Filename="ch6.doc"

6.0 Model Structure

Models consists of an Entity Declaration and Architecture Body.

6.1.1 Entity Declaration

The Entity Declaration names entity and defines interface between entity and its environment. The syntax is:

```
ENTITY entity_name IS
        [PORT clause;]
END [entity_name];
```

PORT clause indentifies ports used by entity to communicate with its environment. Its syntax is:

```
PORT (name_list: mode type;….; name_list: mode type);
```

Mode type indentifies direction of data flow through port. The allowable modes are:

- IN --flow is into entity
- OUT --flow is out of entity
- INOUT --flow may be either in or out
- BUFFER --flow is out of entity, but its value can be read internally.

Example:

```
ENTITY and_gate IS
        PORT(a, b : IN BIT; q: OUT BIT);
END and_gate;
```

6.1.2 Architecture Body

Architecture body establishes relationship between inputs and outputs of design. The syntax is:

```
ARCHITECTURE arch_name OF entity_name IS
        --declaration statements;
BEGIN
        --concurrent statements;
END [arch_name];
```

Example:

```
ARCHITECTURE arch_and_gate OF and_gate IS
BEGIN
        q <= a AND b;
END arch_and_gate;
```

6.1.3 Example of INOUT Mode Type

Synthesis of asynchronous circuit from a given state transition table.

	Q(t+1)	
Q(t) y1y2	x=0 Y1Y2	x=1 Y1Y2
00	00	01
01	01	10
10	10	11
11	11	00

Where x is the input, y1y2 is the present state, and Y1Y2 is the next state. Convert the table to a table lookup as follows:

xy1y2	Y1Y2
000	00
001	01
010	10
011	11
100	01
101	10
110	11
111	00

When implementing this if we don't distinguish the present state from the next state by using the same state variables y1y2 for both, then the state variables will both appear as input as well as output of the state machine, INOUT ports are required. The VHDL implementation is given below:

Using the "**IEEE.STD_LOGIC_SIGNED**" package

```
1.    LIBRARY IEEE;
2.    USE IEEE.STD_LOGIC_1164.ALL;
3.    USE IEEE.STD_LOGIC_SIGNED.ALL;
4.
5.    ENTITY asyn_ctr IS
6.        PORT(x:IN STD_LOGIC; y1, y2:INOUT STD_LOGIC);
7.    END asyn_ctr;
8.
9.    ARCHITECTURE arch_asyn_ctr OF asyn_ctr IS
10.       TYPE array_2d IS ARRAY(0 TO 7, 0 TO 1) OF STD_LOGIC;
11.   BEGIN
12.       PROCESS(x, y1, y2)
13.           VARIABLE mem  : array_2d := ((‘0’,’0’), (‘0’,’1’),(‘1’,’0’),(‘1’,’1’),
14.                                        (‘0’,’1’), (‘1’,’0’), (‘1’,’1’), (‘0’,’0’));
15.       BEGIN
16.           y1 <= mem(CONV_INTEGER(‘0’x&y1&y2),0);
17.           y2 <= mem(CONV_INTEGER(‘0’&x&y1&y2  ),1);
18.       END PROCESS;
19.   END arch_asyn_ctr;
```

Using the "**IEEE.STD_LOGIC_UNSIGNED**" package,

```
20.   LIBRARY IEEE;
21.   USE IEEE.STD_LOGIC_1164.ALL;
```

```
22.     USE IEEE.STD_LOGIC_UNSIGNED.ALL;
23.
24.     ENTITY asyn_ctr IS
25.             PORT(x:IN STD_LOGIC; y1, y2:INOUT STD_LOGIC);
26.     END asyn_ctr;
27.
28.     ARCHITECTURE arch_asyn_ctr OF asyn_ctr IS
29.             TYPE array_2d IS ARRAY(0 TO 7, 0 TO 1) OF STD_LOGIC;
30.     BEGIN
31.             PROCESS(x, y1, y2)
32.                     VARIABLE mem  : array_2d := ((‘0’,’0’), (‘0’,’1’),(‘1’,’0’),(‘1’,’1’),
33.                                                 (‘0’,’1’), (‘1’,’0’), (‘1’,’1’), (‘0’,’0’));
34.             BEGIN
35.                     y1 <= mem(CONV_INTEGER(x&y1&y2),0);
36.                     y2 <= mem(CONV_INTEGER(x&y1&y2  ),1);
37.             END PROCESS;
38.     END arch_asyn_ctr;
```

Using Karnaugh map to the above state transition table one can simplify the next state equations to:

Y1 = y1* NOT(y2) + y2 * (x XOR y1)
Y2 = x XOR y2

Its corresponding VHDL implementation is given below:

```
ARCHITECTURE arch2_asyn_ctr OF asyn_ctr IS
BEGIN
        y1 <=( y1 AND NOT(y2)) OR (y2 AND (x XOR y1));
        y2 <= x XOR y2;
END arch2_asyn_ctr;
```

Autologic VHDL, however requires that INOUT ports to be bi-directional ports driven by tri-state drivers. They can be driven internally or externally and must be of the kind BUS.

SR Flip Flop Example:

s	r	q	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY srff IS
```

```
        PORT(s,r : IN STD_LOGIC; q : INOUT STD_LOGIC);
END srff;

ARCHITECTURE t_srff OF srff IS
    TYPE array_1d IS ARRAY(0 TO 7) OF STD_LOGIC;
BEGIN
    PROCESS(s, r, q)
                VARIABLE x: STD_LOGIC;
                VARIABLE y: STD_LOGIC_VECTOR(3 DOWNTO 0);
                VARIABLE mem : array_1d:=('0','1','0','0','1','1','X','X');
    BEGIN
                y:='0' & s & r & q;
                x:=TO_INTEGER(y);
         q <= mem(x);
    END PROCESS;
END t_srff
```

Karnaugh map can be used to obtain an optimum hardware implementation.

rq

s	00	01	11	10
0	0	1	0	0
1	1	1	X	X

Q = s + r'q = s + (r + q')'

Figure 1 shows two possible implementations using OR gates or NAND gates.

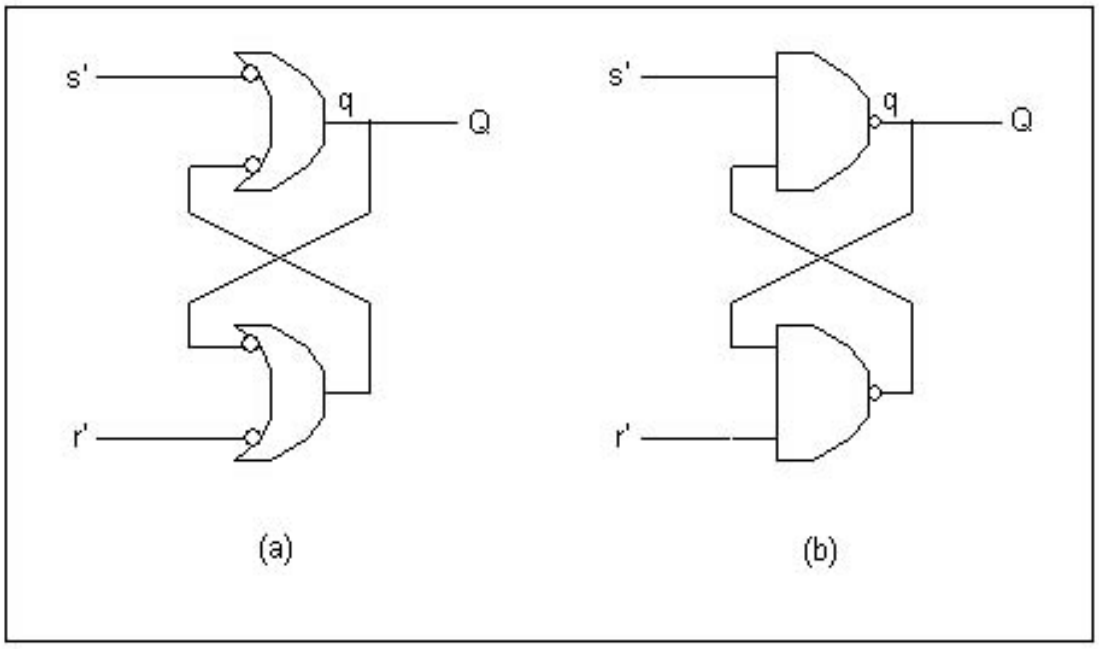

Figure 1 SR Flip Flop implementation: (a) with OR gates, (b) with NAND gates.

6.2 Example of BUFFER Mode Type

Buffer ports are simply out ports whose value can be read internally. They are used when you have values that are calculated and used internally, but are also being passed to the outside world. Instead of BUFFER port, you can achieve the same results using an OUT port with an intermediate signal.

The following VHDL codes yield the same hardware synthesis:

Implementation 1:

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY buff_port IS
5               PORT(i1, i2: IN BIT; o1: BUFFER BIT; o2: OUT BIT);
6       END buff_port;
7       ARCHITECTURE arch_buff_port OF buff_port IS
8       BEGIN
9               o1 <= NOT i1;           --o1 is being outputted
10              o2 <= i2 AND o1;        --o1 is being read
11      END arch_buff_port;
```

Implementation 2:

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY buff_port IS
5               PORT(i1, i2: IN BIT; o1: OUT BIT; o2: OUT BIT);
6       END buff_port;
7       ARCHITECTURE arch_buff_port OF buff_port IS
8               SIGNAL temp: BIT;
9       BEGIN
10              temp <= NOT i1;
11              o1 <= temp;             --o1 is being outputted
12              o2 <= i2 AND temp;
13      END arch_buff_port;
```

Both yield the following synthesized circuit:

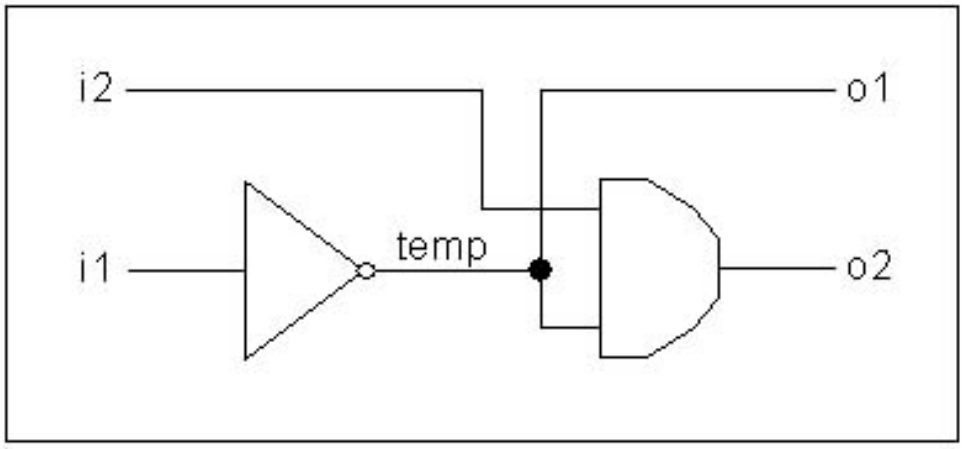

Figure 2. Hardware implementation of internal signal that is also outputted.

Another example, implementing a D-type flip flop without using Buffer data type:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dff IS
        PORT(d, clk : IN STD_LOGIC:='X';
                q : OUT STD_LOGIC:='X';
                qb : OUT STD_LOGIC:='X');
END dff;

ARCHITECTURE arch_dff OF dff IS
BEGIN
        PROCESS(clk)
        BEGIN
                IF(clk'EVENT AND clk='1')THEN
                        q <= d;
                        qb <= NOT d;
                END IF;
        END PROCESS;
END arch_dff;
```

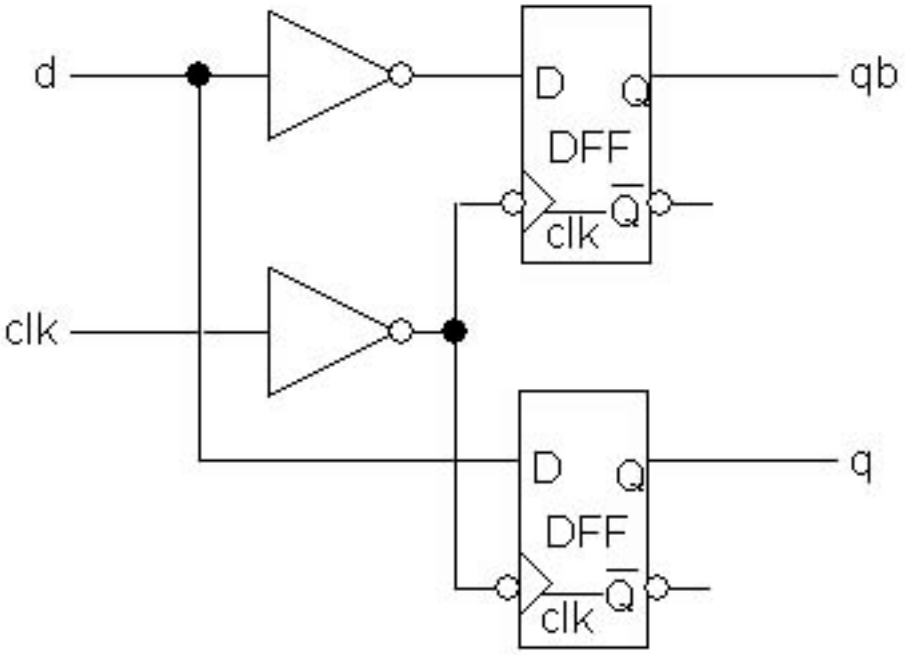

Figure 3. DFF coding that resulted in two flip flops.

Using temp signal and out ports:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

```
ENTITY dff IS
	PORT(d, clk : IN STD_LOGIC:='X';
		q : OUT STD_LOGIC:='X';
		qb : OUT STD_LOGIC:='X');
END dff;

ARCHITECTURE arch_dff OF dff IS
	SIGNAL tmp : STD_LOGIC;
BEGIN
	PROCESS(clk)
	BEGIN
		IF(clk'EVENT AND clk='1')THEN
			tmp <= d
			q <= tmp;
			qb <= NOT tmp;
		END IF;
	END PROCESS;
END arch_dff;
```

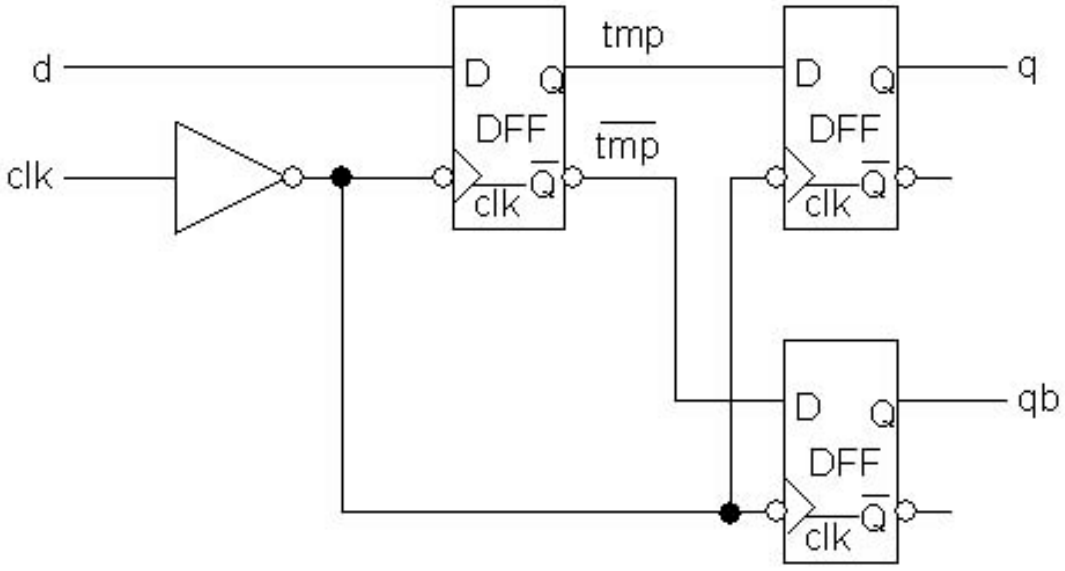

Figure 4. DFF coding that resulted in three flip flops.

Note each signal assignment is implemented by a dff.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dff IS
	PORT(d, clk : IN STD_LOGIC:='X';
		q : BUFFER STD_LOGIC:='X';
		qb : OUT STD_LOGIC:='X');
END dff;

ARCHITECTURE arch_dff OF dff IS
```

```
BEGIN
	PROCESS(clk)
	BEGIN
		IF(clk'EVENT AND clk='1')THEN
			q <= d;
			qb <= NOT q;
		END IF;
	END PROCESS;
END arch_dff;
```

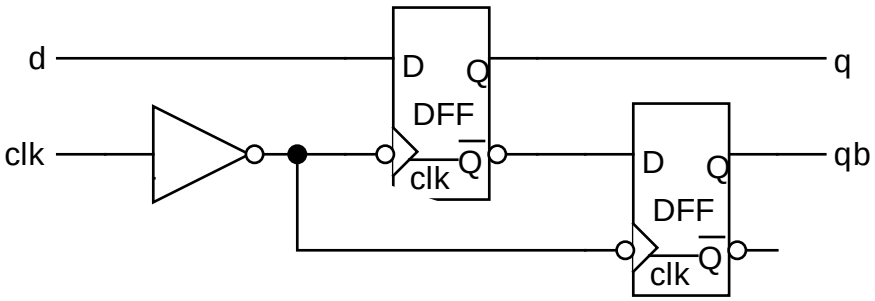

Figure 5. DFF coding that resulted in two flip flops.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dff IS
	PORT(d, clk : IN STD_LOGIC:='X';
		q : BUFFER STD_LOGIC:='X';
		qb : OUT STD_LOGIC:='X');
END dff;

ARCHITECTURE arch_dff OF dff IS
BEGIN
	PROCESS(clk)
	BEGIN
		IF(clk'EVENT AND clk='1')THEN
			q <= d;
		END IF;
	END PROCESS;
	qb <= NOT q;
END arch_dff;
```

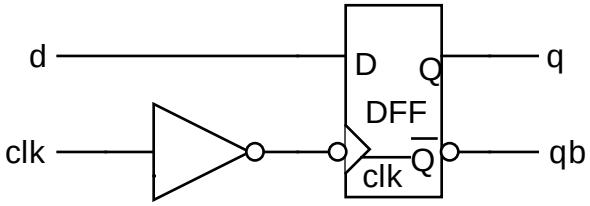

Figure 6. DFF coding that resulted in one flip flop, the desired code.

To achieve a single dff implementation the qb signal must be moved out of the process. That is its assignment is not sensitive to clk but only to changes in q. Hence it is implemented as the negation of q.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dff IS
	PORT(d, clk : IN STD_LOGIC:='X';
		q, qb : OUT STD_LOGIC:='X');
END dff;

ARCHITECTURE arch_dff OF dff IS
	SIGNAL q_temp: STD_LOGIC;
BEGIN
	PROCESS(clk)
	BEGIN
		IF(clk'EVENT AND clk='1')THEN
			q_temp <= d;
		END IF;
	END PROCESS;
	q <= q_temp;
	qb <= NOT q_temp;
END arch_dff;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dff IS
	PORT(d, clk : IN STD_LOGIC:='X';
		q, qb : OUT STD_LOGIC:='X');
END dff;

ARCHITECTURE arch_dff OF dff IS
BEGIN
	PROCESS(clk)
		VARIABLE q_temp : STD_LOGIC;
	BEGIN
		IF(clk'EVENT AND clk='1')THEN
			q_temp := d;
		END IF;
		q <= q_temp;
		qb <= NOT q_temp;

	END PROCESS;
END arch_dff;
```

These are both implemented with one dff as shown above.

6.2.1 Concurrent Statements

Concurrent statements are executed concurrently the order of their presentation is irrelevant. A model's architecture is made up of one or more concurrent statements. Each concurrent statement represents a unit of functionality. The concurrent statements are:

- BLOCK Statement
- PROCESS Statement
- ASSERTION Statement
- Signal Assignment Statement
- Procedure Call
- Component Instantiation

6.2.2 Sequential Statements

Sequential statements are executed in the order they are written. The sequential statements are:

- WAIT Statement
- Variable Assignment
- Signal Assignment*
- IF Statement
- CASE Statement
- Loops
- NEXT Statement
- EXIT Statement
- RETURN Statement
- NULL Statement
- Procedure Call*
- ASSERTION Statement*

* These statements execute sequentially or concurrently depends on its placement.

6.3 PROCESS Statement

PROCESS statement is a concurrent statement which delineates set of sequentially executed statements. Its syntax is:

```
[label:]  PROCESS[(sensitivity_list)]
            --declarative statements
          BEGIN
            --sequential statements
          END PROCESS [label]
```

Statements within a PROCESS are executed in the order they are written. Each PROCESS represents a block of logic, and all PROCESSes execute in parallel. Of course, if the simulator is running on a single processor, they actually execute in turn, but the observed effect from a simulation point of view is that they execute in parallel. In VHDL, a process is activated when a signal in its sensitivity list changes. The process sensitivity list is equivalent to WAIT ON statement. If WAIT ON statement is used in a process, **the signals used on the WAIT statement may not appear on the sensitivity list.**

Example:

```
PROCESS (a, b)
BEGIN
        q <= a AND b;
```

```
END PROCESS;
```

Is equivalent to:

```
PROCESS
BEGIN
        q <= a AND b;
        WAIT ON a, b;
END PROCESS;
```

A typical application of the process statement is to implement algorithms at an abstract level.

6.4 BLOCK Statement

A basic element of a VHDL description is the block. BLOCK statement is a concurrent statement which group a set of concurrent statements into a logical unit. Blocks can be nested to further support partitioning of model. Its syntax is:

```
block_label:    BLOCK [(guard_expression)]
                  --declarative statements
                BEGIN
                  --concurrent statements
                END BLOCK block_label;
```

Thus the architectural body itself is a block. Process is a concurrent statement, hence process can be a statement inside a block. There are two reasons for allowing nesting of blocks. First, it supports a natural form of design decomposition, and second a “guard” condition can be associated with a block. When a guard condition is TRUE, it enables certain types of statements inside the blcok.

6.5 Modeling Concurrency

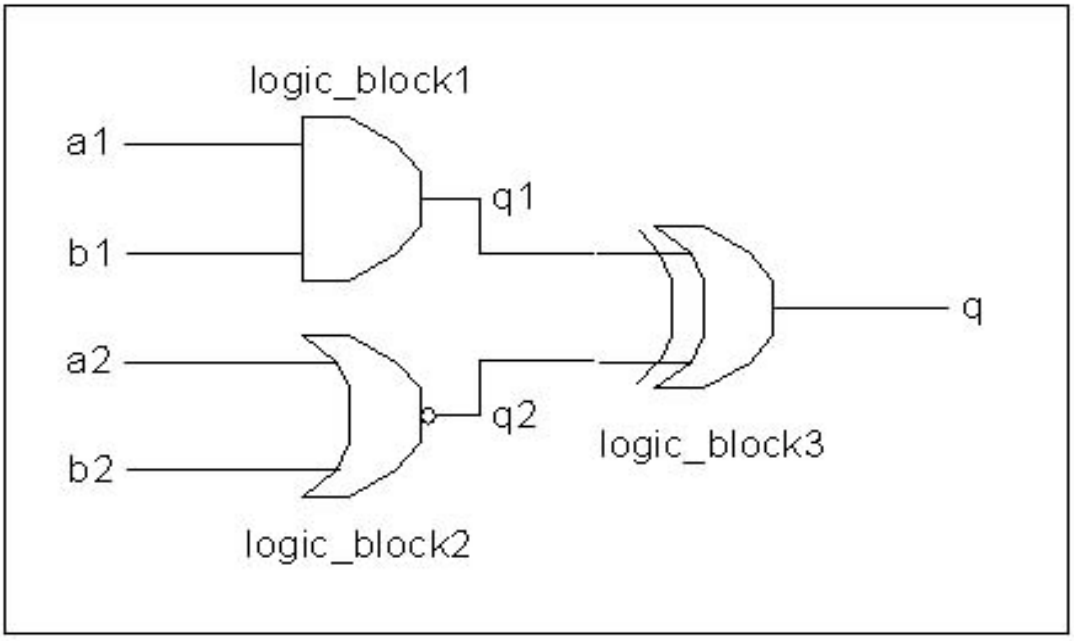

Figure 7. Circuit That Illustrate Modelling Concurrency

The modeling of logic circuits has a requirement that the model must include provision for concurrency of execution, since logic signals flow in parallel. Figure 7 illustrates this concept. Three logic blocks are shown. If one assumes that input set 1 and input set 2 are activated simultaneously, logic block 1 and 2 will be activated together. Logic block 3 will be activated as soon as either of the outputs from logic block 1 (q1) or logic block 2 (q2) change. While signals are propagating their way through block 3, new input signal changes can be propagating their way through block 1 and 2. Thus signal flow can take place through all blocks simultaneously. In VHDL, usually each process statement represents a block of logic, and all processes execute in parallel. Figure 7 is coded as follows:

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY conc_model IS
5               PORT(a1, b1, a2, b2 : IN BIT; q : OUT BIT);
6       END conc_model;
7
8       ARCHITECTURE arch_conc_model OF conc_model IS
9               SIGNAL q1, q2 : BIT;
10      BEGIN
11      logic_block1:   PROCESS(a1, b1)
12                      BEGIN
13                              q1 <= a1 AND b1;
14                      END PROCESS logic_block1;
15
16      logic_block2:   PROCESS (a2, b2)
17                      BEGIN
18                              q2 <= a2 NOR b2;
19                      END PROCESS logic_block2;
20
21      logic_block3:   PROCESS (q1, q2)
22                      BEGIN
23                              q <= q1 XOR q2;
24                      END PROCESS logic_block3;
25      END arch_conc_model;
```

Note that the sensitivity list for the process contains the input signal set for the logic block. Each process is activated when a signal in its sensitivity list changes. Since the process represents a physical system, it cannot execute the signal assignment in zero time. Without the delay specification in the signal assignment, we say that the process executes in “delta” time, a value of time that is infinitesimally small but greater than zero. Its value corresponds to one simulation cycle. Fundamentally, a simulation model consists of a set of processes. During the execution of a simulation cycle, all processes whose inputs have changed since the last cycle are evaluated. Signal outputs from those processes are scheduled to occur at later simulation times. Once all processes have been executed, the simulation cycle is complete. The next simulation cycle starts the next time a signal input to a process changes.

The above implementation is equivalent to the implementation shown below. Each process is replaced by a single signal assignment statement. This is possible because each process models a simple gate. In other words, each signal assignment can be considered as a process with the sensitivity list consisting of the signals on the right hand side of the signal assignment.

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY conc_model
5               PORT(a1, b1, a2, b2 : IN BIT; q : OUT BIT);
```

```
6       END conc_model;
7
8       ARCHITECTURE arch_conc_model OF conc_model IS
9               SIGNAL q1, q2 : BIT;
10      BEGIN
11              q1 <= a1 AND b1;
12              q2 <= a2 NOR b2;
13              q <= q1 XOR q2;
14      END arch_conc_model;
```

6.6 Assignment Statement Execution

The execution of signal assignment depends on whether it is in a concurrent statement section or within the sequential statement section inside a process. To understand this, let us look at an example.

```
PROCESS (r, s)
BEGIN
        a <= x AND y;           -- statement S1
        b <= a OR z;            -- statement S2
END PROCESS;
```

If S1 and S2 are not within a process, S1 is executed whenever x or y changes; S2 is executed whenever a or z changes. Within the process statements S1 and S2 are both executed when a change occurs on r or s. The right-hand side of these statements will use the present values of x, y, z, and a to compute the value of a and b.