

# 합성을 위한 VHDL Coding Style (1)

국일호

[goodkook@csvlsi.kyunghee.ac.kr](mailto:goodkook@csvlsi.kyunghee.ac.kr)

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

VHDL 을 이용한 설계의 최종 목표는 합성을 통한 디지털 회로를 얻는 것이다. 따라서 최적의 설계를 위해서는 디지털 회로를 기술 할 때 설계자는 합성결과에 대하여 상당히 정확한 예측을 할 수 있어야 한다. VHDL 구문과 합성(synthesis) 후에 생성되는 디지털 회로와 주의 점에 대하여 알아보기로 하자. 이를 위하여 합성기에서 이루어지는 합성방식과 조합 논리회로의 경우 합성 및 논리 최적화 과정에 대한 이해가 필요하다. 앞으로 2 회에 걸쳐 합성을 위한 VHDL Coding Style 을 살펴보기로 하겠다. 첫번째에는 플립 플롭(Flip-Flop)과 래치(Latch)의 기술 방법과 주의점, Tri-State Buffer, 양방향 버스의 기술 등에 대해서 살펴보고, 이어서 State Machine 의 기술 방법, 최적의 합성을 얻기 위한 기술 방법으로 산술 연산자들의 활용, Resource Sharing, Mux & Selector, Priority encoder, ROM, PLA, Decoder 등 조합 논리회로의 기술방법을 다루어 보기로 한다.

## 1. 플립 플롭(Flip-Flop)과 래치(Latch)의 기술

VHDL 을 이용한 플립 플롭(Flip-Flop)과 래치(Latch)를 기술하는 방법과 주의 사항을 예제와 합성 결과를 통하여 살펴 보기로 한다.

### [예제 1] 비동기 리셋이 있는 플립-플롭 (Asynchronous Reset Flip-Flop)

클럭의 상승 엣지(rising-edge trigger)에서 작동하는 비동기 리셋 플립-플롭(Asynch. Reset Flip-Flop)의 기술은 다음과 같다.

```
PROCESS (clock)
BEGIN
    IF reset='1' THEN
        q <= '0'; -- RESET 조건에서 상수할당
    ELSIF clock='1' AND clock'EVENT THEN
        q <= d;
    END IF;
END PROCESS;
```

리셋(reset) 조건에서 할당은 상수인 경우가 일반적이며 만일 시그널 할당인 경우 합성결과는 다소 복잡해 질뿐만 아니라 경우에 따라 합성해 내지 못하는 합성기도 있다.

```
u1 : PROCESS(clk,reset)
BEGIN
```

```

IF reset='1' THEN
    o1 <= '0'; -- RESET 조건에서 상수 '0' 할당
ELSIF clk'event AND clk='1' THEN
    o1 <= b;
END IF;
END PROCESS;

```

```

u2 : PROCESS(clk,reset)
BEGIN
    IF reset='1' THEN
        o1 <= a; -- RESET 조건에서 시그널 할당
    ELSIF clk'event AND clk='1' THEN
        o1 <= b;
    END IF;
END PROCESS;

```

그림 1은 리셋 조건에 상수 할당한 경우(u1)와 시그널 할당한 경우(u2)의 합성 결과를 보여준다. 시그널 할당의 경우 합성 결과를 보면 입력 'a'에 대하여 Set/Reset의 경우로 나누어 합성되어 있다.

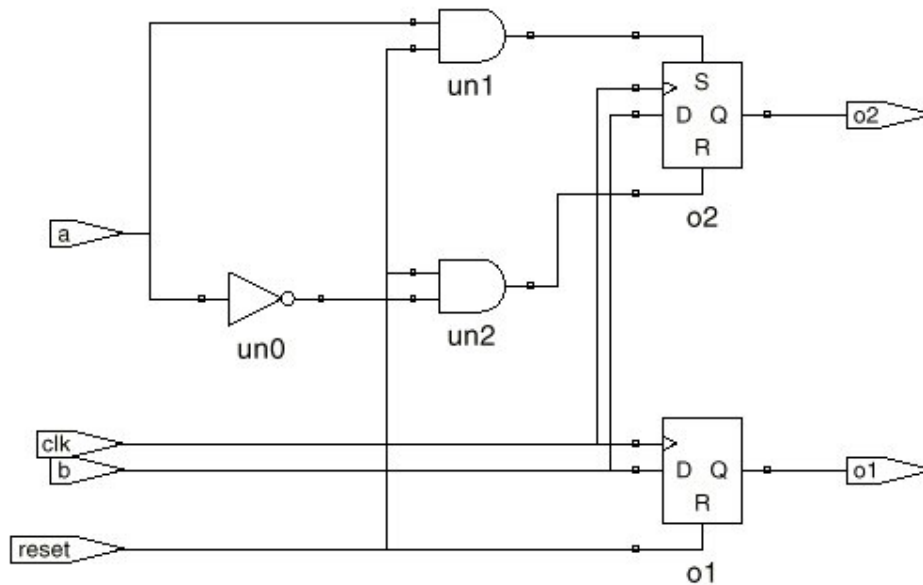


그림 1. 리셋 조건에 상수 할당한 경우(o1)와 시그널 할당(o2)한 경우의 합성 결과

**[예제 2] 동기 셋/리셋 플립 플롭 (Synchronous Set/Reset Flip-Flop)**

클럭의 하강 엣지(falling-edge trigger)에서 작동하는 동기 Set/Reset 8 비트 플립 플롭의 기술은 다음과 같다.

```

PROCESS (clock)
BEGIN
    IF clock='0' AND clock'EVENT THEN
        IF reset='1' THEN
            q(7 downto 0) <= "11110000";
        ELSE
            q(7 downto 0) <= d(7 downto 0);
        END IF;
    END IF;
END PROCESS;

```

위와 같은 경우 q(7 downto 4)는 동기 셀을 갖는 플립 플롭이고 q(3 downto 0)는 리셋을 갖는 플립 플롭이다. IF 문의 조건식에 SIGNAL의 속성 'EVENT'가 사용된 경우 IF ~ END IF;는 무조건 플립 플롭이 된다. 이때 클럭의 상승 혹은 하강 에지(edge)를 표현할 때 'EVENT' 속성이 이용된다. 주의 할 것은 다음과 같이 'EVENT' 속성과 edge의 방향의 조건을 각각 정의된 경우에는 VHDL 구문상 에러는 없으나 합성할 수 없다. 이는 표현된 내용과 실제 디지털 회로를 비교해서 생각해보면 쉽게 불가능한 회로임을 짐작할 수 있을 것이다.

```

PROCESS (clock)
BEGIN
    IF clock'EVENT THEN
        IF clock='1' THEN
            q(7 downto 0) <= e(7 downto 0);
        ELSE
            q(7 downto 0) <= d(7 downto 0);
        END IF;
    END IF;
END PROCESS;

```

그러나 만일 edge의 방향을 나타내는 IF 조건문에 상수 할당을 포함하는 경우에는 합성 가능하다. 신호의 'EVENT' 속성만을 이용한 조건문에서 상수할당이 있는 경우의 합성 결과는 그림 2와 같다. 그러나 이와 같이 합성 되었더라도 이러한 플립-플롭 회로가 설계의도와 부합하는 것인지 따져볼 필요가 있다.

```

PROCESS (clock)
BEGIN
    IF clock'EVENT THEN
        IF clock='1' THEN
            q <= '1';
        ELSE
            q <= d;
        END IF;
    END IF;
END PROCESS;

```

```

END IF;
END PROCESS;

```

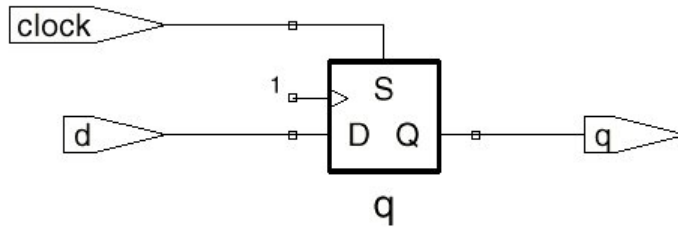


그림 2. 신호의 'EVENT' 속성만을 이용한 조건문에서 상수할당이 있는 경우의 합성 결과

### [예제 3] 래치 (Latch)

조건문의 조건 식에 'EVENT' 속성을 사용하지 않은 경우 래치(Latch)가 된다. 이때 조건문에서 모든 조건의 경우에 대하여 할당되지 않는 경우 불필요한 래치가 되므로 MUX 와 같은 조합 회로를 기술 할 때 주의 하여야 한다. VHDL 을 이용한 회로의 기술에서 가장 흔하게 범할수 있는 실수중 하나가 원치 않은 래치의 발생이다. 따라서 많은 합성기의 결과 리포트를 보면 래치가 발생했다는 사실을 경고(warning)으로 출력하므로 주의 깊게 살펴볼 필요가 있다. 그림 3 은 래치에 대한 합성 결과 이다.

```

PROCESS (LE, d)
BEGIN
    IF LE='1' THEN – Latch Enable
        q <= d;
    END IF;
END PROCESS;

```

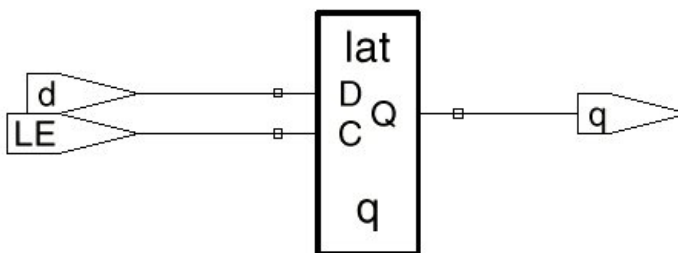


그림 3. 래치의 합성결과

모든 조건의 경우를 빠짐없이 표현되도록 함으로서 불필요한 래치의 생성을 피할수 있다.

IF 조건문에는 ELSE 를 사용하여 불필요한 래치가 발생하는 것을 피할 수 있다.

```

IF (SEL="01") THEN
    Output <= a;
ELSIF (SEL="10") THEN
    Output <= b;
ELSE
    Output <= (OTHERS=>'-'); -- 기타조건에 '-'(don't care)를 할당
END IF;

```

선택문에서는 OTHERS 조건을 사용하여 불필요한 래치가 발생하는 것을 피할 수 있다.

```

CASE SEL IS
    WHEN "01" =>
        Output <= a;
    WHEN "10" =>
        Output <= b;
    WHEN OTHERS
        Output <= (OTHERS=>'-');
END CASE;

```

#### [예제 4] Clock Enable 과 Gated Clock

플립-플롭의 클럭 트리거를 제어하기 위한 방법으로 Clock Enable 과 Gated Clock 을 사용한다. Gated Clock 은 다음과 같다.

```

PROCESS(clk, CE)
BEGIN
    IF CE='1' AND clk'event AND clk='1' THEN -- Gated Clock
        q <= d;
    END IF;
END PROCESS;

```

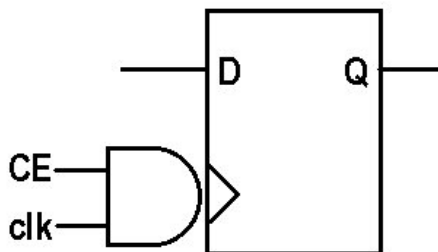


그림 4. Gated Clock 플립 플롭

Gated Clock 과 같이 클럭에 게이트를 통과 시키는 방법은 순차회로의 동작에 글리치를 발생시키는 원인이 되므로 매우 좋지 않다. 순차회로

를 기술할 때에는 어떠한 경우에도 클럭에 게이트를 붙이지 않도록 해야 한다. 따라서 VHDL 합성기들은 설계에 Gated Clock 이 발견되는 경우 경고 메시지를 출력하거나 Clock Enable 플립 플롭으로 합성해 낸다. 다음은 Clock Enable 플립 플롭의 VHDL 기술과 합성 결과이다.

```

PROCESS(clk,CE)
BEGIN
  IF clk'event AND clk='1' THEN
    IF CE='1' THEN
      o2 <= d;
    END IF;
  END IF;
END PROCESS;

```

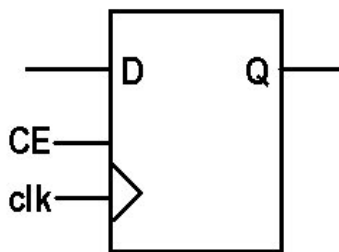


그림 5. Clock Enable 플립 플롭

Gated Clock 을 사용한 경우 플립-플롭의 클럭에 글리치 발생으로 인하여 회로의 동작이 불안해지고 정확한 최대 동작속도의 평가가 이루어지지 않을 수 있다. 대부분 합성기들은 Gated Clock 을 방지하기 위하여 Target Device 에 Clock Enable 라이브러리를 갖지 않은 경우 MUX 를 이용하여 합성한다.

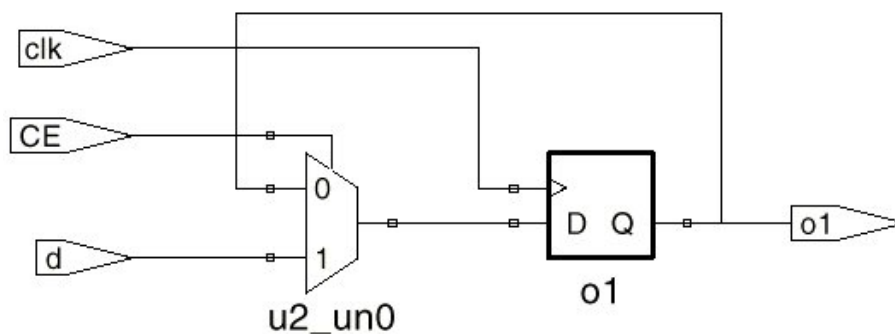


그림 6. Clock Enable 플립 플롭 라이브러리를 갖지 않는 경우 합성결과

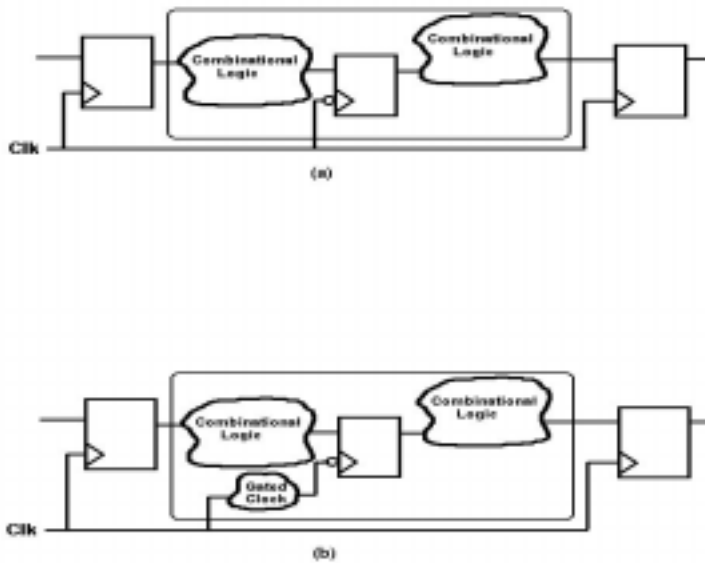


그림 7. 순차회로 설계에 있어서 Clock Enable(a)과 Gated Clock(b)을 사용한 경우의 비교

[예제 4] 병렬 구문에 의한 플립-플롭과 래치의 기술

병렬 구문(Dataflow)에서 래치 및 플립-플롭의 기술은 다음과 같다.

```
-- Edge Sensitive Sync Reset F/F
B1 : BLOCK (clk'EVENT AND clk='1')
BEGIN
    q <= GUARDED '0' WHEN reset='1' ELSE d;
END BLOCK;
```

```
-- Clock Enable
B2 : BLOCK (clk'EVENT AND clk='1')
BEGIN
    q <= GUARDED D WHEN CE='1' ELSE q;
END BLOCK;
```

```
-- Level Sensitive Latch
B1 : BLOCK (le)
BEGIN
    q <= GUARDED d;
END BLOCK;
```

PROCESS 은 순차구문 블록이며 BLOCK 은 병렬구문을 포함한다. 최근 대부분 합성기들이 PROCESS 와 BLOCK, GUARDED 구문들을 모두 지원 하지만 일부 합성기들은 GUARDED 를 지원하지 않는 경우도 있다. 병렬 구문이 좀더 하드웨어적이긴 하지만 PROCESS 구문을 사용하는 것이 일반적이며 이는 순차 구문에 의한 처리가 다양한 표현에 유용하기 때문이다.

**[예제 5] WAIT 구문에 의한 플립 플롭의 기술**

다음은 WAIT 구문에 의한 플립-플롭의 기술이다. PROCESS 블록에 WAIT 구문을 사용할 경우 Sensitivity list 를 사용할 수 없다. WAIT 구문은 조건이 “참(True)”일 경우에 다음 구문을 실행할 수 있으므로 IF 조건문에 비하여 플립-플롭의 명확한 표현이 가능하고 실수에 의한 래치를 방지할 수 있는 장점이 있다. 따라서 플립-플롭의 기술에 WAIT 구문을 선호되기도 한다.

```

PROCESS
BEGIN
    WAIT UNTIL clk'EVENT AND clk='1';
    q <= d;
END PROCESS;

```

**[예제 6] 반전 출력을 갖는 플립 플롭**

반전 출력을 갖는 플립-플롭의 기술은 다음과 같다.

```

architecture behave of resolve is
signal sq : std_logic;
begin

    a : PROCESS (d, clk)
    BEGIN
        IF clk='1' AND clk'EVENT THEN
            sq <= d;
        END IF;
    END PROCESS;
    q <= sq;
    qbar <= NOT sq;

end behave;

```

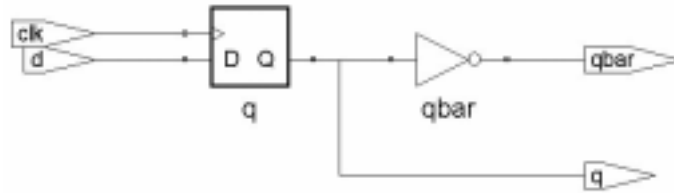
다음과 같이 기술된 경우 반전 출력을 위해서 2 개의 플립-플롭으로 합성되므로 주의 한다. 두가지 반전 출력 플립-플롭의 합성 결과는 그림 8 과 같다.



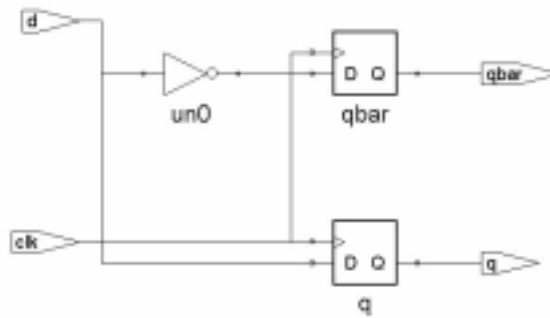
```

b: PROCESS (d, clk)
BEGIN
  IF clk='1' AND clk'EVENT THEN
    q <= d;
    qbar <= NOT d;
  END IF;
END PROCESS;

```



(a)



(b)

그림 8. 반전 출력을 갖는 플립-플롭 합성

## 2. Tri-State, Bi-Directional Buffers

HDL 을 이용한 설계에서 시그널의 할당은 병렬로 이루어진다. 이때 종종 발생하는 문제로서 Multiple-Drive 의 경우이다. 다음은 Multiple-Drive 된 경우의 예이다. 상승과 하강 클럭에서 각각 출력 0 로 할당이 이루어지는 Multiple-Drive 를 볼수 있다. 특히 양 방향성 버스를 다루는 경우 입력과 출력을 표현할 때 버스 충돌(Multiple-Drive)의 경우가 발생하지 않도록 각별히 주의하여야 한다.

### [예제 6] Multiple-Drive 와 Resolved Signal “std\_logic”

```

architecture behave of resolve is
begin

```

```

  u1 : PROCESS(clk)
  BEGIN
    IF clk'event AND clk='1' THEN
      o <= a;

```

```

        END IF;
    END PROCESS;

    u2 : PROCESS(clk)
    BEGIN
        IF clk'event AND clk='0' THEN
            o <= b;
        END IF;
    END PROCESS;

end behave;

```

포트 혹은 시그널의 선언에 `std_ulogic` 의 resolved type 인 `std_logic` 을 사용하는 것은 Multiple-Drive 된 경우 되는 에러 처리에 필요하기 때문이다. 위의 [예제 6]과 같은 ARCHITECTURE 에 입출력 신호가 아래와 같이 resolved type 인 `std_logic` 으로 선언된 경우 에러 없이 VHDL 컴파일-시뮬레이션 된다.

```

ENTITY resolve IS
PORT ( a, b, clk : IN  std_logic;
       o   : OUT std_logic );
END resolve;

```

그러나 시뮬레이션을 수행하면 a 가 '1', b 가 '0' 혹은 a가 '0', b 가 '1' 일 때 출력 o는 'X' (unknown)이 된다. 이는 출력 신호 o에 multiple-drive 충돌이 있다는 것을 의미하는 것이다. 비록 Resolved 된 시그널 타입 `std_logic` 을 사용하여 에러 없이 시뮬레이션 수행된 경우라도 Multiple-Drive 는 합성되지 않는다. 합성기에 따라 “Combinational loop”가 있다는 경고 메시지와 함께 합성되기도 하는데, 이러한 합성결과는 결국 쓸모가 없게 되므로 주의 하여야 한다. 그림 9는 위의 Multiple-Drive 예를 VHDL 컴파일 시뮬레이션 한 결과이다. 그러나 이와 같은 Multiple-Drive 는 합성되지 않는다. 경우에 따라 합성될 때도 있는데 합성한 결과를 살펴보면 VHDL 기능(functional) 시뮬레이션과는 전혀 다를 뿐만 아니라 전혀 가치가 없다.

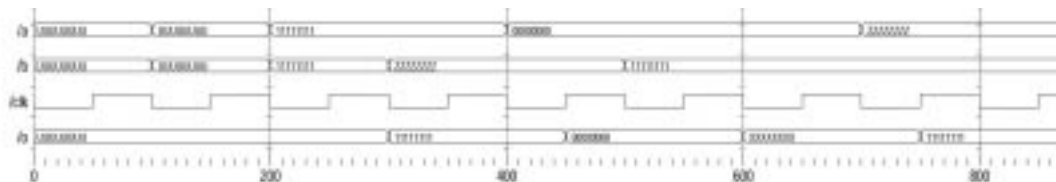


그림 9. Multiple-Drive 의 시뮬레이션

만일 출력 o 가 다음과 같이 Unresolved type 인 `std_ulogic`, 혹은 bit type 으로 선언된 경우에는 VHDL 컴파일-시뮬레이션 되지 않고 “non-resolved signal multiple drive” 에러를 낸다.

```

ENTITY unresolve IS
PORT ( a, b, clk : IN bit;
      o  : OUT bit );
END unresolve;

```

VHDL 에서 Std\_logic 과 같이 resolved 신호를 수용하는 이유는 시스템 인터페이스를 기술을 가능하게 할 수 있기 때문이다. 양방향성 시스템 공통 버스에 연결되는 경우 Tri-state 버퍼를 사용한다. 따라서 Resolved type 인 std\_logic 을 사용하면 시스템 인터페이스를 기술할 수 있다.

[예제 7] VHDL 에 의한 **Tri-State** 의 기술

```

out_bus <= in_bus1 WHEN ena_1='1' ELSE (OTHERS=>'Z');
out_bus <= in_bus2 WHEN ena_2='1' ELSE (OTHERS=>'Z');

```

이 Tri-State Bus 예에서는 두개의 enable 신호, ena\_1 과 ena\_2 에 의하여 선택되는 값이 출력 버스 out\_bus 로 할당된다. 이때 out\_bus 는 Multiple-Drive 되어 있으므로 ena\_1 과 ena\_2 가 동시에 '1'이 되지 않도록 해야 한다. Tri-State Buffer 의 Dataflow 기술은 다음과 같다.

```

u1 : PROCESS(clk)
BEGIN
  IF ena_1='1' THEN
    out_bus <= in_bus1;
  ELSE
    out_bus <= (OTHERS=>'Z');
  END IF;
END PROCESS;

u2 : PROCESS(clk)
BEGIN
  IF ena_2='1' THEN
    out_bus <= in_bus2;
  ELSE
    out_bus <= (OTHERS=>'Z');
  END IF;
END PROCESS;

```

[예제 8] 양방향 버스

데이터 버스(data bus)는 시스템 공통 버스에 양방향(Bi-Directional)으로 인터페이스 되므로 시스템 버스를 점유하지 않을 때에는 버스에 Tri-State 상태이어야 한다. 다음은 양방향 버스를 기술한 예이다.

```

ENTITY uart_top IS
PORT ( clkx16 : IN std_logic;
      read, write, reset : IN std_logic;
      rx : IN std_logic;
      tx : OUT std_logic;
      rxrdy, txrdy : OUT std_logic;
      parityerr, framingerr, overrun : OUT std_logic;
      data : INOUT std_logic_vector(7 downto 0) ); -- Bidirectional
data bus
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
    data <= rxhold WHEN read = '1' ELSE (OTHERS=>'Z');

```

```

    -- Latch data bus during write
    txhold <= data WHEN write = '1' ELSE txhold;

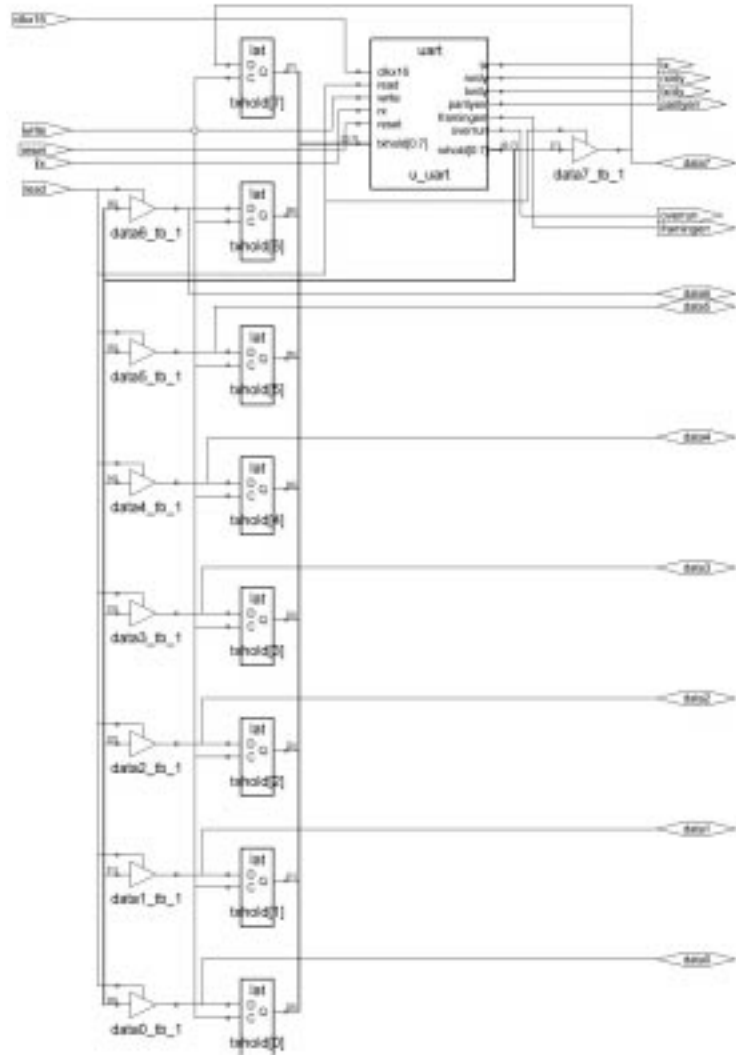
```

```

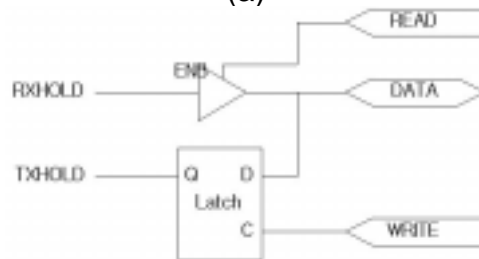
END struct;

```

양방향성(INOUT) PORT 인 “data”를 모듈 내부에서 입력 혹은 출력으로 분리하여 사용하기 위하여 두개의 내부 SIGNAL “txhold”와 “rxhold”를 선언하였다. 양방향 버스가 출력으로 작동 하는 경우 Tri-State Buffer를 이용하며 입력일 때는 래치를 포함한다. 그림 10은 [예제 8]의 양방향성 버스의 합성 결과이다. 출력의 Tri-State와 입력일 경우 래치 부분은 그림 10(b)와 같다.



(a)



(b)

그림 10. 양방향 버스의 합성 결과