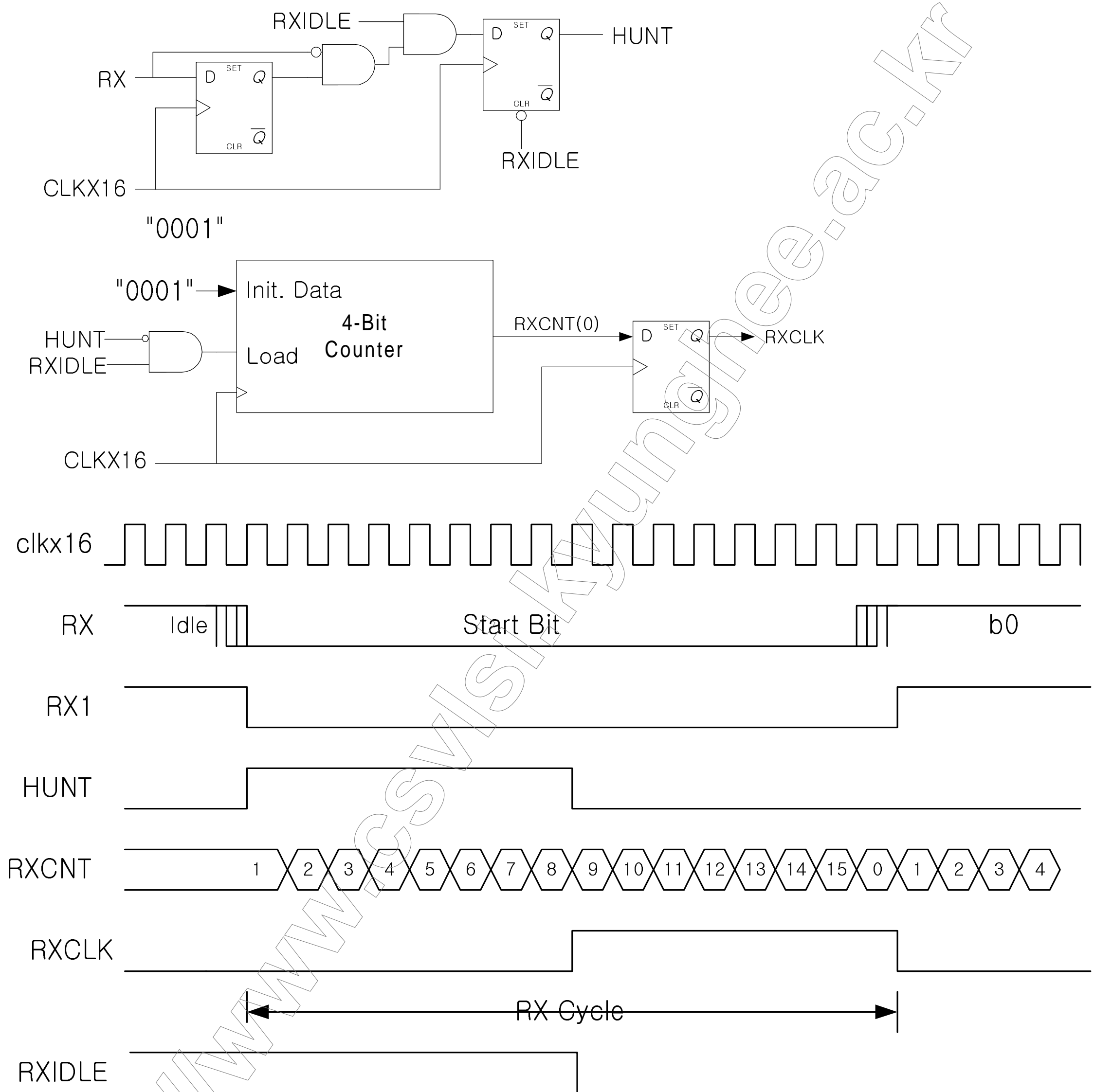


그림 10. 수신 클럭 발생기(make_rxclk) 회로와 타이밍

- 직렬 데이터 수신 (rx_proc)

송신 때와 마찬가지로 8비트 데이터와 패리티 비트 시작 비트를 포함한 10비트 데이터 프레임을 수신한다. 데이터 비트 수신 중 패리티 계산을 수행하고 패리티 에러를 검사한다. 직렬 데이터 수신부의 VHDL 코드는 다음과 같다.

```
PROCESS (reset, rxclk)
```

http://www.csvlsi.kyunghee.ac.kr

```
    BEGIN
        IF reset='1' THEN
            rxreg <= (OTHERS=>'0') ;
            rxparity <= '0' ;
            paritygen <= '0' ;
            rxstop <= '0' ;
        ELSIF rxclk'event AND rxclk = '1' THEN
            IF rxidle = '1' THEN
                rxreg <= (OTHERS=>'1');
                rxparity <= '1';
                paritygen <= '1';        -- Odd parity
                rxstop <= '0';
            ELSE
                rxreg <= rxreg (1 TO 7) & rxparity;
                rxparity <= rxstop;
                paritygen <= paritygen XOR rxstop;
                rxstop <= rx;
            END IF ;
        END IF;
    END PROCESS;
```

위의 VHDL 코드로 기술된 직렬 데이터 수신부의 쉬프트 레지스터 구조는 그림 11 과 같다. 수신부 제어의 시작은 대기상태를 나타내는 rxidle 이다. “rxidle” 은 10 비트 직렬 데이터 프레임이 모두 수신된 것을 감지한 후 직렬 수신의 완료를 표시하게 된다. 비동기 수신 제어선인 “rxidle” 의 발생시키는 회로의 VHDL 코드는 다음과 같다.

```
    PROCESS ( reset, rxclk )
    BEGIN
        IF reset = '1' THEN
           rxidle <= '0';
        ELSIF rxclk'EVENT and rxclk = '1' THEN
           rxidle <= NOT rxidle AND NOT rxreg(0);
        END IF;
    END PROCESS;
```

수신부의 쉬프트 레지스터는 초기화 될 때 ‘1’이며 RX 를 통해서 시작 비트가 수신 쉬프트 레지스터 LSB 인 “b0”에 도달하면 10 비트 프레임이 모두 수신된 것과 같다. 이때부터 직렬 수신은 대기 상태가 되며 수신된 데이터를 외부 시스템에 출력할 수 있다.

http://www.csvlsi.kyunghee.ac.kr

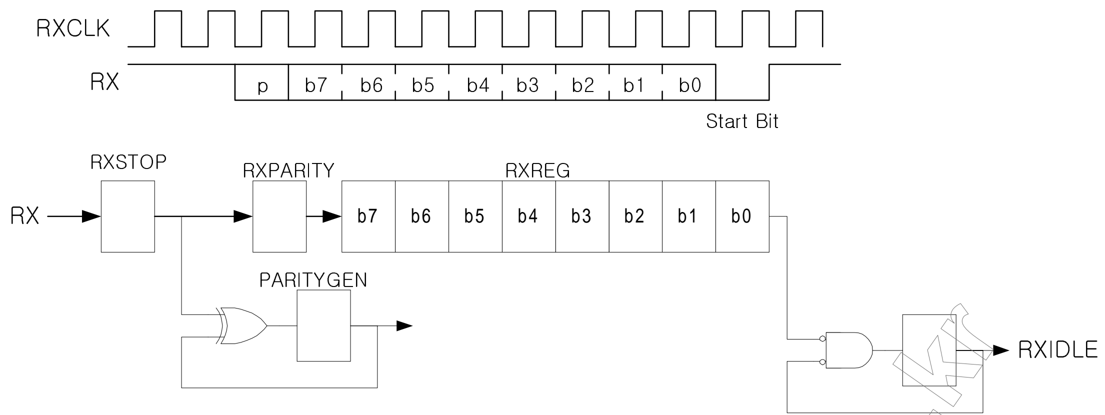


그림 11. 직렬 데이터 수신부의 쉬프트 레지스터 구조

- 수신 데이터의 시스템 인터페이스 (rxio)

직렬 데이터의 수신이 완료 되면 대기 신호 rxidle이 상승하는 것을 감지하여 수신 레지스터의 8비트 값을 시스템에 출력할 수 있도록 수신 데이터 준비 신호 rxdatardy를 출력한다. 외부 시스템에서 읽기 스트로브 신호인 read가 주어지면 비로서 외부 버스에 수신한 8비트 데이터를 출력하고 아울러 에러 신호 등도 함께 출력 한다. 수신 데이터의 시스템 인터페이스를 VHDL 기술하면 다음과 같다.

```
PROCESS (reset, clkx16)
    VARIABLE rd1, rd2 : std_logic;
    VARIABLE rxidle1  : std_logic;
BEGIN
    IF reset='1' THEN
        overrun <= '0' ;
        rxhold <= (OTHERS=>'0') ;
        parityerr <= '0' ;
        framingerr <= '0' ;
        rxdatardy <= '0' ;
        rd1 := '0' ;
        rd2 := '0' ;
        rxidle1 := '0' ;
    ELSIF clkx16'event AND clkx16 = '1' THEN
        -- 수신 대기신호가 상승하면
        -- 수신데이터 시스템 인터페이스 준비
        IF rxidle = '1' AND rxidle1 = '0' THEN
            IF rxdatardy = '1' THEN
                overrun <= '1';
            ELSE
                overrun <= '0';
                -- 수신 데이터 래치
                rxhold <= rxreg;
                -- 패리티  및 프레이밍에러
```

http://www.csvlsi.kyunghee.ac.kr

```
                    parityerr  <= paritygen;
                    framingerr <= NOT rxstop;
                    -- 수신 데이터 준비신호
                    rxdatardy <= '1';
                END IF;
            END IF;
            -- 수신 대기신호 상승 감지
            rxidle1 := rxidle;

            -- 읽기 신호 하강이면 초기화
            IF (NOT rd2 AND rd1) = '1' THEN
                rxdatardy  <= '0';
                parityerr  <= '0';
                framingerr <= '0';
                overrun    <= '0';
            END IF;
            -- 외부 시스템으로부터의 읽기 신호 감지
            rd2 := rd1;
            rd1 := read;

            IF reset = '1' THEN
                rxdatardy <= '0';
            END IF;

        END IF ;
    END PROCESS;
```

2. UART 기능 시뮬레이션

설계한 UART 의 기능을 검증하기 위하여 시뮬레이션 한다. 시뮬레이션을 위해서 Testbench 를 작성하여야 한다. Testbench 는 일종의 시험 틀로서 이 역시 VHDL 을 이용하여 작성하도록 한다. Testbench 에 들어 가게 될 내용은 클럭, 입력 테스트 벡터, 예상되는 출력, 그리고 출력 값의 검증을 위한 비교 등이다. UART 의 시뮬레이션을 위한 testbench 는 다음과 같다. 송수신 부를 모두 가지고 있으므로 RX 와 TX 를 서로 연결하여 루프 테스트가 되도록 한다.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity tb_uart is
end tb_uart;

architecture tb_uart_arch of tb_uart IS
```

http://www.csvlsi.kyunghee.ac.kr

```
  component uart
    port ( clkx16     : IN    std_logic;
           read       : IN    std_logic;
           write      : IN    std_logic;
           rx         : IN    std_logic;
           reset      : IN    std_logic;
           tx         : OUT   std_logic;
           rxrdy      : OUT   std_logic;
           txrdy      : OUT   std_logic;
           parityerr  : OUT   std_logic;
           framingerr : OUT   std_logic;
           overrun    : OUT   std_logic;
           data       : INOUT std_logic_vector(0 TO 7));
  end component;

  signal clkx16 : std_logic;
  signal read, write, rx, reset, tx, rxrdy, txrdy,
         parityerr, framingerr, overrun : std_logic := '0';
  signal data : std_logic_vector(0 TO 7);
  signal data_in : std_logic_vector(0 TO 7);
  signal data_out : std_logic_vector(0 TO 7);

  constant CLK_PERIOD : time := 125 ns;
begin

  ---------------------------------------
  -- Instantiate Device Under Test (DUT)
  ---------------------------------------
  dut : uart port map (
     clkx16, read, write, rx, reset,
     tx, rxrdy, txrdy, parityerr, framingerr, overrun,
     data );

  --------------------------
  -- Clock and Reset Signals
  --------------------------
  clk: process
  begin
    clkx16 <= '0';
    wait for CLK_PERIOD/2;
    clkx16 <= '1';
    wait for CLK_PERIOD/2;
  end process;

  rst : reset <= '1', '0' after CLK_PERIOD;
```

```
  ------------------------------------------------
  -- loop the output (tx) back into the input (rx)
  ------------------------------------------------
  loopback: rx <= tx;

  -----------------------------------------------------
  -- Write up-count sequence to Parallel Data Input bus
  -----------------------------------------------------
  Counter: process
    variable count : std_logic_vector (0 to 7) := "00000000";
  begin
    wait until txrdy = '1';
    count := count - "00000001";
    data_in <= count;  -- display the data going to the UART
    data <= count, ('Z','Z','Z','Z','Z','Z','Z','Z')
                                 after 2*CLK_PERIOD;
    write <= '1', '0' after 2*CLK_PERIOD;
  end process;

  ----------------------------------------------------------
  -- Read data from serial input buffer when rxrdy goes high
  ----------------------------------------------------------
  reader: process
  begin
    wait until rising_edge(rxrdy);
    read <= '1', '0' after 2*CLK_PERIOD;
    wait for CLK_PERIOD;
    data_out <= data ;         -- data_out latches data from the last
rxrdy
  end process;

end tb_uart_arch;
```

3. VHDL 합성과 회로의 구현

VHDL 로 기술한 것은 UART 의 기능을 표현한 것이다. 이는 다른 프로그래밍 언어로서 시뮬레이터를 만든 것과 다를 바 없다. 합성기는 VHDL 로 표현된 하드웨어를 디지털 소자의 네트리스트로 변환하여 준다. 이때 논리 최적화도 이루어진다. 합성을 수행하기 위해서 필요한 사항은 어느 제조 공정을 이용할 것인지 결정해야 한다. 이 결정에 참고할 사항은 생산 수량과 비용이다. ASIC 화 할 경우 수개월의 기간과 상당한 초기 비용이 소요된다. 그러나 수 만개 이상의 대량생산을 할 경우 생산 단가 면에서 이점이 있다. CPLD 나 FPGA 를 이용하는 경우 제작 기간과 초기 비용에 있어서 상당한 강점이 있으나 대량생산 할 경우 경제성이 떨어진다. 이러한 점을 고려하여 개발단계에서 PLD 를 이용하여 Prototype 설계를 마친 후 대량생산 할 때 ASIC 공정을 이용하는 것이 일반적인 수순이라 하겠다. 이렇게 함으로서 성공적인 설계가 되기까지의 위험 부담

http://www.csvlsi.kyunghee.ac.kr

을 을 감소시키고 신속한 설계를 완성할 수 있다.

설계한 한 UART 를 CPLD 로 합성해 보고 타이밍 시뮬레이션을 수행함으로써 완전한 설계가 되도록 검증해 보기로 한다. 합성을 위해서는 대상 소자를 정해야 하는데, 최근 들어 상당히 많은 벤더들에서 다양한 PLD 들이 생산되고 있으며 규모 면에서도 10 만 게이트 급의 소자들이 일반화 되고 있는 추세이다. 합성 결과 얻어지는 네트리스트 파일 이외에 로그 파일을 살펴보면 소요된 하드웨어의 규모를 어느 정도 예상해 볼 수 있다.

위에서 설계한 UART 를 메타모어 사(Metamor Inc.) 합성기를 이용하여 네트리스트를 얻은 후 알테라 사의 ISP 타입 CPLD 인 7000S 시리즈로 구현하도록 한다. 합성과 회로를 구현할 대상 CPLD 소자를 선정해야 하므로 특정 회사명을 거론하게 된 것이 유감이다.

모든 VHDL 구문이 합성에 의하여 회로로 변환될 수 없다는 것은 이미 알고 있는 사실일 것이다. 이외에 합성을 위하여 몇 가지 준비해 두어야 할 사항이 있다. 지금부터 거론하는 내용은 일반적인 사항은 아니며 사용하는 합성 툴이나 대상 PLD 소자에 따라 다소 다를 수 있다. 다만 합성과 구현의 과정상 알아두어야 할 내용에 대하여 설명 하고자 한다.

3-1. Top-Entity VHDL 의 준비

칩은 코어와 외부의 핀 연결을 위한 패드로 구분 된다. 패드는 포트의 입출력 방향에 따라 입력, 출력, 양방향 3-state 로 나뉘고, 클럭과 리셋과 같이 코어의 전반에 걸쳐 강력한 구동능력을 갖춘 것들도 있다. ASIC 의 경우 적당한 패드 위치에 패드 라이브러리 쎌을 불러다가 배치 하면 되지만 PLD 의 경우 미리 위치가 정해진채 생산된다. 따라서 구현 대상 PLD 를 선정한 후 데이터 북 등을 참조하여 입출력 핀들을 지정해 주어야 한다.

VHDL 상에서 입출력 핀을 지정하는 방법은 ATTRIBUTE 구문을 이용하여 지정하며 구체적인 내용은 합성기 마다 다르므로 매뉴얼을 참조한다. 또한 다수의 서브 디자인으로 계층적인 설계의 경우 입출력 핀을 지정할 때에는 Top-Entity 에 기술하도록 한다. 또한 버스 형태의 경우 각각의 핀은 단일 핀으로 분할하여 기술하는 것이 좋다.

다음은 UART 의 Top-Entity 로서 아래의 예는 메타모어 사 합성기에서 지원하는 ATTRIBUTE 구문을 이용하여 입출력 핀 번호를 지정하고 양방향성 3-state 패드를 기술한 것이다.

```
LIBRARY ieee;
use ieee.std_logic_1164.all;

library metamor;
use metamor.attributes.all;

ENTITY uart_top IS
```

http://www.cs.kyungnee.ac.kr

```
PORT ( clkx16     : IN    std_logic;
       read       : IN    std_logic;
       write      : IN    std_logic;
       rx         : IN    std_logic;
       reset      : IN    std_logic;
       tx         : OUT   std_logic;
       rxrdy      : OUT   std_logic;
       txrdy      : OUT   std_logic;
       parityerr  : OUT   std_logic;
       framingerr : OUT   std_logic;
       overrun    : OUT   std_logic;
       data0      : INOUT std_logic;
       data1      : INOUT std_logic;
       data2      : INOUT std_logic;
       data3      : INOUT std_logic;
       data4      : INOUT std_logic;
       data5      : INOUT std_logic;
       data6      : INOUT std_logic;
       data7      : INOUT std_logic );

-------------------------------------------------------------------------
-- Synthesis specific ATTRIBUTES
-- Following attributes are applied to "METAMOR's"
-- usage of part_name ------------------------------------------
ATTRIBUTE part_name OF uart_top : ENTITY IS  "EPM7128STC100-10";
-- define pinnum ATTRIBUTE -------------------------------------
ATTRIBUTE pinnum : string;
ATTRIBUTE pinnum OF clkx16     : SIGNAL IS "87";
ATTRIBUTE pinnum OF read       : SIGNAL IS "88";
ATTRIBUTE pinnum OF write      : SIGNAL IS "97";
ATTRIBUTE pinnum OF rx         : SIGNAL IS "93";
ATTRIBUTE pinnum OF tx         : SIGNAL IS "75";
ATTRIBUTE pinnum OF rxrdy      : SIGNAL IS "67";
ATTRIBUTE pinnum OF txrdy      : SIGNAL IS "85";
ATTRIBUTE pinnum OF parityerr  : SIGNAL IS "71";
ATTRIBUTE pinnum OF framingerr : SIGNAL IS "65";
ATTRIBUTE pinnum OF overrun    : SIGNAL IS "64";
ATTRIBUTE pinnum OF reset      : SIGNAL IS "99";
ATTRIBUTE pinnum OF data0      : SIGNAL IS "63";
ATTRIBUTE pinnum OF data1      : SIGNAL IS "68";
ATTRIBUTE pinnum OF data2      : SIGNAL IS "57";
ATTRIBUTE pinnum OF data3      : SIGNAL IS "56";
ATTRIBUTE pinnum OF data4      : SIGNAL IS "55";
ATTRIBUTE pinnum OF data5      : SIGNAL IS "54";
ATTRIBUTE pinnum OF data6      : SIGNAL IS "53";
ATTRIBUTE pinnum OF data7      : SIGNAL IS "52";
```

http://www.cvlsi.kyunghee.ac.kr

```
-- declare clock_buffer ATTRIBUTE ------------------------------
-- and mark port CLKX16 as using a clock buffer
ATTRIBUTE clock_buffer : boolean;
ATTRIBUTE clock_buffer OF clkx16 : SIGNAL IS true;
--------------------------------------------------------------------------

END uart_top;

ARCHITECTURE struct OF uart_top IS

COMPONENT uart
PORT ( clkx16      : IN    std_logic;
       read        : IN    std_logic;
       write       : IN    std_logic;
       rx          : IN    std_logic;
       reset       : IN    std_logic;
       tx          : OUT   std_logic;
       rxrdy       : OUT   std_logic;
       txrdy       : OUT   std_logic;
       parityerr   : OUT   std_logic;
       framingerr  : OUT   std_logic;
       overrun     : OUT   std_logic;
       rxhold      : OUT std_logic_vector(0 TO 7);
       txhold      : IN  std_logic_vector(0 TO 7) );
END COMPONENT;

SIGNAL txhold : std_logic_vector(0 TO 7);
SIGNAL rxhold : std_logic_vector(0 TO 7);

BEGIN

    u_uart : uart
    PORT MAP ( clkx16, read, write, rx, reset,
               tx, rxrdy, txrdy, parityerr, framingerr, overrun,
               rxhold, txhold );

    -- Drive data bus only during read
    data0 <= rxhold(0) WHEN read = '1' ELSE 'Z';
    data1 <= rxhold(1) WHEN read = '1' ELSE 'Z';
    data2 <= rxhold(2) WHEN read = '1' ELSE 'Z';
    data3 <= rxhold(3) WHEN read = '1' ELSE 'Z';
    data4 <= rxhold(4) WHEN read = '1' ELSE 'Z';
    data5 <= rxhold(5) WHEN read = '1' ELSE 'Z';
    data6 <= rxhold(6) WHEN read = '1' ELSE 'Z';
    data7 <= rxhold(7) WHEN read = '1' ELSE 'Z';
```

http://www.esvlsi.kyunghee.ac.kr

```
    -- Latch data bus during write
    txhold(0) <= data0 WHEN write = '1' ELSE txhold(0);
    txhold(1) <= data1 WHEN write = '1' ELSE txhold(1);
    txhold(2) <= data2 WHEN write = '1' ELSE txhold(2);
    txhold(3) <= data3 WHEN write = '1' ELSE txhold(3);
    txhold(4) <= data4 WHEN write = '1' ELSE txhold(4);
    txhold(5) <= data5 WHEN write = '1' ELSE txhold(5);
    txhold(6) <= data6 WHEN write = '1' ELSE txhold(6);
    txhold(7) <= data7 WHEN write = '1' ELSE txhold(7);

END struct;
```

위와 같은 버스를 분할하여 작성된 Top-Entity 를 시뮬레이션 하기 위해서 testbench 를 다음과 같이 수정하도록 한다.

```
signal data : std_logic_vector(0 TO 7);

dut : uart_top
port map (
     clkx16, read, write, rx, reset,
     tx, rxrdy, txrdy, parityerr, framingerr, overrun,
     data(0), data(1), data(2), data(3),
     data(4), data(5), data(6), data(7));
```

3-2. 합성 및 배치-배선 결과

Top-Entity 를 합성하여 네트리스트 파일을 얻는다. 여러 개의 서브 디자인으로 구성된 경우라도 Flat 한 1 개의 네트리스트 파일이 된다. 네트리스트 파일은 PLD 벤더들의 배치 및 배선(Place & Route, P&R) 툴에 따라 종류가 다르지만 EDIF 가 대부분이며 XNF, SRC,TDF 등이 사용되는 경우도 있다. 다음은 UART 를 합성한후 얻은 로그 파일의 일부분이다.

```
   tri-states = 8
   flip flops with asynchronous reset = 41
   flip flops with asynchronous reset, clock enable = 9
   latches with no set or reset = 8
   macrocell instances = 1
   combinational logic area estimate = 87 two input gates
```

합성결과 사용된 하드웨어의 규모를 짐작할 수 있다. 8 개의 3-state 와 8 개의 래치, 총 50 개의 플립-플롭이 사용된 것을 알 수 있다. 그 이외에 1 개의 매크로셀(이는 PLD 벤더 사 마다 내용이 달라지는데 설계한 UART 의 경우 클럭 분주용 카운터가 매크로 셀로 합성 되었다), 대략 87 개정도의 2 입력 게이트와 등가인 조합 논리 회로들이 사용되었음을 보여준다. 대략 60 여개의 플립-플롭과 래치가 사용 되었으므로 알테라의 7000 시리즈 CPLD 중 7064 혹은 7128 을 이용하여 구현하도록 한다. 회로의 구현은 합성으로 얻어진 EDIF 네트리스트를

이용한 P&R 과정이다. P&R 에 사용되는 툴은 PLD 벤더사에 의하여 제공되는데 알테라 사의 툴은 맥스 플러스 이다. 맥스 플러스 툴에서는 P&R 과정을 “Compile” 한다고 한다. 컴파일 과정에서 다음과 같은 메시지를 보게될 것이다.

```
Info : All assignments from EDIF Input File(s) have been added to the project's ACF
```

이는 VHDL 을 합서하여 얻은 EDIF 네트리스트로부터 핀 번호 와 같은 파라메터를 반영하여 P&R 을 수행 하겠다는 것을 의미한다. P&R 을 수행한 후 리포트 파일을 살펴보자. 다음은 맥스 플러스에서 생성된 P&R 리포트 파일의 일부분이다. P&R 에 사용된 CPLD 의 종류 (EPM7128STC100), 사용된 로직 셀의 개수, 각종 입출력 핀의 할당표 등등을 확인할 수 있다.

```
** DEVICE SUMMARY **
Chip                    Input Output Bidir      Shareable
POF      Device          Pins  Pins   Pins  LCs Expanders  % Utilized
uart_top EPM7128STC100    5     6      8    66     1         51 %

** PIN/LOCATION/CHIP ASSIGNMENTS **
                    Actual
    User         Assignments
Assignments    (if different)     Node Name

uart_top@87                        clkx16
uart_top@63                        data0
uart_top@68                        data1
uart_top@57                        data2
uart_top@56                        data3
uart_top@55                        data4
uart_top@54                        data5
uart_top@53                        data6
uart_top@52                        data7
uart_top@65                        framingerr
uart_top@64                        overrun
uart_top@71                        parityerr
uart_top@88                        read
uart_top@99                        RESET
uart_top@93                        rx
uart_top@67                        rxrdy
uart_top@75                        tx
uart_top@85                        txrdy
uart_top@97                        write
```

http://www.csvlsi.kyunghee.ac.kr

4. 타이밍 시뮬레이션

P&R 을 마치면 이제 회로를 실제 칩에 제작한 것과 같은 수준의 타이밍 시뮬레이션을 할 수 있게 된다. 타이밍 시뮬레이션을 함으로서 실제 제작된 칩이 정확한 동작을 하는지 동작속도는 만족하는지 등등을 검증할 수 있다. 타이밍 검증을 만족하면 제작된 칩이 정확한 동작을 확신 할 수 있다. 즉, 타이밍 시뮬레이션을 만족하고도 제작된 칩이 정상 동작 하지 않는다면 제조 공정에 문제가 있거나 P&R 툴의 문제일 것이다. 이로 인한 모든 시간과 경제적인 책임은 벤더사의 책임이 된다. 따라서 디자인을 칩의 공정 혹은 제작에 넘길 때 (Sign-Off) 매우 까다롭게 검증하게 된다.

타이밍 시뮬레이션 방법은 여러 가지가 있을 수 있으나 최근에는 시간 지연 모델을 이용한 VITAL 기법이 거의 표준화 되어 가고 있다. VITAL(VHDL Initiative Toward ASIC Library)은 IEEE 1076.4 로 표준화 되어 있으며 오늘날 대부분 ASIC 과 PLD 벤더들이 이 기준에 따라 VHDL 시뮬레이션 모델 라이브러리을 제공하고 있고 시간 지연 모델 파라메터 추출 툴을 제공 한다.

시간 지연 모델의 추출은 P&R 의 결과로 얻어지므로 사전에 VITAL 옵션을 지정해 주어야 한다. P&R 후 얻어지는 시간 지연 모델은 벤더사에서 제공하는 타이밍 시뮬레이션 라이브러리의 네트리스트 형태로 생성되는 Structural VHDL 과 지연 파라메터 파일(Standard Delay Form, SDF)이다. 맥스 플러스 툴의 경우 생성되는 파일의 확장자는 각각 .vho, .sdo 이다. 이 두개의 파일을 이용하여 타이밍 시뮬레이션을 수행하려면 먼저 알테라 사의 VITAL 라이브러리를 컴파일 하여야 한다. 알테라 사의 VITAL 라이브러리 명은 "alt_vtl"이다. 따라서 Structural VHDL 인 .vho 파일의 내용을 보면 다음과 같은 내용을 볼수 있다.

```
LIBRARY alt_vtl;
USE alt_vtl.VCOMPONENTS.all;
```

VITAL 라이브러리는 VHDL 소스 형태로 P&R 툴과 함께 벤더사에서 제공 하므로 이를 컴파일 한후 지정된 라이브러리 이름으로 매핑하여둔다. 다음은 Modeltech 사의 VHDL 시뮬레이터 에서 알테라 VITAL 라이브러리를 컴파일하고 라이브러리 매핑하는 매크로 파일의 내용이다.

```
vlib alt_vtl
vcom -87 -work alt_vtl alt_vtl.vhd
vcom -87 -work alt_vtl alt_vtl.cmp
vmap alt_vtl alt_vtl
```

VITAL 라이브러리가 준비 되면, P&R 에 의하여 생성된 vho 와 sdf 를 이용하여 타이밍 시뮬레이션 한다. 이때 사용되는 Testbench VHDL 은 기능 시뮬레이션 때의 것을 그대로 사용 하여 두 가지 시뮬레이션 결과가 일치함을 확인 하도록 한다. 이 시점에서 UART 의 Top-Entity "uart_top"은 두개의 ARCHITECTURE 를 갖게 된다.

```
ARCHITECTURE struct OF uart_top IS
```

http://www.csvlsi.kyunghee.ac.kr

VHDL 설계 때 기능 시뮬레이션에 사용된 ARCHITECTURE 이다.

ARCHITECTURE ₩EPM7128STC100-10₩ OF uart_top IS

P&R 결과 얻어진 VITAL 네트리스트 형태의 ARCHITECTURE 이다. 맥스 플러스 툴에서는 ARCHITECTURE 명을 사용된 PLD 의 형명으로 짖는다. 이와 같은 상황에서 ENTITY 와 ARCHITECTURE 의 짝 연결을 지정할 때 CONFIGURATION 구문을 이용한다. 다음은 한 개의 ENTITY "uart_top 에 각각 서로 다른 ARCHITECTURE "struct"와 "₩EPM7128STC100-10₩"의 짝 연결을 나타낸 것이다.

```
CONFIGURATION FUNCTIONAL_FOR_uart OF tb_uart IS
    FOR tb_uart_arch
        FOR dut : uart_top
            USE ENTITY work.uart_top (struct);
        END FOR;
    END FOR;
END FUNCTIONAL_FOR_uart;

CONFIGURATION TIMING_FOR_uart OF tb_uart IS
    FOR tb_uart_arch
        FOR dut : uart_top
            USE ENTITY work.uart_top (₩EPM7128STC100-10₩);
        END FOR;
    END FOR;
END TIMING_FOR_uart;
```

기능 시뮬레이션을 수행 할때에는 "FUNCTIONAL_FOR_uart"를 이용한다.

```
vsim FUNCTIONAL_FOR_uart
```

타이밍 시뮬레이션을 수행 할 때에는 P&R 에서 추출된 시간 지연 모델 파라메터를 지정해 준다.

```
vsim  -sdftyp /dut=uart_top.sdo TIMING_FOR_uart
```

설계한 UART 는 비교적 저속 칩인 EPM7128STC100-10 을 상대로 맥스 플러스 툴을 이용하여 P&R 한 후 타이밍 분석결과 55Mhz 의 시스템 클럭으로 동작되는 것으로 나타났으며 타이밍 시뮬레이션 결과 이를 만족하는 것을 검증하였다. 이때 전송율은 약 3.5M bps 정도가 된다.

http://www.csvlsi.kyunghee.ac.kr