**To change in vhdl**

**Half adder programs**

library ieee;

use ieee.std\_logic\_1164.all;

entity halfadder is

port(a,b:in std\_logic;

sum,carry:out std\_logic);

end halfadder;

architecture halfadder\_arch of halfadder is

begin

sum<=a xor b;

carry<=a and b; **removed braces in this expression pg 660**

end halfadder\_arch;

**HALF subtractor**

library ieee;

use ieee.std\_logic\_1164.all;

entity halfsubtractor is

port(a,b:in std\_logic;

difference,borrow:out std\_logic);

end halfsubtractor;

architecture halfsubtractor\_arch of halfsubtractor is

begin

difference<=a xor b;

borrow<=a and b; **changed borrow spelling instead of barrow pg 660**

end halfsubtractor \_arch;

**pg.661**

***Four Bit Parallel Binary Adder:*** The VHDL program for the *Four Bit Parallel Binary Adder* shown in Fig. 5.10 can be written as follows. This program follows *structural approach*.

library ieee;

use ieee.std\_logic\_1164.all;

entity adder\_4bit is

port (A, B:in std\_logic\_vector(3 downto 0);

S:out std\_logic\_vector(3 downto 0);

Cout:out std\_logic);

end adder\_4bit;

architecture adder\_4bit\_arch of adder\_4bit is

component fulladder is – *Full Adder program is used as*– *a component.*

port (A, B, C:in std\_logic;

Sum, Carry:out std\_logic);

end component;

signal Cin,C1,C2,C3:std\_logic; – *Holds current value and a set* ***removed c4***

begin – *of possible future values.*

Cin<= ‘0’;

a1:fulladder port map (A(0),B(0),Cin,S(0),C1); *– Calling of full adder program*

a2:fulladder port map (A(1),B(1),C1,S(1),C2); – *first three bits i/p;* – *last 2bit o/p*

a3:fulladder port map(A(2),B(2),C2,S(2),C3); – *carry of map is passed as i/p*

a4:fulladder port map(A(3),B(3),C3,S(3),Cout); – *to the next map* ***C4 is changed as Cout***

end;

**16.9.3 Multiplexer & Demultiplexer no change**

***4-to-1 Multiplexer*** : The VHDL program for the *4-to-1 Multiplexer* shown in Fig. 6.2 can be written as follows. This program follows *data flow approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity mux is

port (S1,S0,D0,D1,D2,D3:in std\_logic;

Y:out std\_logic);

end mux;

architecture arch\_mux of mux is

begin Y<=((not S1)and(not S0)and D0)or – *Any one of D0, D1, D2 or D3*

((not S1) and S0 and D1)or – *is taken as output, according to*

(S1 and(not S0)and D2)or – *values of S1 and S2.*

(S1 and S0 and D3);

end arch\_mux;

***1-to-4 Demultiplexer:*** The VHDL program for the *1-to-4 Demultiplexer* shown in Fig. 6.7 can be written as follows. This program follows *data flow approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity demux is

port(D,S1,S0:in std\_logic;

Y0,Y1,Y2,Y3:out std\_logic);

end demux;

architecture demux\_arch of demux is

begin

Y0<=D and (not S1) and (not S0); – *The i/p is given to any one of*

Y1<=D and (not S1) and (S0); – *the o/p, which is selected by*

Y2<=D and (S1) and (not S0); – *the values of S0 and S1*

Y3<=D and (S1) and (S0);

end;

**16.9.4 Encoder and Decoder**

***Octal-to-Binary Encoder:*** The VHDL program for the *Octal-to-Binary Encoder* shown in Fig. 6.26 can be written as follows. This program follows *data flow approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity octal\_binary**\_**encoder is

port(D:in std\_logic\_vector(7 downto 0);

Y:out std\_logic\_vector(7 downto 0));

end octal\_binary\_encoder;

architecture encoder\_arch **of** octal\_binary\_encoder is

begin

Y(0)<= D(1) or D(3) or D(5) or D(7);

Y(1)<= D(2) or D(3) or D(6) or D(7);

Y(2)<= D(4) or D(5) or D(6) or D(7);

end encoder\_arch;

***3-to-8 Decoder:*** The VHDL program for the *3-to-8 Decoder* shown in Fig. 6.12 can be written as follows. This program follows *data flow approach.* ***No change***

library ieee;

use ieee.std\_logic\_1164.all;

entity decoder is

port (A,B,C: in std\_logic;

D: out std\_logic\_vector(7 downto 0));

end decoder;

architecture decoder\_arch of decoder is

begin

D(0)<=(not A) and (not B) and (not C);

D(1)<=(not A) and (not B) and C;

D(2)<=(not A) and B and (not C);

D(3)<=(not A) and B and C;

D(4)<=A and (not B) and (not C);

D(5)<=A and (not B) and C;

D(6)<=A and B and (not C);

D(7)<=A and B and C;

end;

**16.9.5 4-Bit Parity Checker NO CHANGE**

The VHDL program for the *4-bit Parity Checker* shown in Fig. 6.30 (a) can be written as follows. This program follows *data flow approach.* Here, the output is 1 when the number of 1’s in the inputs *W, X, Y* and *Z* is odd and 0 when the number of 1’s in the inputs is even.

library ieee;

use ieee.std\_logic\_1164.all;

entity parity\_even is

port(W,X,Y,Z:in std\_logic;

Output:out std\_logic);

end parity\_even;

architecture arch\_parity\_even of parity\_even is

begin

output<= ((W xor X) xor Y) xor Z;

end;

16.9.6 4-bit Magnitude Comparator

The VHDL program for the *4-bit Magnitude Comparator* shown in Fig.6.48 can be written as follows. This program follows *data flow approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity comparator is

port(A:in std\_logic\_vector(3 downto 0);

B:in std\_logic\_vector(3 downto 0);

E,GT,LT:out std\_logic);

end comparator;

architecture arch\_comparator of comparator is

signal E3,E2,E1,E0,GT3,GT2,GT1,GT0, LT3, LT2,LT1,LT0:std\_logic;

begin

E3<=A(3) xnor B(3);

E2<=A(2) xnor B(2);

E1<=A(1) xnor B(1);

E0<=A(0) xnor B(0);

E <= E3 and E2 and E1 and E0;

GT3<=A(3) and (not B(3));

GT2<= E3 and A(2) and (not B(2));

GT1<= E3 and E2 and A(1) and (not B(1));

GT0<= E3 and E2 and E1 and A(0) and (not B(0));

GT<= GT3 or GT2 or GT1 or GT0;

LT3<=(not A(3)) and B(3);

LT2<= E3 and (not A(2)) and B(2);

LT1<= E3 and E2 and (not A(1)) and B(1);

LT0<= E3 and E2 and E1 and (not A(0)) and B(0);

LT<= LT3 or LT2 or LT1 or LT0;

end;

**16.9.7 Code Converters**

***4-bit Binary-to-Gray Code Converter:*** The VHDL program for the *4-bit Binary-*to- *Gray Code Converter*, shown in Fig. 6.41 can be written as follows. This program follows data flow *approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity Binary\_to\_gray is

port (B:in std\_logic\_vector (3 downto 0);

G:out std\_logic\_vector (3 downto 0));

end Binary\_to\_gray;

architecture arch\_Binary\_to\_gray of Binary\_to\_gray is

begin

G(3)<= B(3);

G(2)<= B(3) xor B(2);

G(1)<= B(2) xor B(1);

G(0)<= B(1) xor B(0);

end;

***4-bit Gray Code-to-Binary Converter:*** The VHDL program for the *4-bit Gray*

*code-to-Binary Converter* shown in Fig. 6.44 can be written as follows. This program

follows *data flow approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity Gray\_to\_Binary is

port(G:in std\_logic\_vector(3 downto 0);

B:inout std\_logic\_vector(3 downto 0)); **inout**

end Gray\_to\_Binary;

architecture arch\_Gray\_to\_Binary of Gray\_to\_Binary is

begin

B(3)<= G(3);

B(2)<= G(3) xor G(2);

B(1)<= B(2) xor G(1);

B(0)<= B(1) xor G(0);

end;

**16.10 HDL FOR SEQUENTIAL LOGIC CIRCUITS**

**16.10.1 Realization of Flip-Flops**

***SR-Flip-Flop:*** The VHDL program for the *SR-Flip-Flop* shown in Fig. 7.4 (b) can be written as follows. This program follows *data flow approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity srff1 is – *Declaration of entity.*

port(S,R:in std\_logic;Q,

NQ:inout std\_logic); – *Set of i/p & o/p in declaration*

end srff1; – *End of entity declaration*

architecture srff\_arch of srff1 is

begin

Q<= R nor NQ;

NQ<=S nor Q;

end ;

***Clocked SR-Flip-Flop:*** The VHDL program for the *Clocked SR-Flip-Flop* shown in Fig.7.10 can be written as follows. This program follows *structural approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity clksr is

port (S,R,CLK : in std\_logic;

M,N : inout std\_logic;

Q, NQ:inout std\_logic);

end clksr;

architecture clksr\_arch of clksr is

component srff1 is

port(S,R : in std\_logic; – *SR FF program is used as component*

Q,NQ:inout std\_logic);

end component;

begin

M <= S and Clk; – *AND logic operation S with CLK*

N <= R and CLK; – *AND logic operation R with CLK*

a1: srff1 port map (M, N, Q, NQ); – *SR FF component is used at*

end; – *instance a1*

***D-Flip-Flop:*** The VHDL program for the *D-Flip-Flop* can be written as follows. This program follows *behavioral approach* as per truth table given in Table 7.6.

library ieee;

use ieee.std\_logic\_1164.all;

entity dfff1 is

port(D,CLK,reset:in std\_logic;

Q:out std\_logic);

end dfff1;

architecture arch\_dflipflop of dfff1 is

begin

process (CLK)

begin

if reset= '0' then

Q<='0';

elsif CLK='1' and CLK 'event then

Q<=D;

end if;

end process;

end;

***JK-Flip-Flop:*** The VHDL program for the *JK-Flip-Flop* can be written as follows. This program follows *behavioral approach* as per truth table given in Table 7.9.

library ieee;

use ieee.std\_logic\_1164.all;

entity jkff1 is

port (J,K,CLK,reset: in std\_logic;

Q, NQ: inout std\_logic);

end jkff1;

architecture jkff\_arch of jkff1 is

begin

process(CLK,J,K)

begin

if reset= '0' then

Q<='0';

NQ<='0';

elsif (CLK='1' and CLK'event)then

if (J='0' and K='0') then

Q<=Q;

NQ<=NQ;

elsif (J= '1' and K='0')then

Q<='1';

NQ<='0';

elsif(J='0' and K='1')then

Q<='0';

NQ<='1';

elsif(J='1' and K='1')then

Q<=not Q;

NQ<=not NQ;

end if;

end if;

end process;

end;

***T-Flip-Flop:*** The VHDL program for the *T-Flip-Flop* can be written as follows. This program follows *behavioral approach* as per truth table given in Table 7.12.

library ieee;

use ieee.std\_logic\_1164.all;

entity tff1 is

port (T,CLK,reset:in std\_logic;

Q, NQ:inout std\_logic);

end tff1;

architecture arch\_tff of tff1 is

begin

process (CLK,T)

begin

if reset= '0' then

Q<='0';

NQ<='0';

if (CLK='1' and CLK'event) then

if(T='1') then

Q<= not Q;

NQ <= not (Q) after 0.5 ns;

else

Q <= Q;

NQ <= not (Q) after 0.5 ns;

end if;

end if;

end if;

end process;

end;

**16.10.2 Realization of Shift Registers**

***4-bit Serial in Serial out Shift Register:*** The VHDL program for the *4-bit Serial in Serial out Shift Register* shown in Fig.9.4 (a) can be written as follows. This program follows *structural approach.* When clock pulse is applied, output of one flip-flop is given as input of next flip-flop. The serial output is taken from the last flip-flop.

library ieee;

use ieee.std\_logic\_1164.all;

entity siso is

port(D:in std\_logic;

reset,CLK:in std\_logic;

Q:out std\_logic);

end siso;

architecture arch\_siso of siso is

signal QA,QB,QC,QD:std\_logic;

component dfff1 is

port(D,CLK,reset:in std\_logic; – *D Flip-Flop program is used as*

Q:out std\_logic); – *Component*

end component;

begin

a1:dfff1 port map (D,CLK,reset,QA);

a2:dfff1 port map (QA,CLK,reset,QB);

a3:dfff1 port map (QB,CLK,reset,QC);

a4:dfff1 port map (QC,CLK,reset,QD);

end;

***4-bit Serial in Parallel out Shift Register:*** The VHDL program for the *4-bit Serial in Parallel out Shift Register* can be written as follows. This program follows *behavioural approach.* When clock pulse is applied, output of one flip-flop is given as input of the next one and the output is obtained from all the flip-flops.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity shiftreg is

port(CLK,reset,d,enable:in std\_logic;

shiftedop: out std\_logic\_vector(3 downto 0));

end shiftreg;

architecture arch\_shiftreg of shiftreg is

signal a:std\_logic\_vector(3 downto 0);

begin

process(CLK,reset)

begin

if reset='0' then

a<=(others=>'0');

elsif CLK'event and CLK='1' then

if enable='1' then

a<=shl(a,"1");

a(0)<=d;

end if;

end if;

end process;

shiftedop<=a;

end;

***4-bit Parallel in Serial out Shift Register:*** The VHDL program for the *4-bit Parallel in Serial out Shift Register* using *behavioral approach* can be written as follows.

library ieee;

library ieee;

use ieee.std\_logic\_1164.all;

entity dpiso is

port(CLK,load: in std\_logic;

d: in std\_logic\_vector (3 downto 0);

dout: out std\_logic);

end dpiso;

architecture arch\_dpiso of dpiso is

signal reg: std\_logic\_vector (3 downto 0);

begin

process (CLK)

begin

if (CLK' event and CLK= '1') then

if (load = '1') then reg <= d;

else reg <= reg (2 downto 0) & '0';

end if;

end if;

end process;

dout <= reg (3);

end;

***4-bit Parallel in Parallel out Shift Register:*** The VHDL program for the *4-bit Parallel in Parallel out Shift Register* shown in Fig. 9.13 can be written as follows. This program follows *structural approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity pipo is

port(D:in std\_logic\_vector(3 downto 0);

reset,CLK:in std\_logic;

Q:out std\_logic\_vector(0 to 3));

end pipo ;

architecture arch\_pipo of pipo is

component dfff1 is

port(D,CLK,reset:in std\_logic;

Q:out std\_logic);

end component;

begin

a1 :dfff1 port map (D(0),CLK,reset,Q(0));

a2:dfff1 port map (D(1),CLK,reset,Q(1));

a3:dfff1 port map (D(2),CLK,reset,Q(2));

a4:dfff1 port map (D(3),CLK,reset,Q(3));

end;

**16.10.3 Counters**

***4-bit Asynchronous / Ripple Counter:*** The VHDL program for the *4-bit Asynchronous / Ripple Counter* shown in Fig. 8.1 can be written as follows. This program follows *structural approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity ripple\_counter is

port(Vcc,CLK,reset:in std\_logic;

Q, NQ:inout std\_logic\_vector(0 to 3));

end ripple\_counter;

architecture arch\_ripple\_counter of ripple\_counter is

component jkff1 is

port (J,K,CLK,reset: in std\_logic;

Q, NQ: inout std\_logic);

end component;

begin

a:jkff1 port map (Vcc,Vcc,CLK, reset,Q(0), NQ (0));

b:jkff1 port map (Vcc,Vcc,Q(0),reset,Q(1), NQ (1));

c:jkff1 port map (Vcc,Vcc,Q(1),reset,Q(2), NQ (2));

d:jkff1 port map (Vcc,Vcc,Q(2),reset,Q(3), NQ (3));

end;

The external clock pulse is applied to the first flip-flop only and, ‘1’ signal (Vcc) is given to JK inputs of all the Flip-Flops. The output of one flip-flop is connected to clock input of the next flip-flop.

***4-bit Synchronous Binary Counter:*** The VHDL program for the *4-bit Synchronous Binary Counter* shown in Fig. 8.11 can be written as follows. This program follows *structural approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity binarycounter is

port (Vcc,CLK:in std\_logic;

Q,NQ:inout std\_logic\_vector(0 to 3));

end binarycounter;

architecture arch\_binarycounter of binarycounter is

signal X1,Y1:std\_logic;

component jkff1 is

port (J,K,CLK: in std\_logic;

Q,NQ: inout std\_logic);

end component;

component andgate is

port(A,B:in std\_logic;

Y:out std\_logic);

end component;

component andgates is

port(A,B,C:in std\_logic;

Y:out std\_logic);

end component;

begin

a: jkff1 port map (Vcc,Vcc,CLK, Q(0),NQ(0));

b: jkff1 port map (Q(0),Q(0),CLK, Q(1),NQ(1));

c: andgate port map (Q(1),Q(0),X1);

d: jkff1 port map (X1,X1,CLK, Q(2),NQ(2));

f: andgates port map (Q(0),Q(1),Q(2),Y1);

g:jkff1 port map (Y1,Y1,CLK, Q(3),NQ(3));

end;

***3-bit Synchronous Up/Down Counter:*** The VHDL program for the 3*-bit Synchronous Up/Down Counter* shown in Fig. 8.34 can be written as follows. This program follows *structural approach.*

library ieee;

use ieee.std\_logic\_1164.all;

entity updowncounter is

port(Vcc,Up,CLK,reset:in std\_logic;

Q, NQ:inout std\_logic\_vector(0 to 2));

end updowncounter;

architecture arch\_updowncounter of updowncounter is

signal X1,Y1,Z1,A1,B1,C1,D1:std\_logic;

component jkff1 is

port (J,K,CLK: in std\_logic;

Q, NQ: inout std\_logic);

end component;

component andgate is

port(A,B:in std\_logic;

Y:out std\_logic);

end component;

component notgate is

port(A:in std\_logic;

Y:out std\_logic);

end component;

component orgate is

port(A,B:in std\_logic;

Y:out std\_logic);

end component;

begin

ff1:jkff1 port map(Vcc,Vcc,CLK,Q(0),NQ(0));

ag1:andgate port map(Q(0),Up,X1);

ng1:notgate port map (Up,Y1); – *Up control i/p is inverted to get*

ag2:andgate port map(NQ(0),Y1,Z1); – *Down (active low)*

og1:orgate port map(X1,Z1,A1);

ff2:jkff1 port map(A1,A1,CLK, Q(1),NQ(1));

ag3:andgate port map(Q(1),X1,B1);

ag4:andgate port map(Z1,NQ(1),C1);

og2:orgate port map(B1,C1,D1);

ff3:jkff1 port map(D1,D1,CLK, Q(2),NQ(2));

end;

When Up = 1, count starts from 0 to 2*n* – 1, where n is number of Flip-Flops. When Up = 0 (i.e., down mode), count starts from 2*n* – 1 to 0.

***4-bit Ring Counter:*** The VHDL program for the *4-bit Ring Counter* shown in Fig. 9.19 can be written using *behavioral approach* following Table 9.7 as follows.

library ieee;

use ieee.std\_logic\_1164.all;

entity ringcount is

port (CLK ,CLR: in std\_logic; Q : inout std\_logic\_vector (0 to 3));

end ringcount;

architecture ringcountarch of ringcount is

begin

process (CLK,CLR)

begin

if (CLR = ‘0’) then

Q <= “1000”;

elsif (CLR = ‘1’) then

if (CLK = ‘1’) and CLK’event then

Q(0) <= Q(3);

for i in 0 to 2 loop

Q(i+1) <= Q(i);

end loop;

end if;

end if;

end process;

end ringcountarch;

***4-bit Johnson Counter:*** The following is D-Flip-Flop program that follows *behavioural approach* and is used as component in Johnson counter.

library ieee;

use ieee.std\_logic\_1164.all;

entity dff2 is

port(D,CLK,reset:in std\_logic;

Q,NQ:inout std\_logic);

end dff2 ;

architecture arch\_dflipflop of dff2 is

begin

process (clk)

begin

if clk’event and clk=’1' then — CLK = 1:

if reset=’0' then – *Reset = 0 q = 0*

Q<=’0'; – *Reset = 1 q = D*

NQ<=’1';

else

Q<=D;

NQ<=not (Q);

end if;

end if;

end process;

end;

The VHDL Program for the *4-bit Johnson Counter* shown in Fig. 9.25 can be

written as follows that uses the above D-Flip-Flop program as component.

library ieee;

use ieee.std\_logic\_1164.all;

entity johnson is

port (CLK,reset:in std\_logic;

Q,NQ: inout std\_logic\_vector(0 to 3));

end johnson;

architecture arch\_johnson of johnson is

component dff2 is

port(D,CLK,reset:in std\_logic;

Q, NQ :inout std\_logic);

end component;

begin

a:dff2 port map (NQ(3),CLK,reset,Q(0),NQ(0));

b:dff2 port map (Q(0), CLK,reset,Q(1),NQ(1));

c:dff2 port map (Q(1), CLK,reset,Q(2),NQ(2));

d:dff2 port map (Q(2), CLK,reset,Q(3),NQ(3));

end;

**16.11 VHDL PROGRAM FOR ARITHMETIC LOGIC UNIT no change**

An Arithmetic logic unit is a logic circuit that performs various Boolean and arithmetic operations on n-bit operands. In this design, ALU has two 4-bit data inputs, A and B, 3-bit select inputs, S and four-bit output F. The output is defined by various arithmetic and Boolean operations on the inputs A and B as shown in the Table 16.1.

**VERILOG HDL**

**Modules**

A module is the basic building block in Verilog. A module can be an element or a collection of lower level design blocks. Typically, elements are grouped into modules to provide common functionality that is used in many places of design. A module provides the necessary functionality to the higher level block through its port interface( inputs and outputs), but hides the internal implementation.

Each module must have a module\_name , which is the identifier for the module, and a module\_terminal\_list, which describes the input and output terminals of the module.

**module <** module\_name> (module\_terminal\_list>);

…….

module internls>

……

……

end module

Each module can be defined at various levels of abstraction, depending on the needs of the design. The levels are defined below.

**Behavioural or algorithmic level**

A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. For this level, you must know the truth table and behaviour of the circuit.

**Data flow level**

The module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how data is processed in the design. For this level, you must know the input and output relation (logic equation).

**Gate level**

The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of gate level logic diagram.

**Operators**

There are three operators. They are

Unary operators precede the operand. (y= ~ a)

Binary operators appear between two operands. (y=a && c )

Ternary operators have two separate operators that separate three operands. (y= a ? c : d)

**Nets**

Nets represents connection between hardware elements. Nets declared primarily with the keyword **wire**.

**Registers**

Registers represent data storage elements. Registers retain value until another value is placed onto them.

**16.9 Verilog programs for combinational circuits**

**16.9.1 Verilog programs for LOGIC GATES**

The Verilog program for logic gates are written as follows.

***AND gate:*** The Verilog program for the *AND gate* shown in Fig. 3.3 (c) can be written as follows:

The Verilog program for AND gates are written following the *gate level approach or structural approach*

module andgate(a,b,y);

input a,b;

output y;

and a1(y,a,b);

endmodule

**T**he Verilog program for AND gates are written following the *data flow approach*.

module andgate(a,b,y);

input a,b;

output y;

assign y= a & b;

endmodule

The Verilog program for AND gates are written following the *BEHAVIOURAL APPROACH*

module andgate(a,b,y);

input a,b;

output reg y;

always@ (a or b)

assign y =(a & b);

endmodule

***OR gate:*** The Verilog program for the *OR gate* shown in Fig. 3.1(c) are written following the *data flow approach*.

module orgate(a,b,y);

input a,b;

output y;

assign y=a|b;

endmodule

***NAND gate:*** The VERILOG program for the *NAND gate* shown in Fig. 3.5(a) are written following the *data flow approach*.

module nandgate(a,b,y);

input a,b;

output y;

assign y=~(a&b);

endmodule

***NOR gate:*** The VERILOG program for the N*OR gate* shown in Fig. 3.6(a) are written following the *data flow approach*.

module norgate(a,b,y);

input a,b;

output y;

assign y=~(a|b);

endmodule

***NOT gate:*** The VERILOG program for the *MOT gate* shown in Fig. 3.4(b) are written following the *data flow approach*.

**NOT gate**

module notgate(a,y);

input a;

output y;

assign y=~a;

endmodule

***XOR gate:*** The VERILOG program for the X*OR gate* shown in Fig. 3.14 are written following the *data flow approach*.

module xorgate(a,b,y);

input a,b;

output y;

assign y=a^b;

endmodule

***XNOR gate:*** The VERILOG program for the XN*OR gate* shown in Fig. 3.15 are written following the *data flow approach*.

module xnorgate(a,b,y);

input a,b;

output y;

assign y=~(a^b);

endmodule

**16.9.2 ADDERS AND SUBTRACTORS**

***Half Adder:*** The Verilog program for the *Half Adder* shown in Fig. 5.1(b) can be written as follows.

|  |  |  |
| --- | --- | --- |
| Data flow approach | Gate level appraoch | Behavioural approach |
| module halfadder(a,b,sum,cy);  input a,b;  output sum,cy;  assign sum = a ^ b;  assign cy= a & b;  endmodule | module halfadder(a,b,sum,cy);  input a,b;  output sum,cy;  xor a1(sum,a,b);  and a2 (cy,a,b);  endmodule | module halfadder(a,b,sum,cy);  input a,b;  output reg sum,cy;  always@(a or b)  begin  case({a,b})  2'b00:  begin sum=0;cy=0; end  2'b01:  begin sum=1;cy=0; end  2'b10:  begin sum=1;cy=0; end  2'b11:  begin sum=0;cy=1; end  endcase  end  endmodule |

***Half Subtractor:*** The Verilog program for the *Half Subtractor* shown in Fig. 5.6(b) can be written as follows. This program follows *data flow approach*.

module halfsubtractor(a,b,sum,borrow);

input a,b;

output sum,borrow;

assign sum = a ^ b;

assign borrow= ~ a & b;

endmodule

***Full Adder:*** The Verilog program for the *Full Adder* shown in Fig. 5.3 can be written as follows. This program follows *data flow approach*.

module fulladder(a,b,c,sum,cy);

input a,b,c;

output sum,cy;

assign sum = a ^ b ^ c;

assign cy= (a & b)|(b&c)|(c&a);

endmodule

***Full Subtractor :*** The Verilog program for the *Full Subtractor* shown in Fig. 5.9 can be written as follows. This program follows *data flow approach.*

module fulladder(a,b,c,sum,borrow);

input a,b,c;

output sum, borrow;

assign sum = a ^ b ^ c;

assign borrow = (~a & b)|(b&c)|(c&(~a));

endmodule

***Four Bit Parallel Binary Adder:*** The Verilog program for the *Four Bit Parallel Binary Adder* shown in Fig. 5.10 can be written as follows. This program follows *structural approach*.

module fulladder (a,b,c,sum,carry);

input a,b,c;

output sum,carry;

wire y0,y1,y2;

xor(sum,a,b,c);

and(y0,a,b);

and(y1,a,c);

and(y2,b,c);

or(carry,y0,y1,y2);

endmodule

module rca(a,b,sum,cy);

input [3:0]a,b;

output [3:0] sum;

output cy;

wire c1,c2,c3;

fulladder f1 (a[0],b[0],0,sum[0],c1);

fulladder f2 (a[1],b[1],c1,sum[1],c2);

fulladder f3 (a[2],b[2],c2,sum[2],c3);

fulladder f4 (a[3],b[3],c3,sum[3],cy);

endmodule

**16.9.3 Multiplexer & Demultiplexer**

***4-to-1 Multiplexer*** : The Verilog program for the *4-to-1 Multiplexer* shown in Fig. 6.2 can be written as follows. This program follows *data flow approach.*

module mux(s,d,y);

input[1:0]s;

input [3:0]d;

output y;

assign y=(d[3]& (~s[1]) & (~s[0]))|(d[2]& (~s[1]) & (s[0]))|

(d[1]& (s[1]) & (~s[0]))|(d[0]& (s[1]) & (s[0]));

endmodule

***1-to-4 Demultiplexer:*** The Verilog program for the *1-to-4 Demultiplexer* shown in Fig. 6.7 can be written as follows. This program follows *data flow approach.*

module dmux(s,d,y);

input[1:0]s;

input d;

output [3:0]y;

assign y[3]=(d & (~s[1]) & (~s[0]));

assign y[2]=(d & (~s[1]) & (s[0]));

assign y[1]=(d & (s[1]) & (~s[0]));

assign y[0]=(d & (s[1]) & (s[0]));

endmodule

**16.9.4 Encoder and Decoder**

***Octal-to-Binary Encoder:*** The Verilog program for the *Octal-to-Binary Encoder* shown in Fig. 6.26 can be written as follows. This program follows *data flow approach.*

module octtobin(d,y);

input [7:0]d;

output [2:0]y;

assign y[2]=(d[1] & d[3] & d[5]& d[7]);

assign y[1]=(d[2] & d[3] & d[6]& d[7]);

assign y[0]=(d[4] & d[5] & d[6]& d[7]);

endmodule

***3-to-8 Decoder:*** The Verilog program for the *3-to-8 Decoder* shown in Fig. 6.12 can

be written as follows. This program follows *data flow approach.*

module decoder(a,b,c,d);

input a,b,c;

output [7:0]d;

assign d[0]=~a&~b&~c;

assign d[1]=~a&~b&c;

assign d[2]=~a&b&~c;

assign d[3]=~a&b&c;

assign d[4]=a&~b&~c;

assign d[5]=a&~b&c;

assign d[6]=a&b&~c;

assign d[7]=a&b&c;

endmodule

**16.9.5 4-Bit Parity Checker**

The Verilog program for the *4-bit Parity Checker* shown in Fig. 6.30 (a) can be written as follows. This program follows *data flow approach.* Here, the output is 1 when the number of 1’s in the inputs *W, X, Y* and *Z* is odd and 0 when the number of 1’s in the inputs is even.

library ieee;

module paritychecker(a,b,c,d,y);

input a,b,c,d;

output y;

assign y=((a^b)^c)^d;

endmodule

**16.9.6 4-bit Magnitude Comparator**

The Verilog program for the *4-bit Magnitude Comparator* shown in Fig.6.48 can be written as follows. This program follows *data flow approach.*

module mc(a,b,eq1,lt,gt);

input [3:0]a,b;

output eq1,lt,gt;

wire [3:0]eq;

assign eq[3]=(~(a[3])^b[3]);

assign eq[2]=(~(a[2])^b[2]);

assign eq[1]=(~(a[1])^b[1]);

assign eq[0]=(~(a[0])^b[0]);

assign eq1=eq[3]&eq[2]&eq[1]&eq[0];

assign lt=((~a[3])&b[3])|(eq[3]&(~a[2])&b[2])|(eq[3]&eq[2]&(~a[1])&b[1]) |

(eq[3]&eq[2]&eq[1]&(~a[0])&b[0]);

assign gt=(a[3]&(~b[3]))|(eq[3]&(a[2])&(~b[2]))|(eq[3]&eq[2]&(a[1])&(~b[1]))|

(eq[3]&eq[2]&eq[1]&(a[0])&(~b[0]));

endmodule

**16.9.7 Code Converters**

***4-bit Binary-to-Gray Code Converter:*** The Verilog program for the *4-bit Binary-*to- *Gray Code Converter*, shown in Fig. 6.41 can be written as follows. This program follows data flow *approach.*

module btog(b,g);

input [3:0]b;

output [3:0]g;

assign g[3]=b[3];

assign g[2]=b[3]^b[2];

assign g[1]=b[2]^b[1];

assign g[0]=b[1]^b[0];

endmodule

***4-bit Gray Code-to-Binary Converter:*** The Verilog program for the *4-bit Gray code-to-Binary Converter* shown in Fig. 6.44 can be written as follows. This program follows *data flow approach.*

module gtob(b,g);

input [3:0]b;

output [3:0]g;

assign b[3]=g[3];

assign b[2]=g[3]^g[2];

assign b[1]=g[1]^b[2];

assign b[0]=g[0]^b[1];

endmodule

**16.10 HDL FOR SEQUENTIAL LOGIC CIRCUITS**

**16.10.1 Realization of Flip-Flops**

***SR-Flip-Flop:*** The Verilog program for the *SR-Flip-Flop* shown in Fig. 7.4 (b) can be written as follows. This program follows *data flow approach.*

module srff(s,r,q,nq);

input s,r;

inout q,nq;

assign q= ~(r|nq);

assign nq=~(s | q);

endmodule

***Clocked SR-Flip-Flop:*** The Verilog program for the *Clocked SR-Flip-Flop* shown in Fig.7.10 can be written as follows. This program follows *behavioural approach.*

module srff(s,r,clk,clr,q);

input s,r,clr,clk;

output q;

reg q;

always@(posedge clk)

begin

if(clr==1)

q=0;

else

begin

case({s,r})

2'b00:

q=q;

2'b01:

q=0;

2'b10:

q=1;

endcase

end

end

endmodule

***D-Flip-Flop:*** The Verilog program for the *D-Flip-Flop* can be written as follows. This program follows *behavioral approach* as per truth table given in Table 7.6.

module dff(clk,d,clr,q);

input d,clk,clr;

output q;

reg q;

always@(posedge clk)

begin

if(clr==1)

q=0;

else

q=d;

end

endmodule

***JK-Flip-Flop:*** The Verilog program for the *JK-Flip-Flop* can be written as follows. This program follows *behavioral approach* as per truth table given in Table 7.9.

module jkff(clk,rst,j,k,q,qb);

input clk,rst,j,k;

output q,qb;

reg q,qb;

always@(negedge clk)

begin

if(rst==1)

begin

q<=0;

qb<=1;

end

else

begin

case({j,k})

2'b00:

begin

q<=q;

qb<=qb;

end

2'b01:

begin

q<=0;

qb<=1;

end

2'b10:

begin

q<=1;

qb<=0;

end

2'b11:

begin

q<=~q;

qb<=~qb;

end

endcase

end

end

endmodule

***T-Flip-Flop:*** The Verilog program for the *T-Flip-Flop* can be written as follows. This program follows *behavioral approach* as per truth table given in Table 7.12.

module tff(clk,t,clr,q);

input t,clk,clr;

output q;

reg q;

always@(posedge clk)

begin

if(clr==1)

q=0;

else

begin

if(t==0)

q=q;

else if(t==1)

q=~q;

end

end

endmodule

**16.10.2 Realization of Shift Registers**

***4-bit Serial in Serial out Shift Register:*** The Verilog program for the *4-bit Serial in Serial out Shift Register* shown in Fig.9.4 (a) can be written as follows. This program follows *structural approach.* When clock pulse is applied, output of one flip-flop is given as input of next flip-flop. The serial output is taken from the last flip-flop.

module siso(clk,q,d,clr);

input d;

input clk,clr;

output q;

wire a,b,c;

dff a1(a,d,clk,clr);

dff a2(b,a,clk,clr);

dff a3(c,b,clk,clr);

dff a4(q,c,clk,clr);

endmodule

module dff(q,d,clk,clr);

input d,clk,clr;

output q;

reg q;

always@(posedge clk)

begin

if(clr==0)

q=0;

else

q=d;

end

endmodule

***4-bit Serial in Parallel out Shift Register:***

The Verilog program for the *4-bit Serial in Parallel out Shift Register* shown in Fig. 9.7 can be written as follows. This program follows *structural approach.*

module sipo(clk,q1,q2,q3,q4,d);

input d;

input clk;

output q1,q2,q3,q4;

dff a1(q1,d,clk,clr);

dff a2(q2,q1,clk,clr);

dff a3(q3,q2,clk,clr);

dff a4(q4,q3,clk,clr);

endmodule

***4-bit Parallel in serial out Shift Register:*** The Verilog program for the *4-bit Parallel in Parallel out Shift Register* shown in Fig. 9.10 can be written as follows. This program follows *structural approach.*

module piso(a,b,c,d,clk,rst,sl,q1,q2,q3,q4);

input a,b,c,d,clk,rst,sl;

output q1,q2,q3,q4;

wire d2,d3,d4;

wire x1,x2,x3,y1,y2,y3;

dff f1(q1,a,clk,rst);

and(x1,q1,sl);

and(y1,~sl,b);

or(d2,x1,y1);

dff f2(q2,d2,clk,rst);

and(x2,q2,sl);

and(y2,~sl,c);

or(d3,x2,y2);

dff f3(q3,d3,clk,rst);

and(x3,q3,sl);

and(y3,~sl,d);

or(d4,x3,y3);

dff f4(q4,d4,clk,rst);

endmodule

***4-bit Parallel in Parallel out Shift Register:*** The Verilog program for the *4-bit Parallel in Parallel out Shift Register* shown in Fig. 9.13 can be written as follows. This program follows *structural approach.*

module pipo(d1,d2,d3,d4,clk,rst,q1,q2,q3,q4);

input d1,d2,d3,d4,clk,rst;

output q1,q2,q3,q4;

dff fa(q1,d1,clk,rst);

dff fb(q2,d2,clk,rst);

dff fc(q3,d3,clk,rst);

dff fd(q4,d4,clk,rst);

endmodule

**16.10.3 Counters**

***4-bit Asynchronous / Ripple Counter:*** The Verilog program for the *4-bit Asynchronous / Ripple Counter* shown in Fig. 8.1 can be written as follows. This program follows *structural approach.*

module ripp(clk,rst,vcc,q,qb);

input clk,rst,vcc;

output [3:0]q,qb;

jkff ff1(clk,rst,vcc,vcc,q[0],qb[0]);

jkff ff2(q[0],rst,vcc,vcc,q[1],qb[1]);

jkff ff3(q[1],rst,vcc,vcc,q[2],qb[2]);

jkff ff4(q[2],rst,vcc,vcc,q[3],qb[3]);

endmodule

The external clock pulse is applied to the first flip-flop only and, ‘1’ signal (Vcc) is given to JK inputs of all the Flip-Flops. The output of one flip-flop is connected to clock input of the next flip-flop.

***4-bit Synchronous Binary Counter:*** The Verilog program for the *4-bit Synchronous Binary Counter* shown in Fig. 8.11 can be written as follows. This program follows *structural approach.*

module sync(clk,rst,vcc,q,qb);

input clk,rst,vcc;

output [3:0]q,qb;

wire q1,q2;

jkff ff1(clk,rst,vcc,vcc,q[0],qb[0]);

jkff ff2(clk,rst,q[0],q[0],q[1],qb[1]);

assign q1=q[0] & q[1];

jkff ff3(clk,rst,q1,q1,q[2],qb[2]);

assign q2=q[0]& q[1] & q[2];

jkff ff4(clk,rst,q2,q2,q[3],qb[3]);

endmodule

***3-bit Synchronous Up/Down Counter:*** The Verilog program for the 3*-bit Synchronous Up/Down Counter* shown in Fig. 8.34 can be written as follows. This program follows *structural approach.*

module updown(clk,rst,vcc,q,qb,up);

input clk,rst,vcc,up;

output [2:0]q,qb;

wire up1,up2,down1,down2,updown1,updown2;

jkff ff1(clk,rst,vcc,vcc,q[0],qb[0]);

assign up1=up & q[0];

assign down1= ~up & qb[0];

assign updown1= up1|down1;

jkff ff2(clk,rst,updown1,updown1,q[1],qb[1]);

assign up2=up1&q[1];

assign down2=down1&qb[1];

assign updown2=up2|down2;

jkff ff4(clk,rst,updown2,updown2,q[2],qb[2]);

endmodule

When Up = 1, count starts from 0 to 2*n* – 1, where n is number of Flip-Flops. When Up = 0 (i.e., down mode), count starts from 2*n* – 1 to 0.

**MOD 10 COUNTER:**

The Verilog program for the mod-10 *Counter*  follows *behaviural approach.*

module mod\_ten(clk, reset, Q);

input clk, reset;

output [3:0] Q;

reg [3:0] Q;

always @ (posedge clk)

begin

if (reset)

Q = 4'b0;

else

Q =(Q+1)%10;

end

endmodule

***4-bit Ring Counter:*** The Verilog program for the *4-bit Ring Counter* shown in Fig. 9.19 can be written using *structural approach* following Table 9.7 as follows.

module ringcounter (clk, rst, q,qb);

input clk,rst;

output [3:0] q,qb;

dff aa (clk,1'b0,q[3],q[0],qb[0],rst);

dff ab (clk,rst,q[0],q[1],qb[1],1'b0);

dff ac (clk,rst,q[1],q[2],qb[2],1'b0);

dff ad (clk,rst,q[2],q[3],qb[3],1'b0);

endmodule

The following is D-Flip-Flop program that follows *structural* and is used as component in Ring counter.

module dff (clk, rst, d,q,qb,pre);

input clk, rst, d,pre;

output reg q,qb;

always @ (posedge clk)

begin

if (rst==1 && pre==0)

begin

q=0;

qb=1;

end

else if (rst==0 && pre==1)

begin

q=1;

qb=0;

end

else

begin

q=d;

qb=~d;

end

end

endmodule

***4-bit Johnson Counter:*** The following is D-Flip-Flop program that follows *structural* and is used as component in Johnson counter.

module dff (clk, rst, d,q,qb,pre);

input clk, rst, d,pre;

output reg q,qb;

always @ (posedge clk)

begin

if (rst==1 && pre==0)

begin

q=0;

qb=1;

end

else if (rst==0 && pre==1)

begin

q=1;

qb=0;

end

else

begin

q=d;

qb=~d;

end

end

endmodule

The Verilog Program for the *4-bit Johnson Counter* shown in Fig. 9.25 can be written as follows that uses the above D-Flip-Flop program..

module ringcounter (clk, rst, q,qb);

input clk,rst;

output [3:0] q,qb;

dff aa (clk,1'b0,qb[3],q[0],qb[0],rst);

dff ab (clk,rst,q[0],q[1],qb[1],1'b0);

dff ac (clk,rst,q[1],q[2],qb[2],1'b0);

dff ad (clk,rst,q[2],q[3],qb[3],1'b0);

endmodule

**16.11 Verilog PROGRAM FOR ARITHMETIC LOGIC UNIT**

An Arithmetic logic unit is a logic circuit that performs various Boolean and arithmetic operations on n-bit operands. In this design, ALU has two 4-bit data inputs, A and B, 3-bit select inputs, S and four-bit output F. The output is defined by various arithmetic and Boolean operations on the inputs A and B as shown in the Table 16.1.

***Table 16.1*** *4-bit ALU operation*

|  |  |  |
| --- | --- | --- |
| **Operation** | **Inputs (S3S2S1S0)** | **Outputs (F)** |
| Clear | 0000 | 0000 |
| ADD | 0001 | A + B |
| SUB | 0010 | A – B |
| MUL | 0011 | A\* B |
| DIV | 0100 | A/B |
| AND | 0101 | A AND B |
| OR | 0110 | A OR B |
| NOT | 0111 | NOT A |
| NAND | 1000 | A NAND B |
| NOR | 1001 | A NOR B |
| XOR | 1010 | A XOR B |
| XNOR | 1011 | A XNOR B |
| Preset | 1100 | 1111 |

module alu (a,b,s,y);

input [3:0]s;

input [3:0]a,b;

output reg [3:0] y;

always@(s,a,b)

begin

case (s)

4'b0000:

y=a+b;

4'b0001:

y=a-b;

4'b0010:

y=a\*b;

4'b0011:

y=a/b;

4'b0100:

y=a&b;

4'b0101:

y=a|b;

4'b0110:

y=~a;

4'b0111:

y=~(a&b);

4'b1000:

y=~(a|b);

4'b1001:

y=a^b;

4'b1010:

y=~(a^b);

4'b1011:

y=4'b1111;

default:

y=4'b0000;

endcase

end

endmodule