2 to 4 decoder

```
entity dec is
   Port ( i : in  STD_LOGIC_VECTOR (1 downto 0);
          o : out  STD_LOGIC_VECTOR (3 downto 0);
          en : in  STD_LOGIC);
end dec;
architecture Behavioral of dec is
begin
process (en,i)
begin
if (en = '1') then
if ( i = "00" ) then
o <= "0001";
elsif (i = "01") then
o <= "0010";
elsif (i = "10") then
o <= "0100";
else o <= "1000";
end if;
else o<= "0000";
end if;
end process;
end Behavioral;
```

4 to 1 MUX

```
entity mux4to1 is

port ( w0, w1, w2, w3 : in std_logic ; s : in std_logic_vector(1 downto 0) ;

f : out std_logic ) ;

end mux4to1 ;

architecture behavior of mux4to1 is

Begin

Process(s,w0,w1,w2,w3)

begin

case s is

 when "00" => f <= w0;

when "01" => f <= w1;

when "10" => f <= w2;

When others => f <= w3;

End case;

End process;

end behavior ;
```

PARALLEL ADDER

```
Entity parallel_adder is

port ( a,b : in std_logic_vector (3 downto 0); cin : in std_logic ;

O : out std_logic_vector (4 downto 0)) ;

End parallel_adder ;

architecture behavior of parallel_adder is

Begin

Process(a,b,cin)
```

```
Variable c : std_logic;

Begin

C := cin;

For i in 0 to 3 loop

O(i) <= a(i) xor b(i) xor c;

C := (a(i) and b(i)) or (a(i) and c) or ( b(i) and c);

End loop;

O(4) <= c;

End process;

end behavior ;
```

SR LATCH

```
entity srlatch is

  port(r,s:in std_logic ; q,qbar:out std_logic);

end srlatch;

architecture behav of srlatch is

begin

process (s,r)

variable s1: std_logic:= '0';

variable r1: std_logic:= '1';

begin

s1:= s nand r1;

  r1:= r nand s1;

         q <= s1;

         qbar <= r1;

         end process;    end behav;
```

JK FLIPFLOP

```
   entity jk1 is
   port (j,k,clk:in std_logic ; q:inout std_logic:= '0';qbar:inout std_logic:= '1');
end jk1;
architecture behav of jk1 is
begin
process (j,k,clk)
variable s1,r1,d: std_logic;
begin
   s1 := '0';
   r1 := '1';
              if (clk = '1' and clk' event) then
   s1:= (j and qbar) or ( not(k) and q);
   r1:= not (s1);
              q <= s1;
              qbar <= r1;
              else
                q <= q;
qbar <= qbar;
              end if;
              end process;
```

SISO REGISTER

```
Entity siso is

Port (din,clk,clr : in std_logic; q : inout std_logic);

End siso;


Architecture behav of siso is

signal q1:std_logic_vector (3 downto 0) := "0000";

begin

Process(din,clk,clr,q1)

Begin

If (clr = '1') then

q <= '0';

q1 <= "0000";

elsIf (clk = '1' and clk'event) then

q1(0) <= din;

For I in 0 to 2 loop

q1(i+1) <= q1(i) ;

End loop;

q <= q1(3);

Else q <= q;

End if;

End process;

End behav;
```

SIPO REGISTER

```
entity sipo is
```

```
 port(
 clk, clear : in std_logic;
 Input_Data: in std_logic;
 Q: inout std_logic_vector(3 downto 0) );
end sipo;
architecture arch of sipo is
begin
 process (clk)
 begin

 if clear = '1' then
 Q <= "0000";
 elsif (CLK'event and CLK='1') then
 Q(3 downto 1) <= Q(2 downto 0);
 Q(0) <= Input_Data;
 end if;
 end process;
end arch;
```

PISO REGISTER

```
Entity piso is
Port (clk,clr,en : in std_logic;  pin : in std_logic_vector (3 downto 0);q : inout std_logic);
End piso;
Architecture behav of piso is
signal q1:std_logic_vector (3 downto 0) := "0000";
begin
```

```
Process(clk,clr,q1,pin,en)

Begin

If (clr = '1') then

q <= '0';

q1 <= "0000";

elsIf (clk = '1' and clk'event) then

q1(3) <= pin(3);

  For I in 2 downto 0 loop

 If (en = '0') then

Q1(i) <= q1(i+1);

Else q1(i) <= pin(i);

End if;

End loop ;

q <= q1(0);

Else q <= q;

End if;

End process;

End behav;
```

UP-DOWN COUNTER

```
entity updown_count is

 Port ( clk,rst,updown : in  STD_LOGIC;  count : out  STD_LOGIC_VECTOR (3 downto 0));

end updown_count;

architecture Behavioral of updown_count is

signal temp:std_logic_vector(3 downto 0):="0000";

begin
```

```
process(clk,rst)
 begin
 if(rst='1')then
temp<="0000";
elsif(rising_edge(clk))then
if(updown='0')then
temp<=temp+1;
else temp<=temp-1;
end if;
end if;
 end process;
 count<=temp;
 end Behavioral;
```

MOD-10 COUNTER

```
entity mod10_counter is
     port(
         clk : in STD_LOGIC;
         reset : in STD_LOGIC;
         dout : out STD_LOGIC_VECTOR(3 downto 0)
         );
end mod10_counter;
architecture behav of mod10_counter is
begin
    counter : process (clk,reset) is
    variable m : integer range 0 to 15 := 0;
```

```
begin
   if (reset='1') then
      m := 0;
   elsif (rising_edge (clk)) then
      m := m + 1;
   end if;
   if (m=10) then
      m := 0;
   end if;
   dout <= conv_std_logic_vector (m,4);
end process counter;
         end behav;
```

BARREL SHIFTER

```
Entity  barrel_shifter  is

port ( I: in std_logic_vector ( 7 downto 0); S : in std_logic_vector (2 downto 0); O: out std_logic_vector (7
downto 0)) ;

End barrel_shifter ;

architecture behavior of barrel_shifter is

Component mux1 is port (d1,d2 : in std_logic; s : in std_logic; o: out std_logic); end component;

Signal o1,o2 : std_logic_vector ( 7 downto 0) := ( others => '0') ;

Begin

L1 : for j in 0 to 2 generate

L2:  if (j = 0) generate

L3 : for k in 0 to 7 generate

L4 : if  (k <4) generate

Mux11 : mux1 port map (i(k), i(k+4),s(2), o1(k));

End generate l4;

L5 : if (k >3) generate

Mux22 : mux1 port map (i(k), '0',s(2), o1(k));

End generate l5;

End generate l3;

End generate l2;

L6 : if (j = 1) generate

L7 : for k in 0 to 7 generate

L8 : if  (k <6) generate

Mux33 : mux1 port map (o1(k), o1(k+2),s(1), o2(k));

End generate l8;

L9 : if (k >5) generate
```

```
Mux44 : mux1 port map (o1(k), '0',s(1), o2(k));

End generate l9;

End generate l7;

End generate l6;

L10 : if (j = 2) generate

L11 : for k in 0 to 7 generate

L12 : if  (k <7) generate

Mux55 : mux1 port map (o2(k), o2(k+1),s(0), o(k));

End generate l12;

L13 : if (k >6) generate

Mux66 : mux1 port map (o2(k), '0',s(0), o(k));

End generate l13;

End generate l11;

End generate l10;

End generate l1;

End behavior;
```

SIGNED MULTIPLIER

```
Entity multiplier is

Generic (n: integer := 4);

Port(a,b : in std_logic_vector (n-1 downto 0);

o: out std_logic_vector(2*n-1 downto 0));

End multiplier;

Architecture arch of multiplier is

Signal a_signed, b_signed : signed(n-1 downto 0);
Signal o_signed : signed(2*n-1 downto 0);

Begin

a_signed <= signed(a);

b_signed <= signed(b);

o_signed <=  a_signed * b_signed;

o <= std_logic_vector(o_signed);


End arch;
```

MEALEY FSM

```
Entity mealey is

Port (clock,reset,input : in std_logic; output : out std_logic);

End mealey;

Architecture arch of mealey is

TYPE state IS (S0, S1);

SIGNAL Mealy_state: state;

begin

U_Mealy: PROCESS(clock, reset)

BEGIN

        IF(reset = '1') THEN

                Mealy_state <= S0;

        ELSIF (clock = '1' AND clock'event) THEN

                CASE Mealy_state IS

                  WHEN S0 =>    IF input = '1' THEN

            Mealy_state <= S1;

         ELSE
```

```
                 Mealy_state <= S0;
            END IF;
WHEN S1 =>    IF input = '0' THEN
                      Mealy_state <= S0;
                   ELSE
                      Mealy_state <= S1;
                   END IF;
                  END CASE;
           END IF;
END PROCESS;
Output <= '1' WHEN (Mealy_state = S1 AND input = '0') ELSE '0';
End arch;
```

MOORE FSM

```
Entity moore is
Port (clock,reset,x : in std_logic; y : out std_logic);
End moore;
Architecture arch of moore is
TYPE state IS (S0, S1, S2);
SIGNAL Moore_state: state;
begin
U_Moore: PROCESS (clock, reset)
BEGIN
        IF(reset = '1') THEN
                Moore_state <= S0;
        ELSIF (clock = '1' AND clock'event) THEN
```

```
                CASE Moore_state IS
                WHEN S0 =>     IF x = '1' THEN
           Moore_state <= S1;
        ELSE
          Moore_state <= S0;
        END IF;
WHEN S1 =>   IF x = '0' THEN
                    Moore_state <= S2;
                ELSE
                    Moore_state <= S1;
                END IF;
                  WHEN S2 =>
                        IF x = '0' THEN
                    Moore_state <= S0;
                        ELSE
                    Moore_state <= S1;
                END IF;
                END CASE;
        END IF;
END PROCESS;
y <= '1' WHEN Moore_state = S2 ELSE '0';


End arch;
```

The vast array of MOSFET-based digital circuitry is built around the CMOS inverter. This simple yet extremely effective circuit uses the combination of an NMOS transistor and a PMOS transistor to generate output signals that are, under normal operating conditions, always either logic high or logic low. The term pass-transistor logic refers to a different form of MOSFET circuitry that doesn't generate logic-high and logic-low voltages in the way that typical CMOS circuitry does.

The Low-Resistance Path

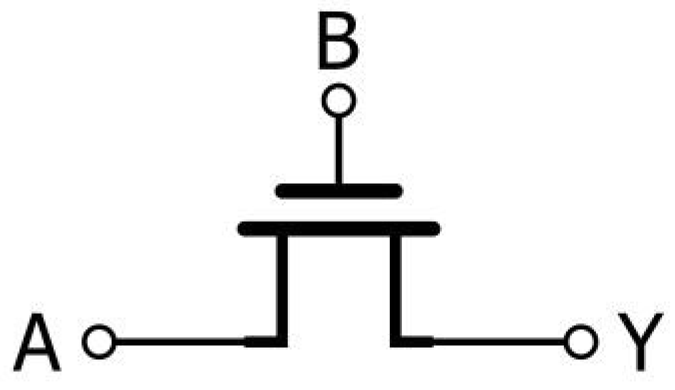
The basic answer to this question is that logic high indicates a voltage at the supply rail, representing a binary 1, and logic low indicates a voltage at ground, representing a binary 0. This description is a good place to start, and it becomes more accurate if we say "near the supply rail" and "near ground" instead of "at the supply rail" and "at ground." This modification accounts for the fact that currents flowing through the channel of the NMOS or PMOS create a small voltage difference between source and drain.

This issue of current flow through the channel leads to a more subtle, but nonetheless crucial, aspect of typical CMOS functionality. A CMOS inverter ensures that the output node will have a low-resistance connection to the supply rail or ground; the inverter always has the NMOS conducting and the PMOS in cutoff or the PMOS conducting and the NMOS in cutoff. This is why we can say that CMOS circuits drive a logic low or logic high. It is also why logic circuits built around the inverter topology are so reliably "digital"—all the nodes have a clearly defined binary state because they always have a low-resistance path to the supply voltage or ground.

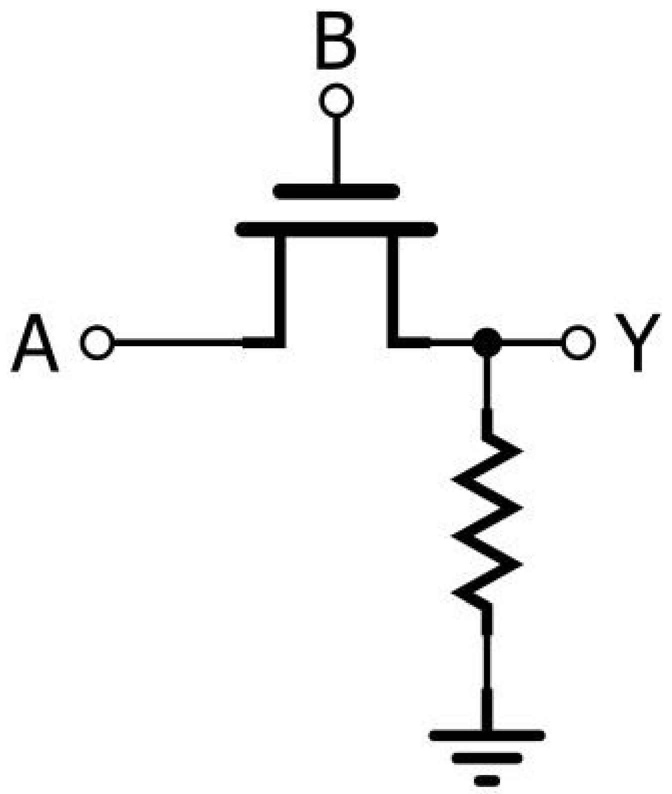
Pass-transistor logic (PTL), also known as transmission-gate logic, is based on the use of MOSFETs as switches rather than as inverters. The result is (in some cases) conceptual simplification, but the CMOS inverter's strict logic-high/logic-low output characteristic is lost.

An Example of Pass-Transistor Logic

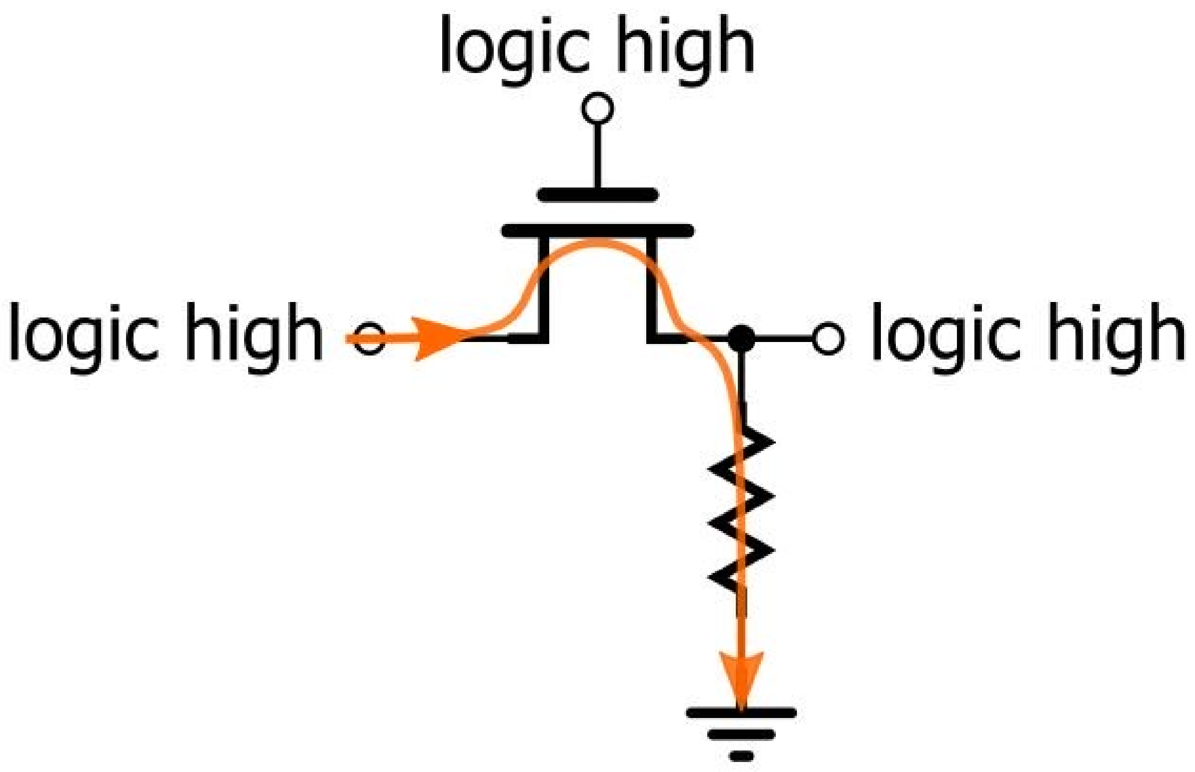
It is possible to use a single NMOS transistor as a PTL switch; the switch is considered closed when the voltage applied to the gate is logic high, and it is considered open when the voltage applied to the gate is logic low. The following diagram shows an AND gate (or at least something similar to an AND gate) that uses only one transistor.



The output (Y) is logic high when the input (A) is logic high and the switch-control signal (B) is logic high, and it is not logic high for all other combinations. That sounds like the AND truth table, but can we really call this an AND gate? That depends on your perspective. The problem is that the circuit doesn't drive a logic low when the B input is logic low. It's simply disconnected, i.e., floating. To establish a logic low, we need a pull-down resistor:



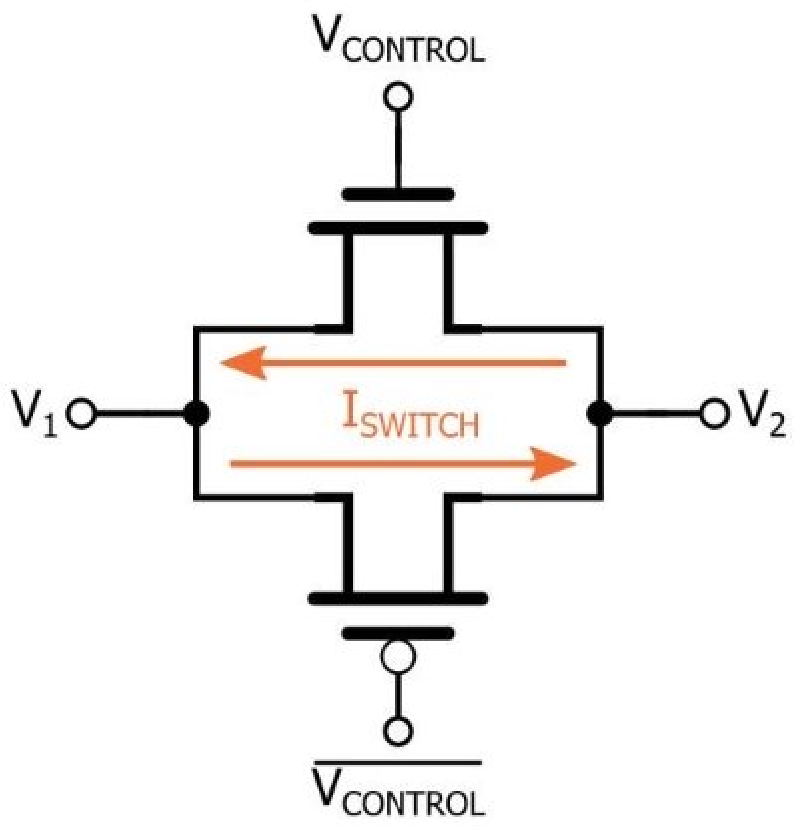
Now we have a functional AND gate, and we've used only one transistor and one resistor, whereas a standard CMOS-inverter-based AND gate requires six transistors. However, the PTL circuit is by no means equivalent to the standard CMOS version. First of all, it does not reliably provide a low-resistance path to ground. Second, it dissipates static power whenever the output is logic high—current flows from the input, through the NMOS, through the pull-down resistor, to ground:



This means that we have lost an extremely beneficial property of inverter-based logic, namely, that the power supply delivers significant amounts of current only during switching. (That's why CMOS power dissipation is proportional to frequency—more switching means more current, and more current means more power.)

NMOS vs. CMOS in Pass-Transistor Logic

PTL is built around MOSFET switches that either pass (hence the name) or block a signal. Using an NMOS transistor as the switch is certainly a good way to reduce transistor count, but a lone NMOS isn't impressive in terms of performance. A much better solution is the CMOS transmission gate:



The lone NMOS and the CMOS transmission gate are briefly compared in this article. There's no doubt that the transmission gate is, in general, the superior implementation, but consider the trade-off. Obviously an additional transistor is required, but note also that the PMOS is driven not by the switch-control signal but by the complement of the switch-control signal. This is not a problem if the circuit that generates the input signal is, for example, a D flip-flop that provides both a QQ and a ¯¯¯¯QQ¯ output. Usually, though, only one input signal is available, and in such cases the use of a CMOS transmission gate means that we must also have an inverter to create the control signal for the second FET.