

Figure 2-18(a) Behavioral Model for Figure 1-17 Using a Single Process

```

library BITLIB;
use BITLIB.Bit_pack.all;
entity SM1_2 is port(X: in bit; Z: out bit); end SM1_2;
architecture Table of SM1_2 is signal State, Nextstate: integer := 0,
begin
    process
        begin
            case State is
                when 0 =>
                    if X='0' then Z<='1'; Nextstate<=1; end if;
                    if X='1' then Z<='0'; Nextstate<=2; end if;
                when 1 =>
                    if X='0' then Z<='1'; Nextstate<=3; end if;
                    if X='1' then Z<='0'; Nextstate<=4; end if;
                when 2 =>
                    if X='0' then Z<='0'; Nextstate<=4; end if;
                    if X='1' then Z<='1'; Nextstate<=4; end if;
                when 3 =>
                    if X='0' then Z<='0'; Nextstate<=5; end if;
                    if X='1' then Z<='1'; Nextstate<=5; end if;
                when 4 =>
                    if X='0' then Z<='1'; Nextstate<=5; end if;
                    if X='1' then Z<='0'; Nextstate<=6; end if;
                when 5 =>
                    if X='0' then Z<='0'; Nextstate<=0; end if;
                    if X='1' then Z<='1'; Nextstate<=0; end if;
            end case;
        end process;
    end;

```

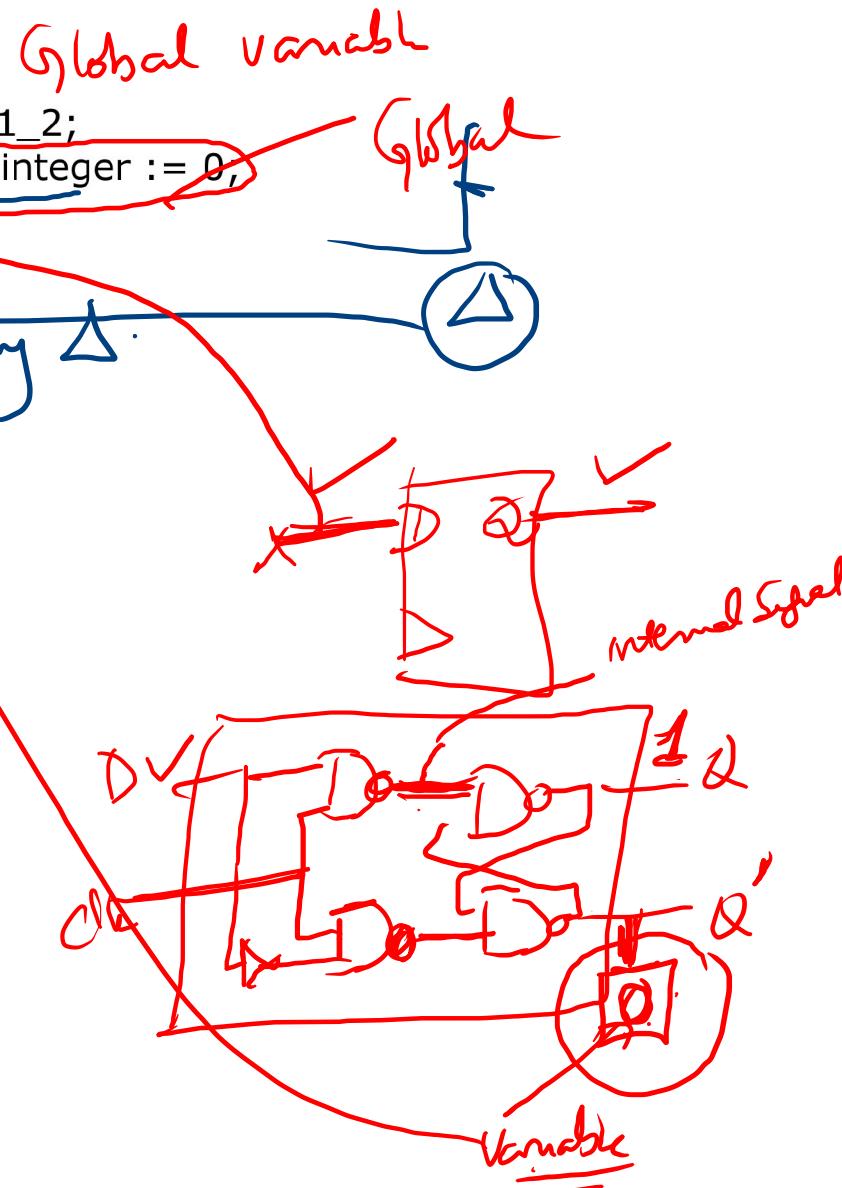


Figure 2-18(b) Behavioral Model for Figure 1-17 Using a Single Process

```

when 6 =>
  if X='0' then Z<='1'; Nextstate<=0; end if;
when others => null;           -- should not occur
end case;

```

```

process placement
  begin
    wait on CLK, X;
    if rising_edge(CLK) then
      State <= Nextstate;
      wait for 0 ns;
    end if;
    end process;
end table;

```

* Alternative:

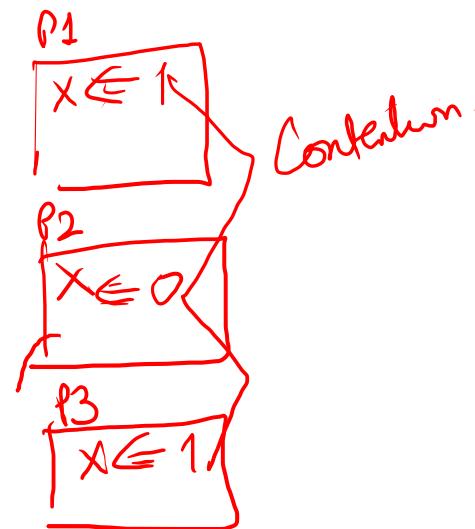
```
if CLK'event and CLK = '1' then
```

-- rising_edge function is in BITLIB *

-- wait for State to be updated

Process placement
replaces
your process.

HDL → HDL
HDL → HDL



Standard way of writing the state of behavior. X <= th2

Figure 2-19 Process Using Variables

```

entity dummy is
end dummy;

architecture var of dummy is
    signal trigger, sum: integer:=0;
begin
    process
        variable var1: integer:=1;
        variable var2: integer:=2;
        variable var3: integer:=3;
        begin
            wait on trigger;
            var1 := var2 + var3; = 5
            var2 := var1;
            var3 := var2;
            sum <= var1 + var2 + var3;
        end process;
    end var;

```

Shared Variable !!

Concurrent

Initial State

$$\begin{aligned}
 \text{var1} &= 2 + 3 = 5 \\
 \text{var2} &= 5 \\
 \text{var3} &= 5 \\
 \text{sum} &= 5 + 5 + 5 = 15 \text{ (after } \Delta \text{)}
 \end{aligned}$$

Within the architecture

- Signals (declaration)
Global

Within the process

- Variable (local).

Figure 2-20 Process Using Signals

```

entity dummy is
end dummy;

architecture sig of dummy is
    signal trigger, sum: integer:=0;
    signal sig1: integer:=1;
    signal sig2: integer:=2;
    signal sig3: integer:=3;
begin
    process
        begin
            wait on trigger;
            sig1 <= sig2 + sig3;
            sig2 <= sig1;
            sig3 <= sig2;
            sum <= sig1 + sig2 + sig3;
        end process;
    end sig;

```

$\text{var1} := \text{var2} + \text{var3}$
 $\text{var2} := \text{var1}$
 \vdots



$\checkmark \text{sig1} = 2 + 3 = 5$
 $\text{sig2} = 1$
 $\text{sig3} = 2$
 $\underline{\text{sum} = 1 + 2 + 3 = 6}$

Infinite number of small delay
 $(\text{after } \Delta) 1$
 $(\text{after } \Delta) 2\Delta$
 $(\text{after } \Delta) 3\Delta$
 $(\text{after } \Delta) 4\Delta$
 PS

From Page 67

Predefined VHDL types include:

bit	'0' or '1'
boolean	FALSE or TRUE
integer	an integer in the range $-(2^{31}-1)$ to $+(2^{31}-1)$ (some implementations support a wider range)
real	floating-point number in the range -1.0E38 to +1.0E38
character	any legal VHDL character including upper- and lower-case letters, digits, and special characters; each printable character must be enclosed in single quotes; e.g., 'd','7','+'
time	an integer with units fs, ps, ns, us, ms, sec, min, or hr

Note that the integer range for VHDL is symmetrical even though the range for a 32-bit 2's complement integer is -2^{31} to $+(2^{31} - 1)$.

Example of enumeration type (user-defined):

```
type state_type is (S0, S1, S2, S3, S4, S5);  
signal state : state_type := S1;
```

From Page 68

Example of array type:

```
type SHORT_WORD is array (15 downto 0) of bit;
```

Examples of array objects of type SHORT_WORD

```
signal DATA_WORD: SHORT_WORD;  
variable ALT_WORD: SHORT_WORD := "0101010101010101";  
constant ONE_WORD: SHORT_WORD := (others => '1');
```

General forms of the array type and array object declarations:

```
type array_type_name is array index_range of element_type;  
signal array_name: array_type_name [ := initial_values ];
```

(signal may be replaced with variable or constant)

Two-dimensional array example:

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;  
variable matrixA: matrix4x3 := ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));
```

The variable matrixA, will be initialized to:

1	2	3
4	5	6
7	8	9
10	11	12

From Page 69 – 70

Example of an unconstrained array type:

```
type intvec is array (natural range <>) of integer;  
signal intvec5: intvec(1 to 5) := (3,2,6,8,1);
```

Two-dimensional array type with unconstrained row and column index ranges:

```
type matrix is array (natural range <>, natural range <>) of integer;
```

Predefined unconstrained array types in VHDL include bit-vector and string:

```
type bit_vector is array (natural range <>) of bit;  
type string is array (positive range <>) of character;
```

The following example declares a constant string1 of type string:

```
constant string1: string(1 to 29) := "This string is 29 characters."
```

A bit_vector literal may be written either as a list of bits separated by commas or as a string. For example, ('1','0','1','1','0') and "10110" are equivalent forms.

```
constant A : bit_vector(0 to 5) := "101011";
```

A subtype specifies a subset of the values specified by a type. Example:

```
subtype SHORT_WORD is bit_vector (15 downto 0);
```

Predefined subtypes of type integer: POSITIVE (all positive integers)
NATURAL (all positive integers and 0)

Figure 2-21 Sequential Machine Model Using State Table

```
entity SM1_2 is
    port (X, CLK: in bit;
          Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
    type StateTable is array (integer range <>, bit range <>) of integer;
    type OutTable is array (integer range <>, bit range <>) of bit;
    signal State, NextState: integer := 0;
    constant ST: StateTable (0 to 6, '0' to '1') :=
        ((1,2), (3,4), (4,4), (5,5), (5,6), (0,0), (0,0));
    constant OT: OutTable (0 to 6, '0' to '1') :=
        (('1','0'), ('1','0'), ('0','1'), ('0','1'), ('1','0'), ('0','1'), ('1','0'));
begin
    NextState <= ST(State,X);           -- concurrent statements
    Z <= OT(State, X);                -- read next state from state table
                                         -- read output from output table
    process(CLK)
    begin
        if CLK = '1' then            -- rising edge of CLK
            State <= NextState;
        end if;
    end process;
end Table;
```

2.8 VHDL Operators

Predefined VHDL operators can be grouped into seven classes:

1. binary logical operators: **and** **or** **nand** **nor** **xor** **xnor**
(lowest precedence -- applied last)

ea. notq & le gt ge

2. relational operators: **=** **/=** **<** **<=** **>** **>=**

3. shift operators: **sll** **srl** **sla** **sra** **rol** **ror**

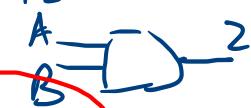
4. adding operators: **+** **-** **&** (concatenation)

5. unary sign operators: **+**

6. multiplying operators: ***** **/** **mod** **rem**

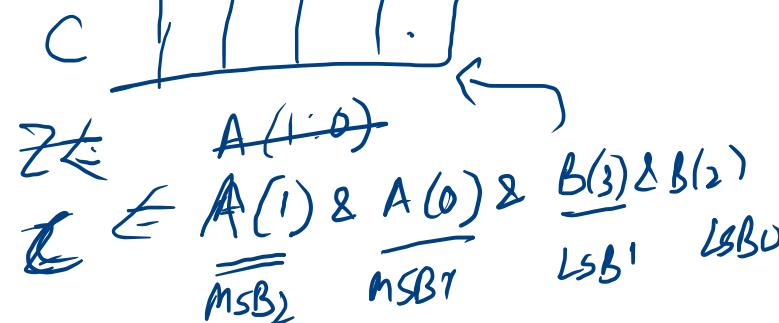
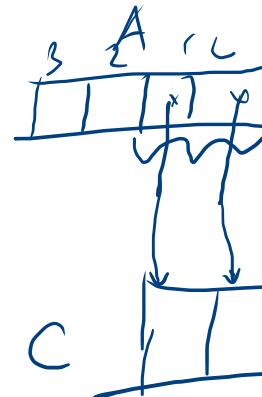
7. miscellaneous operators: **not** **abs** ******
(highest precedence -- applied first)

BODMAS



A **nand** B

A **and** B



Example of VHDL Operators

In the following expression, A, B, C, and D are bit_vectors:

$$(A \& \text{not } B \text{ or } C \text{ ror } 2 \text{ and } D) = "110010"$$

The operators would be applied in the order:

not, &, ror, or, and, =

If $A = "110"$, $B = "111"$, $C = "011000"$, and $D = "111011"$, the computation would proceed as follows:

not $B = "000"$ (bit-by-bit complement)

$A \& \text{not } B = "110000"$ (concatenation)

$C \text{ ror } 2 = "000110"$ (rotate right 2 places)

$(A \& \text{not } B) \text{ or } (C \text{ ror } 2) = "110110"$ (bit-by-bit or)

$(A \& \text{not } B) \text{ or } (C \text{ ror } 2) \text{ and } D = "110010"$ (bit-by-bit and)

$[(A \& \text{not } B) \text{ or } (C \text{ ror } 2) \text{ and } D] = "110010" = \text{TRUE}$

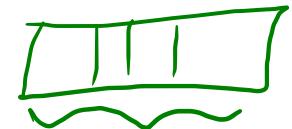
(the parentheses force the equality test to be done last and the result is TRUE)

$$110 \& 000 = \underline{110\ 000} \text{ or } 000110 = \underline{110\ 110}$$

ror = Shift registers.
ro(=

$$\begin{aligned} A(2) &= 1 \\ A(1) &= 0 \\ A(0) &= 1 \\ B(2) &= 1 \\ B(1) &= 1 \\ B(0) &= 1 \end{aligned}$$

$$\begin{aligned} B(2) &\rightarrow \\ B(1) &\rightarrow \\ B(0) &\rightarrow \end{aligned}$$



Example of Shift Operators

The shift operators can be applied to any bit_vector or boolean_vector. In the following examples, A is a bit_vector equal to 10010101: 8bit

- ← A **sll** 2 is "01010100" (shift left logical, filled with '0')
- A **srl** 3 is "00010010" (shift right logical, filled with '0')
- ← A **sla** 3 is '10101111' (shift left arithmetic, filled with right bit)
- A **sra** 2 is "11100101" (shift right arithmetic, filled with left bit)
- A **rol** 3 is "10101100" (rotate left)
- A **ror** 5 is "101011100" (rotate right)

2.9 VHDL Functions

A function executes a sequential algorithm and returns a single value to the calling program. When the following function is called, it returns a bit vector equal to the input bit vector (reg) rotated one position to the right:

```
function rotate_right (reg: bit_vector)
    return bit_vector is
begin
    return reg ror 1;
end rotate_right;
```

A function call can be used anywhere that an expression can be used. For example, if A = "10010101", the statement

```
B <= rotate_right(A);
```

sets B equal to "11001010", and leaves A unchanged.

General form of function declaration:

```
function function-name (formal-parameter-list)
    return return-type is
    [declarations]
begin
    sequential statements -- must include return return-value;
end function-name;
```

General form of function call:

```
function_name (actual-parameter-list)
```

From Page 72

General form of a for loop:

```
[loop-label:] for loop-index in range loop
    sequential statements
end loop [loop-label];
```

Exit statement has the form:

```
exit;           -- or
exit when condition;
```

For Loop Example:

```
-- compare two 8-character strings and return TRUE if equal
function comp_string(string1, string2: string(1 to 8))
    return boolean is

variable B: boolean;
begin
    loopex: for j in 1 to 8 loop
        B := string1(j) = string2(j);
        exit when B=FALSE;
    end loop loopex;
    return B;
end comp_string;
```

Note: The loop index (j) is automatically declared; it must not be declared in the program.

Figure 2-22 Add Function

-- This function adds 2 4-bit vectors and a carry.

-- It returns a 5-bit sum

```
function add4 (A,B: bit_vector(3 downto 0); carry: bit)
    return bit_vector is

variable cout: bit;
variable cin: bit := carry;
variable Sum: bit_vector(4 downto 0):="00000";
begin
loop1: for i in 0 to 3 loop
    cout := (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin);
    Sum(i) := A(i) xor B(i) xor cin;
    cin := cout;
end loop loop1;
Sum(4):= cout;
return Sum;
end add4;
```

Example function call:

```
Sum1 <= add4(A1, B1, cin);
```

Figure 2-23 Procedure for Adding Bit_vectors

```
-- This procedure adds two n-bit bit_vectors and a carry and  
-- returns an n-bit sum and a carry. Add1 and Add2 are assumed  
-- to be of the same length and dimensioned n-1 downto 0.
```

```
procedure Addvec  
  (Add1,Add2: in bit_vector;  
   Cin: in bit;  
   signal Sum: out bit_vector;  
   signal Cout: out bit;  
   n:in positive) is  
   variable C: bit;  
begin  
  C := Cin;  
  for i in 0 to n-1 loop  
    Sum(i) <= Add1(i) xor Add2(i) xor C;  
    C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);  
  end loop;  
  Cout <= C;  
end Addvec;
```

Example procedure call:

```
Addvec(A1, B1, Cin, Sum1, Cout, 4);
```

Table 2-1 Parameters for Subprogram Calls

Mode	Class	Actual Parameter	
		Procedure Call	Function Call
in ¹	constant ²	expression	expression
	signal variable	signal variable	n/a
out/inout	signal	signal	n/a
	variable ³	variable	n/a

¹ default mode for functions ² default for in mode

³ default for out/inout mode

From Page 76

General form of Package declaration:

```
package package-name is
    package declarations
end [package][package-name];
```

General form of Package body:

```
package body package-name is
    package body declarations
end [package body][package name];
```

Packages and associated components can be placed in a library to allow easy access.

Library BITLIB (see Appendix B) contains functions and components that use signals of type bit.

To access components and functions within BITLIB, use the following statements:

```
library BITLIB;
use BITLIB.bit_pack.all;
```

Appendix B Bit Package (i)

-- Bit package for Digital Systems Design Using VHDL

```
package bit_pack is
    function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
        return bit_vector;
    function falling_edge(signal clock:bit)
        return Boolean ;
    function rising_edge(signal clock:bit)
        return Boolean ;
    function vec2int(vec1: bit_vector)
        return integer;
    function int2vec(int1,NBits: integer)
        return bit_vector;
    procedure Addvec
        (Add1,Add2: in bit_vector;
         Cin: in bit;
         signal Sum: out bit_vector;
         signal Cout: out bit;
         n: in natural);

    component jkff
        generic(DELAY:time := 10 ns);
        port(SN, RN, J,K,CLK: in bit; Q, QN: inout bit);
    end component;

    component dff
        generic(DELAY:time := 10 ns);
        port (D, CLK: in bit; Q: out bit; QN: out bit := '1');
    end component;
```

Appendix B Bit Package (ii)

component and2

```
generic(Delay:time := 10 ns);
port(A1, A2: in bit; Z: out bit);
end component;
```

component and3

```
generic(Delay:time := 10 ns);
port(A1, A2, A3: in bit; Z: out bit);
end component;
```

component and4

```
generic(Delay:time := 10 ns);
port(A1, A2, A3, A4: in bit; Z: out bit);
end component;
```

component or2

```
generic(Delay:time := 10 ns);
port(A1, A2: in bit; Z: out bit);
end component;
```

component or3

```
generic(Delay:time := 10 ns);
port(A1, A2, A3: in bit; Z: out bit);
end component;
```

...

(other component declarations go here)

...

end bit_pack;

Appendix B Bit Package (iii)

```
package body bit_pack is
-- This function adds 2 4-bit numbers, returns a 5-bit sum
function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
    return bit_vector is
variable cout: bit:='0';
variable cin: bit:=carry;
variable retval: bit_vector(4 downto 0):="00000";
begin
lp1: for i in 0 to 3 loop
    cout :=(reg1(i) and reg2(i)) or ( reg1(i) and cin) or
        (reg2(i) and cin );
    retval(i) := reg1(i) xor reg2(i) xor cin;
    cin := cout;
end loop lp1;
retval(4):=cout;
return retval;
end add4;

-- Function for falling edge
function falling_edge(signal clock:bit)
    return Boolean is
begin
    return clock'event and clock = '0';
end falling_edge;

-- other functions and procedure declarations go here

end bit_pack
```

Appendix B Bit Package (iv)

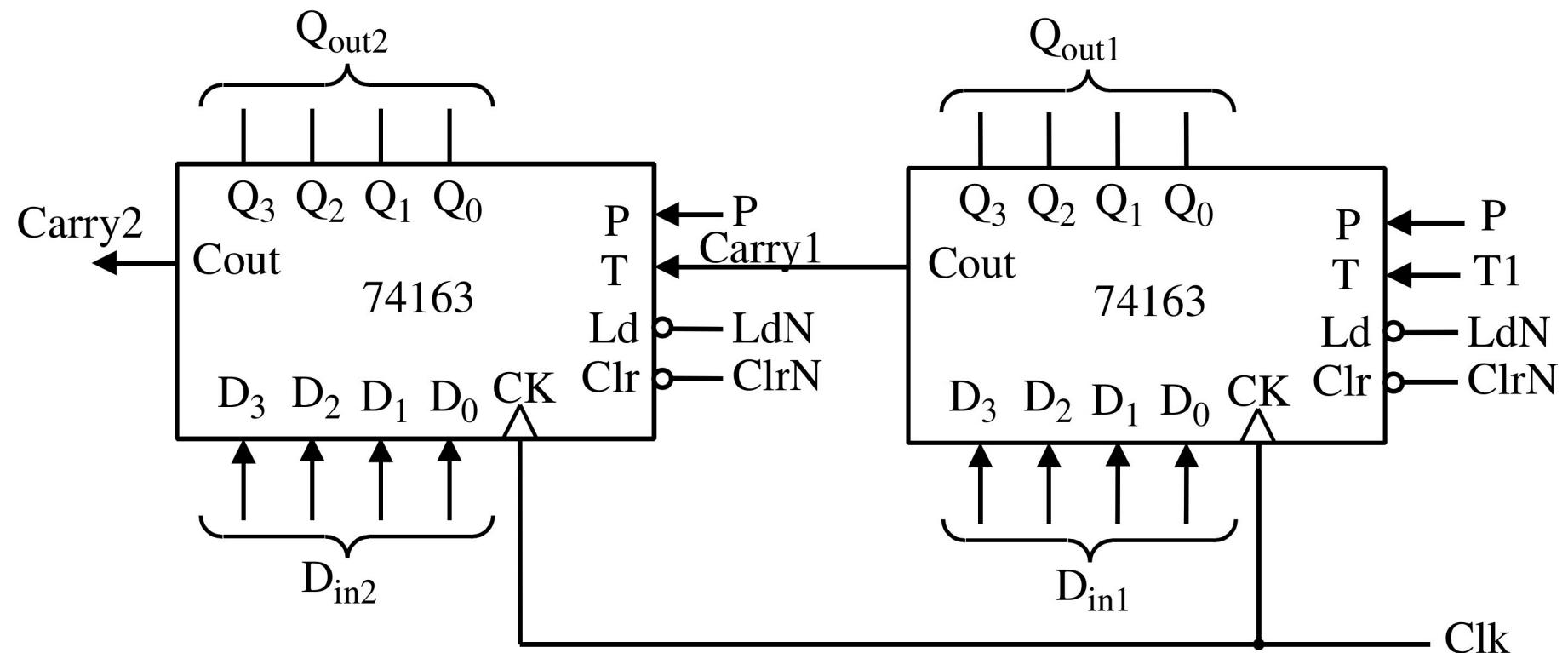
Components in Library BITLIB include:

-- 3 input AND gate

```
entity And3 is
  generic(DELAY:time);
  port (A1,A2, A3: in bit; Z: out bit);
end And3;
architecture concur of And3 is
begin
  Z <= A1 and A2 and A3 after DELAY;
end;

-- D Flip-flop
entity DFF is
  generic(DELAY:time);
  port (D, CLK: in bit;
        Q: out bit; QN: out bit := '1');
  -- initialize QN to '1' since bit signals are initialized to '0' by default
end DFF;
architecture SIMPLE of DFF is
begin
  process(CLK)
  begin
    if CLK = '1' then --rising edge of clock
      Q <= D after DELAY;
      QN <= not D after DELAY;
    end if;
  end process;
end SIMPLE;
```

Figure 2-24 Two 74163 Counters Cascaded to Form an 8-bit Counter



Control Signals			Next State				
ClrN	LdN	P•T	Q3 ⁺	Q2 ⁺	Q1 ⁺	Q0 ⁺	
0	X	X	0	0	0	0	(clear)
1	0	X	D3	D2	D1	D0	(parallel load)
1	1	0	Q3	Q2	Q1	Q0	(no change)
1	1	1	present state + 1				(increment count)

Figure 2-25 74163 Counter Model

```
-- 74163 FULLY SYNCHRONOUS COUNTER

library BITLIB;                                -- contains int2vec and vec2int functions
use BITLIB.bit_pack.all;

entity c74163 is
    port(LdN, ClrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
          Cout: out bit; Q: inout bit_vector(3 downto 0) );
end c74163;

architecture b74163 of c74163 is
begin
    Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
    process
        begin
            wait until CK = '1';           -- change state on rising edge
            if ClrN = '0' then Q <= "0000";
            elsif LdN = '0' then Q <= D;
            elsif (P and T) = '1' then
                Q <= int2vec(vec2int(Q)+1,4);
            end if;
        end process;
    end b74163;
```

Figure 2-26 VHDL for 8-bit Counter

```
library BITLIB;
use BITLIB.bit_pack.all;

entity c74163test is
    port(ClrN,LdN,P,T1,Clk: in bit;
          Din1, Din2: in bit_vector(3 downto 0);
          Qout1, Qout2: inout bit_vector(3 downto 0);
          Carry2: out bit);
end c74163test;

architecture tester of c74163test is
    component c74163
        port(LdN, ClrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
             Cout: out bit; Q: inout bit_vector(3 downto 0) );
    end component;
    signal Carry1: bit;
    signal Count: integer;
    signal temp: bit_vector(7 downto 0);
begin
    ct1: c74163 port map (LdN,ClrN,P,T1,Clk,Din1,Carry1,Qout1);
    ct2: c74163 port map (LdN,ClrN,P,Carry1,Clk,Din2,Carry2,Qout2);
    temp <= Qout2 & Qout1;
    Count <= vec2int(temp);
end tester;
```