Vážení zákazníci,

dovolujeme si Vás upozornit, že na tuto ukázku knihy se vztahují autorská práva, tzv. copyright.

To znamená, že ukázka má sloužit výhradně pro osobní potřebu potenciálního kupujícího (aby čtenář viděl, jakým způsobem je titul zpracován a mohl se také podle tohoto, jako jednoho z parametrů, rozhodnout, zda titul koupí či ne).

Z toho vyplývá, že není dovoleno tuto ukázku jakýmkoliv způsobem dále šířit, veřejně či neveřejně např. umisťováním na datová média, na jiné internetové stránky (ani prostřednictvím odkazů) apod.

redakce nakladatelství BEN – technická literatura
redakce@ben.cz

8

ZÁKLADNÍ FUNKČNÍ BLOKY

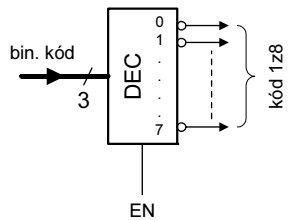
Při návrhu číslicových systémů se pracuje pokud možno s většími funkčními bloky. Mohou být implementovány jako integrované obvody, zpravidla střední integrace (MSI), nebo jako standardní bloky v počítačových návrhových programech. Zde budou postupně probrány ty základní a nejčastější.

8.1 Dekodér

Dekodér převádí binární kód na **kód *1zN***. Při kódu *1zN* je vždy jen na jednom výstupu aktivní stav (předpokládejme, že aktivní stav je 0). Pozice tohoto aktivního výstupu odpovídá binárnímu číslu o *k* bitech, přiváděnému na vstup. Platí, že $N = 2^k$. V *tab. 8.1* je pravdivostní tabulka dekodéru z tříbitového binárního kódu na kód *1 z 8* a na *obr. 8.1* je jeho schematické znázornění.

Tab. 8.1 Pravdivostní tabulka dekodéru

Vstup MSB		LSB	Výstup							
x_2	x_1	x_0	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1

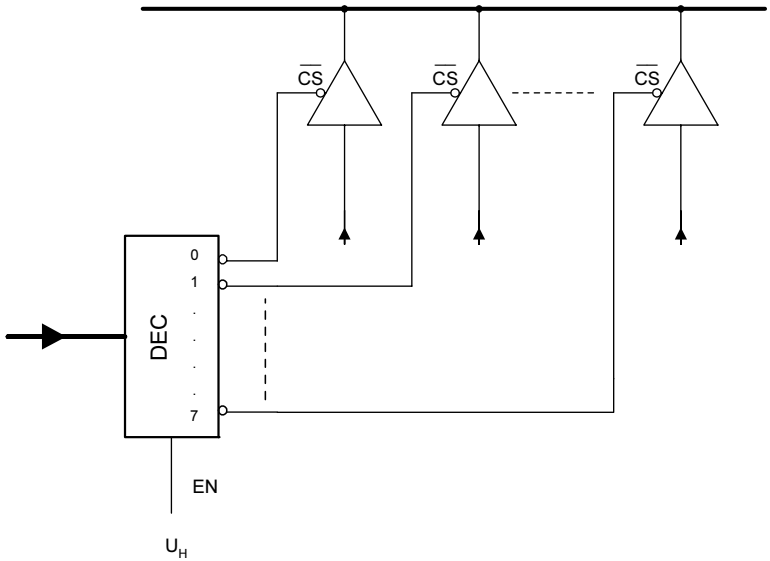


Obr. 8.1 Vstupy a výstupy dekodéru

Dekodéry jsou velmi často opatřeny vstupem pro **blokování** označeným EN (angl. enable), kterým lze všechny výstupy uvést do neaktivního stavu, tj. 1. Při nenegovaném EN na obrázku by byl dekodér blokován při *EN = 0* a odblokován při *EN = 1*. Je třeba pečlivě rozlišit blokování **třístavových členů** od blokování dekodéru – v prvém případě se blokovaný výstup projevuje vysokoimpedančním stavem, ve druhém případě neaktivním stavem (stavem 1).

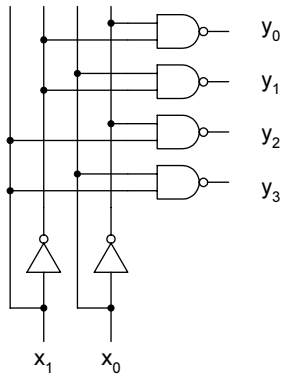
Dekodér se používá pro funkci výběru jednoho z několika obvodů. Velmi často se jedná o obvody třístavové, připojené na společnou sběrnici. Jejich výběrové vstupy jsou zpravidla

negované ($\overline{CS}$) a tomu odpovídají i negované výstupy dekodéru. Při zablokovaném dekodéru pak není vybrán **ani jeden** třístavový obvod ze skupiny. *Obr. 8.2* ukazuje ovládání skupiny třístavových členů. Blokování celé skupiny není využito, neboť na vstupu EN je zde trvale stav 1.



Obr. 8.2 Ovládání obvodů na sběrnici dekodérem

Vnitřní zapojení dekodéru je jednoduché. Každý výstupní signál je tvořen logickým součinem všech vstupních signálů v přímém nebo častěji negovaném tvaru. Zapojení dekodéru *bin/1z4* ukazuje *obr. 8.3*.



Obr. 8.3 Vnitřní zapojení dekodéru 1z4

Pokud se mění **dva nebo více** vstupních signálů téměř současně, mohou se krátkodobě vyskytovat mezistavy, které nepatří k ustálenému předcházejícímu nebo následujícímu stavu.

Vzniknou tak falešné krátké impulzy obecně na **kterýchkoliv** výstupech dekodéru. S tímto jevem je třeba vždy počítat.

Následující dva výpisy kódu ukazují realizaci výše popsaného tříbitového dekodéru z binárního kódu na kód 1 z 8 v jazyce VHDL. První výpis kódu realizuje dekodér pomocí konstrukce procesu s **case** (realizuje pravdivostní tabulku).

```
library ieee;
use ieee.std_logic_1164.all;

entity dekoder is
   port (
      x : in std_logic_vector (2 downto 0);
      y : out std_logic_vector (7 downto 0)
   );
end dekoder;

architecture a_dekoder of dekoder is
begin
   -- proces se spousti pri zmene signalu x
   process (x) begin
      case x is
         when "000" => y <= "11111110";
         when "001" => y <= "11111101";
         when "010" => y <= "11111011";
         when "011" => y <= "11110111";
         when "100" => y <= "11101111";
         when "101" => y <= "11011111";
         when "110" => y <= "10111111";
         when "111" => y <= "01111111";
         when others  => null; -- v ostatnich pripadech zadna cinnost
      end case;
   end process;
end a_dekoder;
```

Výpis 8.1 Dekodér (binární kód na kód 1 z N) pomocí procesu a case

Druhý výpis kódu pak realizuje stejný dekodér pomocí procesu a nastavení příslušného bitu v bitovém poli pomocí indexace s typovou konverzí. Pro typovou konverzi je použito přetypování vstupního vektoru *x* na typ **unsigned** a následně konverze pomocí funkce **to_integer** na typ **integer**. Touto hodnotou je pak indexován vektor *y* a nastaven příslušný bit do log. 0. Kód používá balíček **numeric_std** z knihovny **ieee**, který definuje výše zmíněný typ **unsigned** a konverzní funkce.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dekoder is
   port (
      x : in std_logic_vector (2 downto 0);
      y : out std_logic_vector (7 downto 0)
   );
end dekoder;
```

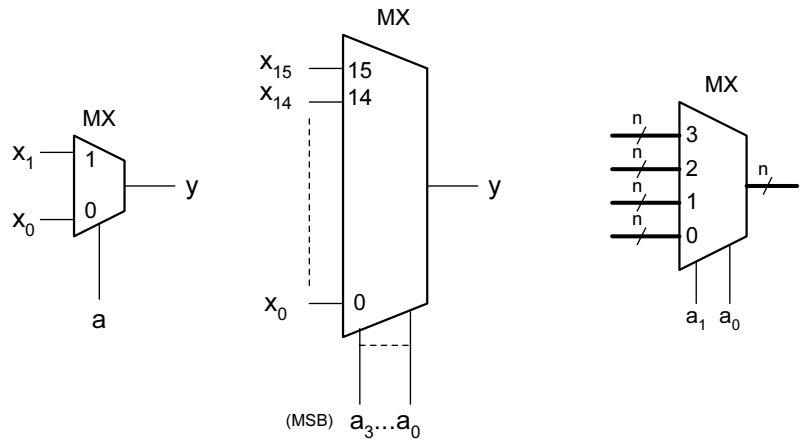
```
architecture a_dekoder of dekoder is
begin
   -- proces se spousti pri zmene signalu x
   process (x)
   begin
      -- naplni vsechny bity vektoru y log. 1
      y <= (others => '1');
      -- podle hodnoty x nastavi prislusny bit vektoru y do log. 0
      y(to_integer(unsigned(x))) <= '0';
   end process;
end a_dekoder;
```

Výpis 8.2 Dekodér (binární kód na kód 1 z N) pomocí procesu a indexace vektoru

8.2 Multiplexor

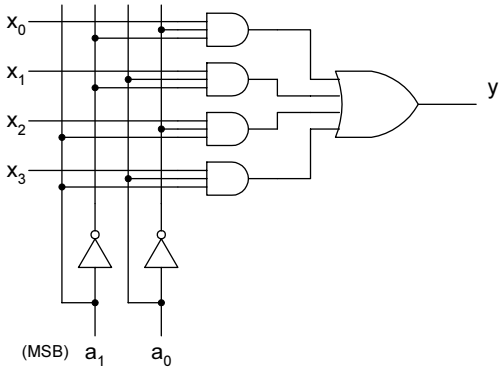
Multiplexor je vlastně číslicový **přepínač**. Má vstupy adresy (v binárním kódu) a vstupy přepínaných signálů. Ty jsou očíslované a čísla vyznačují, při které hodnotě adresy je daný vstupní signál převeden na výstup. Nejjednodušší je dvojkanálový multiplexor s pouze jedním adresovým signálem, který je používán velmi často. Jsou i multiplexory o větším počtu kanálů. Směr přenosu signálu nelze obrátit (na rozdíl od multiplexoru analogového, založeného na přesných spínačích CMOS).

Často je nutné přepínat současně celé **skupiny signálů**. Pak se používá skupinový (sběrnicový) multiplexor. Na *obr. 8.4* je postupně zleva dvoukanálový multiplexor, osmikanálový multiplexor, a skupinový čtyřkanálový multiplexor. U toho je adresou vybrána vždy jedna skupina vodičů a převedena na vodiče výstupní. Všechny vstupní kanály i výstupní kanál mají samozřejmě stejný počet signálů.



Obr. 8.4 Různé varianty multiplexorů

Vnitřní zapojení multiplexoru není složité. *Obr. 8.5* ukazuje vnitřní zapojení čtyřkanálového multiplexoru.



Obr. 8.5 Vnitřní zapojení čtyřkanálového multiplexoru

Následující tři výpisy kódu postupně ukazují realizaci dvoukanálového multiplexoru, šestnáctikanálového multiplexoru a skupinového čtyřkanálového multiplexoru. První výpis kódu realizuje dvoukanálový multiplexor pomocí přiřazení s konstrukcí **when-else**.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux2_1 is
   port (
      x0, x1 : in std_logic;
      a : in std_logic;
      y : out std_logic
   );
end mux2_1;

architecture mux_when_else of mux2_1 is
begin
   y <= x1 when a = '1' else x0;
end mux_when_else;
```

Výpis 8.3 Dvoukanálový multiplexor pomocí konstrukce when-else

Další výpis kódu realizuje šestnáctikanálový multiplexor pomocí procesu a indexace bitového vektoru s typovou konverzí. V citlivostním seznamu procesu musí být uvedeny dva vstupní vektory (*x*, *a*), jinak nedojde k realizaci multiplexoru, ale paměťového obvodu typu „latch“.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux16_1 is
   port (
      x : in std_logic_vector (15 downto 0);
      a : in std_logic_vector (3 downto 0);
      y : out std_logic
   );
end mux16_1;
```

```
architecture process_indexace of mux16_1 is
begin
   -- proces se spousti pri zmene signalu x nebo a
   process (x, a)
   begin
      y <= x(to_integer(unsigned(a)));
   end process;
end process_indexace;
```

Výpis 8.4 Šestnáctikanálový multiplexor pomocí procesu a indexace vektoru

Další výpis kódu ukazuje možnou realizaci skupinového čtyřkanálového multiplexoru pomocí procesu a **case**. V citlivostním seznamu procesu musí být opět uvedeny všechny vstupní vektory – jak řídicí vektor *a*, tak datové vektory *b*, *c*, *d*, *e,* jinak nedojde k realizaci multiplexoru, ale paměťového obvodu typu „latch“.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux4_4 is
   port (
      b : in std_logic_vector (3 downto 0);
      c : in std_logic_vector (3 downto 0);
      d : in std_logic_vector (3 downto 0);
      e : in std_logic_vector (3 downto 0);
      a : in std_logic_vector (1 downto 0);
      y : out std_logic_vector (3 downto 0)
   );
end mux4_4;

architecture process_case of mux4_4 is
begin
   process (a, b, c, d, e)
   begin
      case a is
         when "00" => y <= b;
         when "01" => y <= c;
         when "10" => y <= d;
         when "11" => y <= e;
         when others => null;
      end case;
   end process;
end process_case;
```

*Výpis 8.5 Skupinový čtyřkanálový multiplexor pomocí procesu a **case***

Další výpis kódu pak realizuje skupinový čtyřkanálový multiplexor pomocí přiřazení s konstrukcí **with-select**.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux4_4 is
   port (
      b : in std_logic_vector (3 downto 0);
```

```
        c : in std_logic_vector (3 downto 0);
        d : in std_logic_vector (3 downto 0);
        e : in std_logic_vector (3 downto 0);
        a : in std_logic_vector (1 downto 0);
        y : out std_logic_vector (3 downto 0)
    );
end mux4_4;

architecture mux_with_select of mux4_4 is
begin
    with a select   -- vytvori multiplexor 1 z 4
        y <= b when "00",
           c when "01",
           d when "10",
           e when others;
end mux_with_select;
```

*Výpis 8.6 Skupinový čtyřkanálový multiplexor pomocí konstrukce **with-select***

8.3 Demultiplexor

Demultiplexor je opakem multiplexoru. Má jeden vstupní kanál a několik výstupních kanálů. Na *obr. 8.6* je jeho schematické znázornění. Adresou v binárním kódu je vybrán výstupní kanál, na který je převeden signál ze vstupního kanálu. Na všech ostatních výstupech je stav 0.

Následující výpisy kódu postupně ukazují realizaci dvoukanálového demultiplexoru, šestnáctikanálového demultiplexoru a skupinového čtyřkanálového demultiplexoru. První výpis kódu realizuje dvoukanálový demultiplexor pomocí přiřazení s konstrukcí **when-else**.

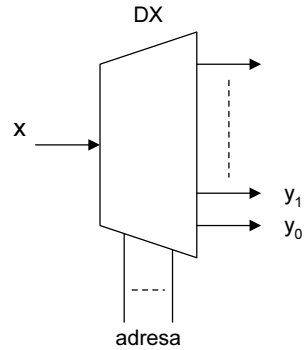
```
library ieee;
use ieee.std_logic_1164.all;

entity dmx1_2 is
    port (
        x : in std_logic;
        a : in std_logic;
        y0, y1 : out std_logic
    );
end dmx1_2;

architecture dmx_when_else of dmx1_2 is
begin
    y0 <= x when a = '0' else '0';
    y1 <= x when a = '1' else '0';
end dmx_when_else;
```

*Výpis 8.7 Dvoukanálový demultiplexor pomocí přiřazení s konstrukcí **when-else***

Další výpis kódu realizuje šestnáctikanálový demultiplexor pomocí procesu a indexace bitového vektoru s typovou konverzí. V citlivostním seznamu procesu musí být uvedeny oba vstupní vektory (*x*, *a*) jinak nedojde k realizaci demultiplexoru ale obvodu typu „latch“.



Obr. 8.6 Vstupy a výstupy demultiplexoru

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dmx1_16 is
   port (
      x : in std_logic;
      a : in std_logic_vector (3 downto 0);
      y : out std_logic_vector (15 downto 0)
   );
end dmx1_16;

architecture a_dmx1_16 of dmx1_16 is
begin
   process (x, a)
   begin
      y <= (others => '0');
      y(to_integer(unsigned(a))) <= x;
   end process;
end a_dmx1_16;
```

Výpis 8.8 Šestnáctikanálový demultiplexor pomocí procesu a indexace vektoru

Další výpis kódu pak realizuje skupinový čtyřkanálový demultiplexor pomocí přiřazení s konstrukcí **when-else**.

```
library ieee;
use ieee.std_logic_1164.all;

entity dmx4_4 is
   port (
      x : in std_logic_vector (3 downto 0);
      a : in std_logic_vector (1 downto 0);
      b : out std_logic_vector (3 downto 0);
      c : out std_logic_vector (3 downto 0);
      d : out std_logic_vector (3 downto 0);
      e : out std_logic_vector (3 downto 0)
   );
end dmx4_4;
```

```
architecture a_dmx4_4 of dmx4_4 is
begin
   b <= x when a = "00" else "0000";
   c <= x when a = "01" else "0000";
   d <= x when a = "10" else "0000";
   e <= x when a = "11" else "0000";
end a_dmx4_4;
```

Výpis 8.9 Skupinový čtyřkanálový demultiplexor realizovaný přiřazeními s konstrukcí ***when-else***

8.4 Prioritní kodér

Opakem dekodéru je **kodér**. Převádí kód *1zN* na kód binární. Je ale nutné zajistit, aby na jeho vstupech byl vždy právě jeden signál aktivní a ostatní neaktivní (tak jako je tomu na výstupech dekodéru). To zpravidla zajistit nelze a proto se častěji používá prioritní kodér.

U prioritního kodéru je přípustné, aby **více než jeden** vstupní signál byl aktivní (aktivní stav předpokládejme jako 1). Vstupům je přiřazena priorita. Nejjednodušší a nejčastější je priorita odstupňovaná **podle připojení**. Ta je pro každý vstup pevně stanovena a nelze ji měnit. Můžeme např. stanovit prioritu vstupu s číslem 0 jako nejnižší, vstupu s číslem 1 jako vyšší, atd. až postupně prioritu vstupu s nejvyšším číslem jako nejvyšší (ale někdy se naopak vstupu s číslem 0 přiřazuje nejvyšší priorita – je třeba se seznámit s dokumentací k obvodu).

Tab. 8.2 a *obr. 8.7* ukazuje příklad prioritního kodéru s 8 vstupy (x) v kódu *1z8* a třemi výstupy (y) v binárním kódu. Samostatný výstup z slouží k informování o tom, že je alespoň jeden vstupní signál aktivní – jsou-li totiž na všech vstupech hodnoty 0, nemá binární kód na výstupech $y_2y_1y_0$ smysl a hodnoty lze považovat za neurčené.

Při 8 vstupních proměnných by pravdivostní tabulka měla mít plných 256 řádků. Lze ji však drasticky zjednodušit prostou úvahou. Je-li aktivní vstup s nejvyšší prioritou (x_7), musí být na výstupech stav 111 (= 7) bez ohledu na vstupy s nižší prioritou – tedy lze doplnit v prvém řádku neurčené stavy. Signál $x_6 = 1$ se uplatní jen při neaktivním vstupu s vyšší prioritou $x_7 = 0$ a na výstupech pak bude stav 110 (= 6) bez ohledu na vstupy s ještě nižší prioritou. Tak lze uvažovat dále. Při všech vstupech neaktivních jsou výstupy y neurčené, ale zpravidla se doplňují na nuly. Tabulka se tak zkrátila jen na devět řádků.

Pro jednotlivé výstupní proměnné platí vztahy:

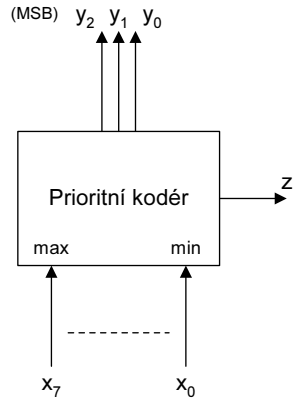
$$y_2 = x_7 + \overline{x}_7 x_6 + \overline{x}_7\overline{x}_6 x_5 + \overline{x}_7\overline{x}_6\overline{x}_5 x_4$$
$$y_1 = x_7 + \overline{x}_7 x_6 + \overline{x}_7\overline{x}_6\overline{x}_5\overline{x}_4 x_3 + \overline{x}_7\overline{x}_6\overline{x}_5\overline{x}_4\overline{x}_3 x_2$$
$$y_0 = x_7 + \overline{x}_7\overline{x}_6 x_5 + \overline{x}_7\overline{x}_6\overline{x}_5\overline{x}_4 x_3 + \overline{x}_7\overline{x}_6\overline{x}_5\overline{x}_4\overline{x}_3\overline{x}_2 x_1$$
$$z = x_7 + x_6 + x_5 + x_4 + x_3 + x_2 + x_1 + x_0$$

Schéma obvodu nebude složité a s uplatněním skupinové minimalizace se ještě dále zjednoduší.

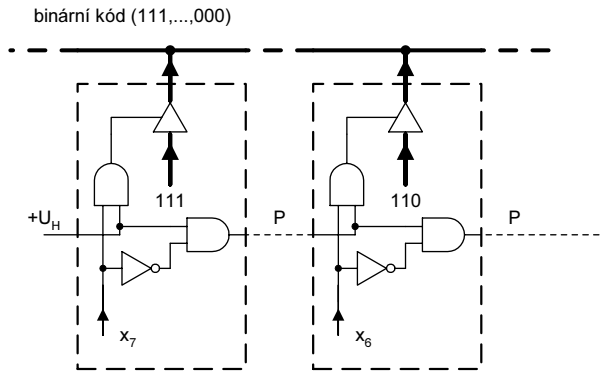
Výše uvedený prioritní kodér je „centralizovaný“ v tom smyslu, že všechny vstupní signály jsou svedeny na vstupy jednoho obvodu, a všechny výstupní signály vznikají v tomtéž obvodu. Prioritní kodér je ale možné sestavit ještě jinak, s využitím decentralizovaných obvodů – viz *obr. 8.8*.

Tab. 8.2 Pravdivostní tabulka prioritního kodéru

Vstupy								Výstupy (MSB)			
x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	y_2	y_1	y_0	z
1	-	-	-	-	-	-	-	1	1	1	1
0	1	-	-	-	-	-	-	1	1	0	1
0	0	1	-	-	-	-	-	1	0	1	1
0	0	0	1	-	-	-	-	1	0	0	1
0	0	0	0	1	-	-	-	0	1	1	1
0	0	0	0	0	1	-	-	0	1	0	1
0	0	0	0	0	0	1	-	0	0	1	1
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0	-	-	-	0



Obr. 8.7 Vstupy a výstupy prioritního kodéru



Obr. 8.8 Prioritní řetězec