ČÍSLICOVÁ ELEKTRONIKA

ONDŘEJ NOVÁK A KOLEKTIV AUTORŮ

Bibliografická reference těchto skript:
NOVÁK, O. a kol. *Číslicová elektronika*. 1. vydání. Liberec: Technická univerzita v Liberci, Fakulta mechatroniky, informatiky a mezioborových studií, 2014. ISBN 978-80-7494-137-5.
DOI: 10.15240/tul/002/2014-11-004

OBSAH

1 Úvod

Tato skripta jsou určena především pro studenty Technické univerzity v Liberci a měla by být chápána jako jedna z mnoha forem podpory výuky předmětů zaměřených na problematiku číslicové elektroniky. V žádném případě se nejedná o vyčerpávající sbírku informací z této oblasti, nezabýváme se ani fyzikou popisovaných dějů, ani základy analogové elektroniky, která s číslicovou elektronikou velmi úzce souvisí (viz [DOL14]); pochopitelně nejaktuálnější přehled je součástí přednášek příslušných předmětů. Je třeba také říci, že každý z námi zajišťovaných předmětů využívá různě velikou podmnožinu tohoto textu, a proto kompletní skripta nejsou v celé své šíři určena všem studentům; osobní zvídavosti však nehodláme nijak bránit... Pokud při svém studiu najdete nějaké nepřesnosti, překlepy atp., kontaktujte libovolného z autorů, abychom mohli zajistit rychlou nápravu.

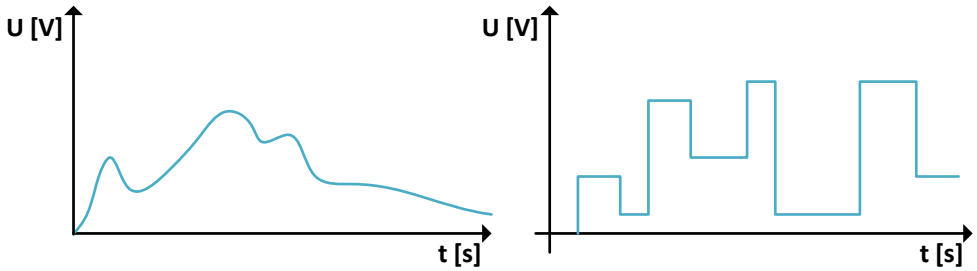
Úvodní kapitoly skript se zabývají základními pojmy s oblasti číslicové elektroniky, definicí logických stavů a operací, číselnými soustavami atp. Následující kapitola popisují základní kombinační obvody, jejich technologickou realizaci a metodiku návrhu těchto obvodů; dále jsou popsány sekvenční obvody, jejich vlastnosti a metodika jak asynchronních, tak synchronních obvodů. Následující kapitoly pak popisují složitější obvody jako jsou paměti, zakázkové obvody a základní přehled z oblasti mikroprocesorů. Oblast na pomezí číslicové a číslicové elektroniky, tedy převodníky, je popsána v již zmiňovaných skriptech [DOL14].

Závěrem této předmluvy bych rád poděkoval všem, kteří sbírali jednotlivé střípky textu a scelovali je do kompaktních celků kapitol.

2 ZÁKLADNÍ POJMY

Pro usnadnění komunikace mezi návrháři v oboru číslicové techniky bylo třeba definovat základní pojmy, zavést konvence a normy, které určují, jaké fyzikální veličiny a hodnoty jsou přiřazeny logickým veličinám, a jak se s nimi pracuje.

V elektronice používáme dva základní typy signálů. Vedle analogových signálů jsou to signály číslicové (digitální). Analogové signály jsou signály, které se mění spojitě (plynule) v čase. Naproti tomu digitální signály mění svou úroveň nespojitě (skokově). Příklady obou typů signálů můžeme vidět na **Obr. 1**. Číslicové signály jsou vlastně řadou impulsů, která mění nespojitě v čase své úrovně.



Obr. 1 Spojitý (vlevo) a nespojitý signál

2.1 LOGICKÉ STAVY

V číslicové technice pracujeme s fyzikálními veličinami, které je možno při určité míře zjednodušení popsat dvěma stavy. Tyto veličiny pro nás tedy představují proměnné nabývající dvou stavů, hovoříme o binárních proměnných. Příkladem může být obvod tvořený spínačem, žárovkou a napájecím zdrojem (např. Obr. 3). Fyzikální veličinou je potom poloha jazýčku spínače, binární proměnnou můžeme nazvat např. SPÍNAČ, stavy této proměnné pak jsou: ZAPNUTO a VYPNUTO. Jiným příkladem je transistor, který v číslicových obvodech přechází mezi dvěma stavy: UZAVŘENÍ a SATURACE.

Logickou konvencí je v uvedeném příkladu spínače to, že stavy ZAPNUTO a VYPNUTO nahradíme logickými hodnotami ‘nepravda’ a ‘pravda‘, symboly '0' a '1' nebo anglickými výrazy 'false' a 'true'. Někdy pro zdůraznění faktu, že pracujeme s logickými veličinami, místo '0' a '1' píšeme log. 0 a log. 1. Přiřazení mezi stavy a logickými hodnotami je libovolné.

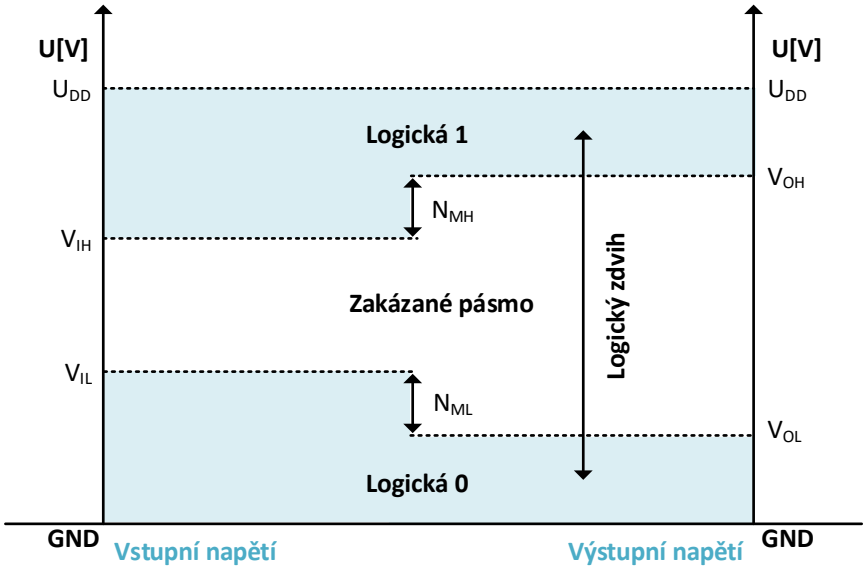
Stavu ZAPNUTO může odpovídat log 0 i log 1. Častěji se však volí přiřazení: ZAPNUTO = '1' a VYPNUTO = '0'. V případě, že logickou hodnotu přiřazujeme k úrovni napětí, potom v pozitivní logice volíme přiřazení: nižší napětí = '0' a vyšší napětí = '1'. V negativní logice přiřazujeme napětí k logickým hodnotám opačně. V anglické terminologii označujeme nízkou úroveň napětí zkratkou L (Low) a vysokou úroveň napětí zkratkou H (High).

V technické praxi je třeba bezpečně rozlišit jednotlivé binární stavy. Typické přiřazení napěťových úrovní je znázorněno na Obr. 2. Na tomto obrázku vidíme, že mezi typickými

napěťovými úrovněmi pro log. 0 a log. 1 je pásmo zakázaných napěťových úrovní. Toto pásmo se nazývá logický zdvih a šířka tohoto pásma určuje míru bezpečnosti rozpoznání logických úrovní.

2.2 LOGICKÉ OPERACE

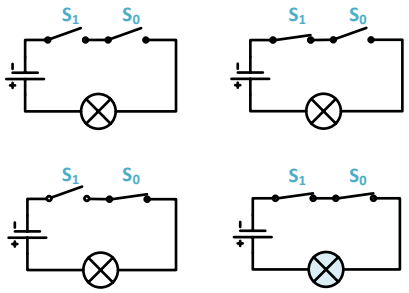
Pro binární proměnné je možno definovat logické operace. Tyto operace se zpravidla popisují pomocí logických operátorů (log. součet, log. součin, negace, …) nebo je můžeme popsat tzv. pravdivostní tabulkou.



Obr. 2 : Přiřazení napěťových úrovní binárním hodnotám

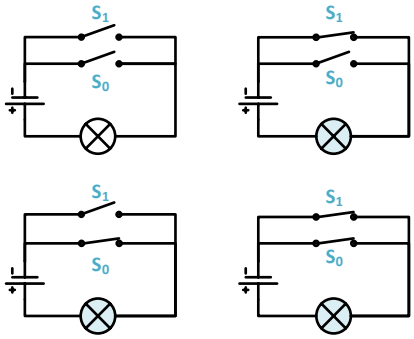
Obr. 3 a Obr. 4 ukazují obvod se dvěma spínači zařazenými a odpovídající pravdivostní tabulku. V tomto příkladu jsou dvě proměnné vstupní (S_0 a S_1) a jedna proměnná výstupní (Žárovka).

Pravdivostní tabulka udává vztah mezi vstupními a výstupními proměnnými, tj. je předpisem, který určuje, jak je možno výstupní proměnnou řídit pomocí vstupů. Logické operace mezi proměnnými S_0 a S_1 jsou logický součin (Obr. 3) a logický součet (Obr. 4). Obvod spínačů a žárovky potom modeluje logickou operaci součinu (součtu) a my o něm hovoříme jako o logickém členu. Logické členy tvoří logický obvod.



S1	S2	Žárovka
0	0	0
0	1	0
1	0	0
1	1	1

Obr. 3 : Obvodové schéma realizující funkci logického součinu (AND) a pravdivostní tabulka



S_0	S_1	Žárovka
0	0	0
0	1	1
1	0	1
1	1	1

Obr. 4 : Obvodové schéma realizující funkci logického součtu (OR) a pravdivostní tabulka

2.3 ČÍSELNÉ SOUSTAVY

S číslicovými signály je třeba provádět nejenom logické ale i aritmetické operace. Abychom porozuměli těmto operacím, uvedeme zde stručné vysvětlení způsobu převodů mezi desítkovou, binární a hexadecimální soustavou, které se v číslicových obvodech používají nejčastěji. Libovolné kladné číslo F(Z) lze zapsat pomocí mnohočlenu ve tvaru:

$$F(Z) = \sum_{i=0}^{m-1} a_i Z^i$$

Kde F(Z) je číslo vyjádřené v číselné soustavě o základu Z, ai jsou číselné koeficienty, m je počet řádových míst. Pro jednoduchost v číselných soustavách zapisujeme pouze koeficienty ai. V praxi potom často řešíme problém, který koeficient vyjadřuje nejvyšší řád a který nejnižší. Zpravidla se řídíme konvencí, která říká, že nejvyšší řád je vlevo a nejnižší vpravo. U dvojkové soustavy nejvyšší řád označujeme jako MSB (Most Significant Bit) a nejnižší řád jako LSB (Least Significant Bit).

Tab. 1 : Vyjádření čísel od 0 do 15 v různých soustavách

Číslo (dekadicky)	Základ 2	Základ 8	Základ 16
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Abychom rozlišili jednotlivé soustavy mezi sebou, používáme v případech, kdy by mohlo dojít k nejasnosti, o kterou soustavu se jedná, označení (2) nebo b pro dvojkovou (binární) soustavu, (8) nebo o pro osmičkovou (oktalovou) soustavu, (10) nebo d pro desítkovou (dekadickou) soustavu a (16) nebo h pro šestnáctkovou (hexadecimální) soustavu. Příklady čísel: 011(2), 011b, 47(8), 0A23(16), 0A123h.

Př. 1

Vyjádřete číslo 275(10) binárně.

Číslo v desítkové soustavě 275(10) je možno vyjádřit jako $2\times10^2 + 7\times10^1 + 5\times10^0$. Číslo 1101(2) ve dvojkové soustavě vyjadřuje hodnotu $1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$, neboť předpokládáme, že MSB je vlevo.

2.4 ZÁPORNÁ ČÍSLA

V případě, kdy chceme rozlišit kladná a záporná čísla ve dvojkové soustavě použijeme znaménko mínus stejně jako v soustavě dekadické. Pro kódování znaménka v počítači se používají různé způsoby. Nejčastěji jsou to tyto tři:

- Vyhrazení znaménkového bitu
- Přičtení konstanty
- Pomocí dvojkového doplňku

V případě, kdy jeden bit (většinou ten nejvýznamnější) představuje znaménko, ostatní bity představují absolutní hodnotu daného čísla. Konvence říká, že v případě, že znaménkový bit je roven jedné, jedná se o záporné číslo a je-li roven nule, jedná se o číslo kladné. Znaménkový bit nám

snižuje rozsah čísel, která můžeme zakódovat daným počtem bitů. Např. pomocí osmi bitů můžeme vyjádřit čísla v rozsahu -127 až 127 (přičemž máme kladnou a zápornou nulu).

Přičtení konstanty

Při využití tohoto způsobu kódování záporných binárních čísel dochází k posouvání nuly tím, že ke každému číslu přičteme vhodnou konstantu. Např. pro čísla v rozsahu -127 až 128 bychom volili konstantu 127.

Dvojkový doplněk

Dvojkový doplněk binárního čísla získáme jako jeho inverzi zvětšenou o jedničku. Kódování záporných čísel pomocí dvojkového doplňku nám umožňuje snadno provádět odčítání ve dvojkové soustavě (přičtením záporného čísla).

Rozsah čísel, který je možné pomocí dvojkového doplňku zakódovat je (-2^{n-1}) až $(2^{n-1} - 1)$, to znamená, že pomocí např. 4 bitů můžeme zakódovat čísla v rozsahu: -8 až 7, pomocí 8mi bitů čísla v rozsahu: -128 až 127 atd.

Tab. 2 Vyjádření čísel ve dvojkovém doplňku

AA	BB
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

Při převodu kladného čísla do dvojkového doplňku musíme nejprve kladné binární číslo doplnit zleva nulami na správný počet bitů (např. číslo 510 = 1012 doplníme na 01012). V případě že tak neučiníme, nedostaneme správnou hodnotu.

Př. 2

Vypočtěte dvojkový doplněk k číslu 11102 =1410.
Číslo 11102 nejprve doplníme na potřebný počet bitů (pro vyjádření čísla -14 potřebujeme 5 bitů) a tím dostaneme 01110. Inverze tohoto čísla je 10001. Přičtením jednotky v LSB dostaneme číslo $10010_2 = -14_{10}$.

Převody mezi číselnými soustavami

Přepočet čísla z libovolné soustavy do soustavy se základem 10 provedeme dosazením do mnohočlenu, uvedeného výše.

Př. 3

Převeďme číslo 11001(2) do dekadické soustavy
11001(2) = 1×24 +1×23 + 0×22 + 0×21 + 1×20 = 16 + 8 + 1 = 25(10).

Přepočet z desítkové soustavy do ostatních soustav se provádí pomocí algoritmu postupného dělení čísla základem nové soustavy.

Př. 4

Převeďme číslo 25(10) do dvojkové soustavy.
Řešení:

25 : 2 = 12,	*zbytek 1*	*…*	a_0	*= 1*
12 : 2 = 6,	*zbytek 0*	*…*	a_1	*= 0*
6 : 2 = 3,	*zbytek 0*	*…*	a_2	*= 0*
3 : 2 = 1,	*zbytek 1*	*…*	a_3	*= 1*
1 : 2 = 0,	*zbytek 1*	*…*	a_4	*= 1*

Při dělení získáváme zbytky, které reprezentují výsledné bity binárního čísla postupně od LSB. Postupné dělení provádíme tak dlouho, až dostaneme nulový výsledek částečného dělení. Výsledek: 25(10) = 11001(2).

Přepočet mezi dvojkovou, osmičkovou a šestnáctkovou soustavou. Využíváme toho, že 8 i 16 jsou mocninou čísla 2. Z toho vyplývá, že jeden řád osmičkové soustavy je representován třemi místy dvojkovými, jedno místo šestnáctkové čtyřmi místy dvojkovými a opačně.

Př. 5

Převeďte číslo 1010111111010001(2) do šestnáctkové a osmičkové soustavy.
Pro převod do šestnáctkové soustavy rozdělíme číslo na čtveřice od nejnižšího řádu a čtveřice nahradíme šestnáctkovou cifrou.

0001	*0101*	*1111*	*1101*	*0001*
1	*5*	*F*	*D*	*1*

Výsledek: 1010111111010001(2) = 15FD1(16)

Pro převod do osmičkové soustavy rozdělíme číslo na trojice od nejnižšího řádu a ty pak nahradíme osmičkovou cifrou

010	*101*	*111*	*111*	*010*	*001*
2	*5*	*7*	*7*	*2*	*1*

Výsledek:1010111111010001(2) = 257721(8).

2.5 SČÍTÁNÍ A ODČÍTÁNÍ VE DVOJKOVÉ SOUSTAVĚ

a) Sčítání: Při sčítání se příslušné koeficienty a_i sčítají obdobně jako v desítkové soustavě:

00011011 (2)

00110010 (2)

01001101 (2)

b) Odčítání: Ručně provádíme odčítání v číselných soustavách tak, jak jsme zvyklí ze soustavy desítkové, respektujeme při tom změny řádu v dané soustavě.

01001011 (2)

-00110010 (2)

00011001 (2)

Výpočetní technika používá pro odčítání převodu na přičítání dvojkového doplňku menšitele. Dvojkový doplněk k menšiteli 00110010(2) získáme jako jeho inverzi (11001101) zvětšenou o 1 (11001110). Dvojkový doplněk reprezentuje záporné číslo o stejné absolutní hodnotě, jako bylo číslo původní. Dvojkový doplněk přičteme k menšenci:

01001011 (2)

11001110 (2)

00011001 (2)

Nula v řádu MSB signalizuje, že výsledek odečítání je kladné číslo. V případě, že výsledek odečítání by byl záporný, získali bychom výsledek s MSB rovným jedné. V případě, že bychom chtěli získat absolutní hodnotu tohoto záporného čísla, určili bychom ji jako dvojkový doplněk výsledku.

Př. 6

Ve dvojkové soustavě proveďte operaci odečtení čísel 1-7.
Pro binární kódování čísel v rozsahu od –7 do +7 využijeme 4 bitů. Při využití dvojkového doplňku čísla 7 provedeme následující výpočet:

0001 (2)
1001 (2)
1010 (2)

Výsledek 1010 je vyjádřením čísla –6. Absolutní hodnotu získáme jako jeho inverzi (0101) a přičtení jednotky (0110).

2.6 DALŠÍ POUŽÍVANÉ KÓDY

Doposud jsme hovořili o binární, oktalové, dekadické a hexadecimální číselné soustavě. Tyto soustavy spolu s pravidly, která říkají, jaké informace jsou přiřazeny jednotlivým číslům, nazýváme kódy. Příkladem kódu může být tzv. doplňkový kód, který jsme využili při odečítání binárních čísel.

Pro snazší práci s čísly desítkové soustavy byl vytvořen BCD (Binary Coded Decimal) kód. Tento kód slouží k zobrazení tzv. desítkových číslic, tj. čísel 0, 1, …, 9. Tento kód je 4bitový, neboť k zakódování každé z deseti číslic vyžaduje 4 bity. V Tab. 3 je tento kód uveden.

Tab. 3 Tabulka BCD kódu

Znak	Obraz			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
0	0	0	0	0

V dalším textu popíšeme ještě kód Grayův. Tento kód má tu vlastnost, že sousední čísla se liší pouze v jednom bitu. Této výhodné vlastnosti je využito např. při sestavování Karnaughovy mapy v kapitole (3.1.3). Tabulka Grayova kódu pro čísla od 0 do 15 je uvedena v Tab. 4.

Tab. 4 Grayův kód

Znak	Obraz			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	0	1	0
4	0	1	1	0
5	0	1	1	1
6	0	1	0	1
7	0	1	0	0
8	1	1	0	0
9	1	1	0	1
10	1	1	1	1
11	1	1	1	0
12	1	0	1	0
13	1	0	1	1
14	1	0	0	1
15	1	0	0	0

Kód 1 z N má tu vlastnost, že obraz je tvořen N-1 nulami a jednou jedničkou, která je na pozici dané vstupním znakem. Například číslo 3 se zobrazí v kódu 1 z 8 jako vektor 00010000. Kód se využívá například u dekodéru popsaného v Tab. 5.

Tab. 5 Kód 1 z N

Znak	Obraz			
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0

Důležitou skupinou kódů jsou alfanumerické kódy. Tyto kódy zobrazují abecedně číslicovou informaci pomocí binárních čísel. Alfanumerické kódy se začaly používat pro přenos zpráv a byly pro tento účel normalizovány. Nejčastěji se používají kódy osmibitové, tzn. každý znak, který se přenáší, je kódován osmi bity. Velmi rozšířený je kód ASCII (viz Tab. 6), který je uzpůsoben k přenosu anglické abecedy a dalších běžných znaků, je nevhodný k přenosu znaků české abecedy. Z tohoto důvodu byl upraven pro přenos těchto českých znaků, čímž vznikl kód Kamenických, později LATIN 2.

Tab. 6 Tabulka ASCII znaků

Kód	Význam	Kód	Symbol	Kód	Symbol	Kód	Symbol
0	Null character	32		64	@	96	`
1	Start of Header	33	!	65	A	97	a
2	Start of Text	34	"	66	B	98	b
3	End of Text	35	#	67	C	99	c
4	End of Transmission	36	$	68	D	100	d
5	Enquiry	37	%	69	E	101	e
6	Acknowledgement	38	&	70	F	102	f
7	Bell	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Horizontal Tab	41	)	73	I	105	i
10	Line feed	42	*	74	J	106	j
11	Vertical Tab	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	Shift Out	46	.	78	N	110	n
15	Shift In	47	/	79	O	111	o
16	Data link escape	48	0	80	P	112	p
17	Device control 1	49	1	81	Q	113	q
18	Device control 2	50	2	82	R	114	r
19	Device control 3	51	3	83	S	115	s
20	Device control 4	52	4	84	T	116	t
21	Negative acknowledgement	53	5	85	U	117	u
22	Synchronous idle	54	6	86	V	118	v
23	End of transmission block	55	7	87	W	119	w
24	Cancel	56	8	88	X	120	x
25	End of medium	57	9	89	Y	121	y
26	Substitute	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	File separator	60	<	92	\	124	|
29	Group separator	61	=	93	]	125	}
30	Record separator	62	>	94	^	126	~
31	Unit separator	63	?	95	_	127	DEL

2.7 KÓDOVÁNÍ ČÍSEL S DESETINNOU ŘÁDOVOU ČÁRKOU

Čísla s pevnou desetinnou řádovou čárkou

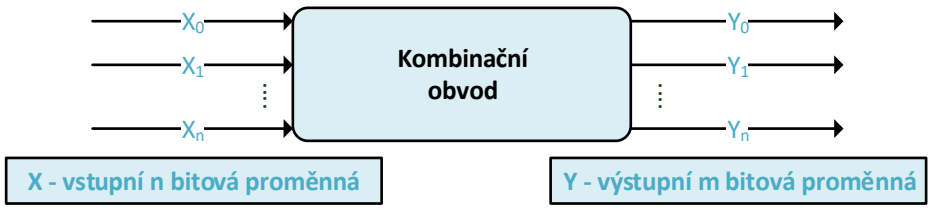
Čísla s plovoucí desetinnou řádovou čárkou

3 LOGICKÉ OBVODY

Logické obvody mohou být založeny na různém fyzikálním principu. Mohou být mechanické, pneumatické, elektrické, elektronické, magnetické, optické, nejnověji také obvody založené na změně spinu elektronu. V tomto textu se zabýváme, elektronickými logickými obvody, jedná se totiž o nejčastěji používaný fyzikální princip. V případě, že chceme měřit jinou fyzikální veličinu (např. teplotu, tlak, polohu, …) popřípadě chceme vytvořit neelektronickou akční veličinu (síla, poloha, teplota, osvětlení, …) používáme snímače a převodníky, které umožní zpracování informací elektronickými logickými obvody. Z hlediska chování logické obvody rozdělujeme na kombinační a sekvenční. Sekvenční obvody pak mohou být dále rozděleny na asynchronní a synchronní. V následujícím textu se budeme nejprve zabývat kombinačními obvody.

3.1 KOMBINAČNÍ LOGICKÉ OBVODY

Ideální kombinační obvod nemá žádnou vnitřní paměť. Odezva ideálního kombinačního obvodu v určitém časovém okamžiku je podmíněna pouze těmi hodnotami, které jsou v uvažovaném okamžiku na vstupech obvodu a je okamžitá. U reálného kombinačního obvodu je třeba uvažovat zpoždění odezvy na změnu vstupů. Při dodržení návrhových pravidel pro konstrukci obvodů však můžeme toto zpoždění zanedbat bez toho, aby došlo ke změně předpokládané funkce.



Obr. 5 Kombinační obvod – výstupní proměnná je jednoznačně určena proměnnou vstupní

Pomocí kombinačních obvodů můžeme realizovat logické a aritmetické funkce, jejichž argumenty jsou vstupní proměnné a výsledky operací jsou dány výstupními proměnnými.

3.2 ZÁKLADNÍ LOGICKÉ FUNKCE

Logický součin

Příklad obvodu, který realizuje funkci součinu a příslušná pravdivostní tabulka je na Obr. 3. Funkce může mít i větší počet operandů než dva. Funkční hodnota je rovna '1' pouze v případě, že na všechny vstupy přivedeme hodnotu '1'. Algebraicky je funkce vyjádřena pomocí symbolu tečka: (Y = A·B). Anglická zkratka funkce je AND.

Logický součet

Příklad obvodu, který realizuje funkci součtu a příslušná pravdivostní tabulka je na Obr. 4. Funkce může mít libovolný počet operandů ne menší než dva. Funkční hodnota je rovna '0' pouze

v případě, že na všechny vstupy přivedeme hodnotu: '0'. Algebraicky vyjadřujeme funkci pomocí znaménka plus: (Y = A+B). Anglická zkratka funkce je OR.

Negace

Negace je unární operace, to znamená, že má pouze jeden operand. Funkce negace mění hodnotu proměnné z '0' na '1' a opačně. Algebraicky vyjadřujeme funkci pomocí svislé čáry nad logickou proměnnou popř. symbolem apostrof (´,): (Y = A , Y = ‘A, Y = A´) Anglická zkratka funkce je INVERT, NON, nebo NOT. V grafické reprezentaci funkcí značíme negaci symbolem kroužku. Symbol kroužku může být v libovolné části schématu obvodu. Jestliže je umístěn na vstupu schématické značky, interpretujeme jej jako negaci příslušné vstupní proměnné, pokud je na výstupu jedná se o negaci výstupní proměnné.

Negovaný součin

Funkce může mít libovolný počet operandů ne menší než dva. Funkční hodnota je rovna '0' pouze v případě, že na všechny vstupy přivedeme hodnotu '1'. Algebraicky vyjadřujeme funkci pomocí tečky (popřípadě tečku vynecháme) a pomocí pruhu popř. (‘): (Y = A·B, Y = AB, Y = ‘(A·B)). Anglická zkratka funkce je NAND.

Negovaný součet

Funkce může mít libovolný počet operandů ne menší než dva. Funkční hodnota je rovna '1' pouze v případě, že na všechny vstupy přivedeme hodnotu '0'. Algebraicky vyjadřujeme funkci pomocí znaménka plus a pomocí pruhu popř. (‘): (Y = A+B , Y = ‘(A+B)). Anglická zkratka funkce je NOR.

Nonekvivalence (výhradní logický součet)

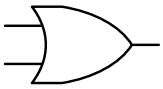
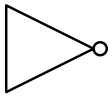
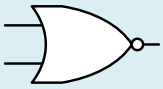
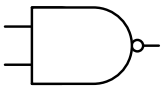
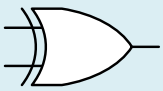
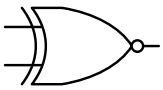
Funkce má dvě proměnné. Funkční hodnota je rovna '1' pouze v případě, že právě na jeden vstup přivedeme '1'. Algebraicky vyjadřujeme funkci pomocí znaménka ⊕: (Y = A⊕B). Anglická zkratka funkce je XOR.

Ekvivalence

Funkce má dvě proměnné. Funkční hodnota je rovna '0' pouze v případě, že právě na jeden vstup přivedeme '1'. Algebraicky vyjadřujeme funkci pomocí znaménka⊕ a pruhu: (Y = A⊕B). Anglická zkratka funkce je XNOR (EXCLUSIVE NOR).

Uvedené funkce je možno znázornit nejen algebraicky a slovně ale také i graficky. Tyto symboly jsou uvedeny v Tab. 7. Každý symbol podle ČSN má nalevo vstupy a napravo výstupy, uvnitř obdélníčků je vepsán symbol funkce. Pospojováním symbolů čarami je možno přehledně znázornit složitější funkce. V současnosti jsou u nás využívány i symboly odpovídající americkým standardům, které umožňují symbol libovolně natáčet, což může vést u složitějších schémat k větší přehlednosti.

Tab. 7 Přiřazení grafických symbolů podle ČSN a podle anglo-americké konvence jednotlivým logickým členům a jejich pravdivostní tabulky

Logický součet (OR)	≥1		a b y 0 0 0 0 1 1 1 0 1 1 1 1
Logický součin (AND)	&		a b y 0 0 0 0 1 0 1 0 0 1 1 1
Negace (NOT)	1		a y 0 1 1 0
Negovaný součet (NOR)	≥1		a b y 0 0 1 0 1 0 1 0 0 1 1 0
Negovaný součin (NAND)	&		a b y 0 0 1 0 1 1 1 0 1 1 1 0
Nonekvivalence (XOR)	=1		a b y 0 0 0 0 1 1 1 0 1 1 1 0
Ekvivalence (XNOR)	=1		a b y 0 0 1 0 1 0 1 0 0 1 1 1

Ve výše uvedených odstavcích jsme popsali nejužívanější logické funkce. Seznam však není úplný, např. pro dvě proměnné existuje 16 možných různých funkcí. Libovolnou funkci však můžeme vytvořit sestavením z několika základních funkcí. Takováto skupina funkcí se nazývá úplným souborem logických funkcí. Tvoří ji například dvojice funkcí: AND + NOT, OR + NOT nebo dokonce může být tvořena samostatnou funkcí jako například funkcí NAND nebo NOR. Tohoto poznatku se využívá u stavebnic integrovaných obvodů, kde je možno skládat jednotlivé integrované obvody obsahující logické obvody pouze s funkcí NAND tak, že pomocí nich realizujeme libovolné složitější zapojení. Symboly složitějších logických funkcí se vytvářejí podobně jako symboly funkcí základních. Symbolická značka je zpravidla svislými pruhy rozdělena na tři pole: pole vstupů, pole vlastní funkce a pole výstupů. Vstupní a výstupní pole je ještě rozděleno vodorovnými čarami na skupiny jednotlivých proměnných, které mají podobnou funkci (viz např. **Obr. 25**).

3.2.1 Booleova algebra

Z matematiky víme, že logické výrazy je možno upravovat pomocí zákonů Booleovy algebry. Tyto zákony je dobré znát, protože nám umožní převádět logické výrazy na takové, které mají požadované vlastnosti (například minimální počet součtových členů), realizovatelnost pomocí jednoho typu hradel atd. Připomeňme ještě, že v Booleově algebře jsou dvě základní binární operace - logický součet (+) a součin (·), unární operace negace (značeno pruhem) a dvě nulární operace (konstanty) - logická 0 a 1. Základní axiomy jsou uvedeny v Tab. 8, odvozené výrazy jsou uvedeny v Tab. 9. Uvedené zákony se s výhodou využívají pro optimalizaci logických výrazů, De Morganovy zákony umožňují převádět logické výrazy ze součinového tvaru na součtový a opačně (tj. např. převod z logického obvodu obsahujícího pouze hradla NOR na obvod obsahující pouze hradla NAND).

Tab. 8 Booleova algebra – Axiomy

Základní axiom	Součet	Součin
Komutativita	a +b=b+a	a·b=b·a
Asociativita	a+(b+c)=(a+b)+c	a(b·c)=(a·b)c
Distributivita	a+(b·c)=(a+b)(a+c)	a(b+c)=(a·b)+(a·c)
Neutralita 0 a 1	a+0=a	a·1=a
Vlastnosti komplementu	$a + \bar{a} = 1$	$a \cdot \bar{a} = 0$
Agresivita 0 a 1	a+1=1	a·0=0
Idempotence	a+a=a	a·a=a
Absorbce	a+a·b=a	a(a+b)=a

Tab. 9 Booleova algebra – odvozené zákony

Pravidlo	Příklad	
Dvojí negace	$\bar{\bar{a}} = a$	
Absorpce negace	$a + \bar{a} \cdot b = a + b$	$a(\bar{a} + b) = a \cdot b$
De Morgan	$\overline{(a + b)} = \bar{a} \cdot \bar{b}$	$\overline{a \cdot b} = \bar{a} + \bar{b}$
Consensus	$ab + \bar{a}c + bc = ab + \bar{a}c$	$(a + b)(\bar{a} + c)(b + c) = (a + b) + (\bar{a} + c)$

3.2.2 PRAVDIVOSTNÍ TABULKA

Logické funkce je možné vyjádřit kromě algebraického popisu také pravdivostní tabulkou. V této kapitole si vlastnosti pravdivostní tabulky rozebereme podrobněji.

V Tab. 10 je označen první sloupec jak stavový index. Ukazuje současně číslo řádku tabulky a binární hodnotu čísla tvořeného vstupními proměnnými, jestliže jsou seřazeny vzestupně (tj. bity s nejvyšší vahou nalevo). Druhý sloupec obsahuje všechny možné kombinace vstupních hodnot jednotlivých proměnných, třetí sloupec funkční hodnoty příslušné odpovídajícím hodnotám vstupů. Sloupec minterm nám ukazuje, jak je možno jednotlivým řádkům tabulky přiřadit logický výraz.

Tab. 10 Pravdivostní tabulka funkce f. Vstupní proměnné a, b, c. Bit c je MSB bitem.

Stavový index S	c	b	a	Funkční hodnota F(c,b,a)	Minterm PS
0	0	0	0	0	$\bar{c}\bar{b}\bar{a}$
1	0	0	1	1	$\bar{c}\bar{b}a$
2	0	1	0	1	$\bar{c}b\bar{a}$
3	0	1	1	0	$\bar{c}ba$
4	1	0	0	1	$c\bar{b}\bar{a}$
5	1	0	1	0	$c\bar{b}a$
6	1	1	0	1	$cb\bar{a}$
7	1	1	1	0	cba

Například jestliže tabulka definuje funkci nabývající hodnoty 1 v řádcích 1, 2, 4 a 6, potom z posledního sloupce můžeme odvodit algebraické vyjádření funkce f:

$$f = \bar{c}\bar{b}a + \bar{c}b\bar{a} + c\bar{b}\bar{a} + cb\bar{a}$$

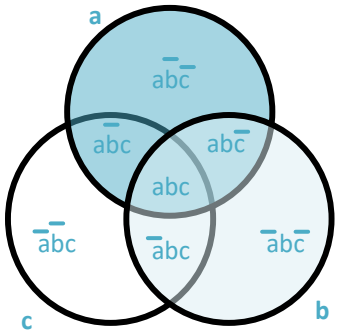
V pravdivostní tabulce se může vyskytovat symbol X a to jak ve sloupci vstupů tak funkčních hodnot. Jeho význam je „nedefinovaná hodnota“. Používáme jej tehdy, když na uvedeném vstupu, resp. výstupu nezáleží. Zápis pravdivostní tabulky s využitím X šetří místo. V Tab. 11 je stejná funkce jako v předchozí pravdivostní tabulce, vidíme však, že zápis je úspornější.

Tab. 11 Pravdivostní tabulka funkce f, využití nedefinovaných hodnot X.

Stavový index	C	B	A	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4,6	1	X	0	1
5,7	1	X	1	0

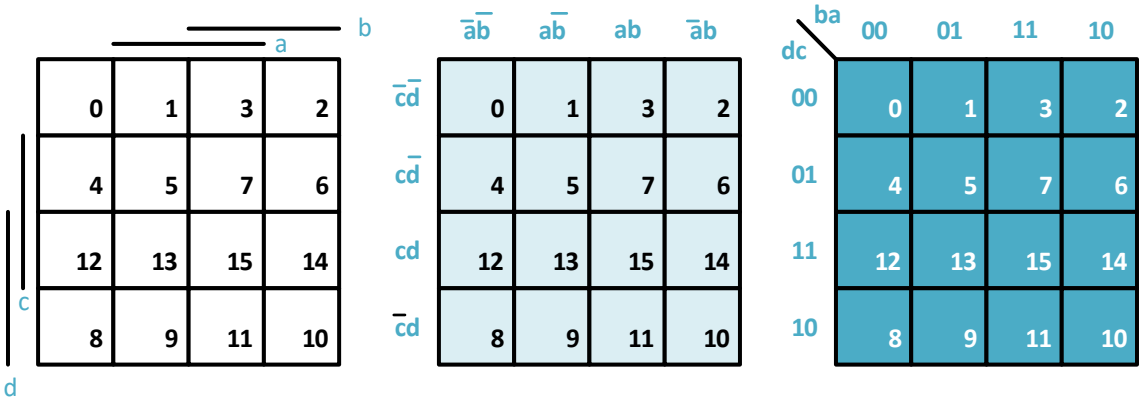
3.2.3 VENŮV DIAGRAM, KARNAUGHOVA MAPA

Přehledněji, než pomocí pravdivostní tabulky zobrazíme logickou funkci pomocí Vénových diagramů (**Obr. 6**) nebo pomocí mapy (Obr. 7). Vyšší přehlednost spočívá v tom, že v diagramu i v mapě jsou zobrazeny vedle sebe ty termy, které se liší v jedné proměnné. Jestliže pak je funkční hodnota zkoumané funkce rovna log. 1 v sousedních políčkách mapy nebo diagramu, můžeme vytvořit jednu společnou oblast, která zahrnuje oba termy, výsledný logický výraz je pak jednodušší.



Obr. 6 Vennův diagram funkce tří proměnných

Vennův diagram funkce tří proměnných je vyobrazen na Obr. 6. Jedná se o způsob grafického vyjádření příslušnosti prvků do množiny. Je tvořený uzavřenými křivkami, přičemž body uvnitř křivky představují prvky dané množiny a body venku prvky, které do množiny nepatří.



Obr. 7 Karnaughova mapa funkce o čtyřech proměnných

Na Obr. 7 je vyobrazena Karnaughova mapa funkce o čtyřech proměnných. U jednotlivých příkladů (mapy A, B a C) je využito rozdílného způsobu popisu mapy. Jednotlivé vstupní proměnné nabývají hodnotu log. 1 v těch sloupcích, resp. řádcích, u kterých je zakreslen pruh (A), proměnná není negována (B), nebo tam kde je sloupec resp. řádek pro danou proměnnou označen 1. V políčkách map jsou uvedeny příslušné stavové indexy funkce.

3.2.4 MINIMALIZACE LOGICKÝCH FUNKCÍ

V technické praxi se často snažíme, aby logická funkce byla vyjádřena co nejjednodušším logickým výrazem (snaha realizovat funkci co nejmenším počtem logických prvků). Procesu hledání tohoto výrazu říkáme minimalizace logického výrazu.

Nejprve je třeba stanovit kritéria minimalizace. V prvé řadě se snažíme zmenšit počet termů, ze kterých se výraz skládá. Dále je třeba minimalizovat počet vstupních proměnných, které tvoří jeden term, a nakonec se snažíme minimalizovat počet negací ve výsledném logickém výrazu. Minimálního výrazu můžeme dosáhnout buď pomocí algebraických úprav výchozího výrazu, nebo využít sousedních stavů k redukci počtu a velikosti termů. K nalezení sousedních stavů využijeme buď mapu nebo některý formální algoritmus, umožňující počítačové řešení problému (například algoritmus Quine-Mc Cluskey). Výsledkem minimalizace může být logická funkce ve tvaru logického součtu jednotlivých součinů tzv. MNDF (Minimální Normální Disjunktivní Forma), nebo součinu jednotlivých součtů tzv. MNKF (Minimální Normální Konjunktivní Forma). Tyto tvary logických výrazů pak mohou být dále upravovány na tvar realizovatelný pomocí základních logických obvodů (Tab. 7).

Př. 7 Úprava logického výrazu

Zjednodušte logický výraz pro funkci: $\bar{a}d + \bar{b}cd + a\bar{b}(c+d) + \bar{b}\bar{c}\bar{d}$***. Využijte algebraických úprav (zákonů Booleovy algebry).***

Řešení: Postupnou aplikací zákonů Booleovy algebry upravíme daný výraz na minimální formu.

Výchozí tvar	$\bar{a}d + \bar{b}cd + a\bar{b}(c+d) + \bar{b}\bar{c}\bar{d}$
Distributivní zákon	$\bar{a}d + \bar{b}cd + a\bar{b}c + a\bar{b}d + \bar{b}\bar{c}\bar{d}$
Zákon absorpce negace	$\{\bar{a}d + \bar{a}bd = d(\bar{a}+\bar{b})\}$
	$\bar{a}d + \bar{b}cd + a\bar{b}c + \bar{b}d + \bar{b}\bar{c}\bar{d}$
Absorpce negace	$\{\bar{b}d + \bar{b}\bar{c}\bar{d} = \bar{b}(d+\bar{c})\}$
	$\bar{a}d + \bar{b}cd + a\bar{b}c + \bar{b}d + \bar{b}\bar{c}$
Absorpce negace	$\{a\bar{b}c + \bar{b}\bar{c} = \bar{b}(\bar{c}+a)\}$
	$\bar{a}d + \bar{b}cd + a\bar{b} + \bar{b}d + \bar{b}\bar{c}$
Absorpce	$\bar{a}d + \bar{b}c + a\bar{b} + \bar{b}d + \bar{b}\bar{c}$
Concensus	$f = \bar{a}d + a\bar{b} + \bar{b}\bar{c}$

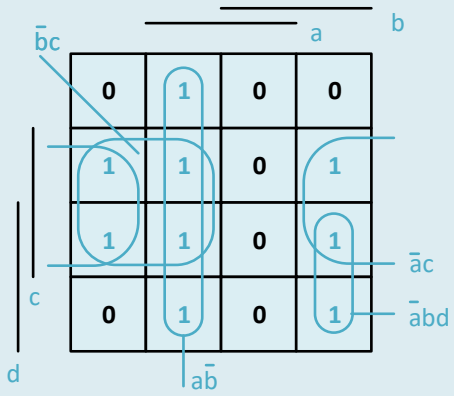
Výsledný logický výraz funkce f je minimální. Logický výraz, který jsme získali po aplikaci zákona o consensu, je minimálním vyjádřením funkce f.

Př. 8 Minimalizace log. výrazu pomocí Karnaughovy mapy

Minimalizujte logickou funkci f danou výčtem jedničkových stavů:

$$f = \sum(1, 4, 5, 6, 9, 10, 12, 13, 14)$$

Řešení: K minimalizaci použijeme Karnaughovu mapu, na které vyznačíme jedničkové stavy. Účelem použití mapy je nalezení sousedních stavů, tj. stavů, které se liší v jednom bitu. K libovolnému políčku mapy je sousední políčko to, které je vedle, pod nebo nad tímto políčkem a všechna políčka, která jsou ve stejném sloupci, resp. řádku na opačném konci, jestliže uvažované políčko je na kraji tabulky.



Obr. 8 Minimalizace do MNDF

Např. na Obr. 8 jsou k políčku 13 sousední políčka 5, 15, 9, a 12, k políčku 0 sousední políčka 1, 4, 8 a 2. Dále hledáme sousední dvojice políček k vybrané dvojici sousedních políček. Např. ke dvojici políček 0,8 je sousední dvojice 1,9 nebo 2,10 nebo 4,12. Ke každé sousední čtveřici je možné podobným postupem hledat sousední osmici atd. Smyslem tohoto hledání je nalezení dvojic, čtveřic, osmic, … políček, které obsahují funkční hodnotu 1.

*Při minimalizaci mají přednost větší skupiny sousedů, protože je možné je popsat jednodušším výrazem. V našem příkladu jsme označili čtveřice políček (**Obr. 8**), které je možno popsat výrazy $\bar{a}c, a\bar{b}, \bar{b}c$ a dvojici $\bar{a}bd$. Označené oblasti se mohou překrývat, všechny jedničky v mapě je však třeba pokrýt alespoň jednou oblastí. V případě, že pro některou jedničku není možné najít žádnou dvojici, je třeba příslušný výraz, popisující dané políčko, do výsledného vyjádření funkce zahrnout. V případě, že se v mapě vyskytují funkční hodnoty X, je možné příslušné políčko využít k vytvoření nějaké oblasti, nebo ve výsledném vyjádření funkce nemusí být zahrnuto. Výsledný tvar funkce je:*

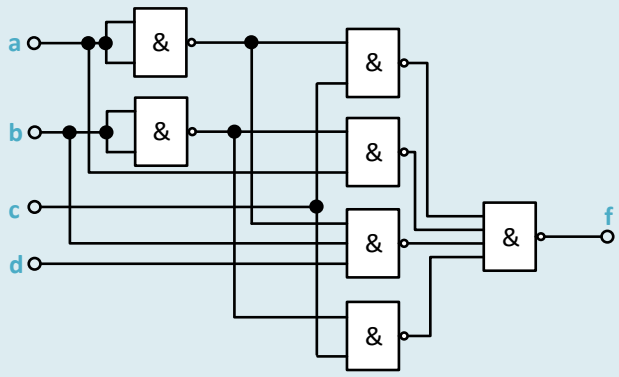
$$f = \bar{a}c + a\bar{b} + \bar{b}c + \bar{a}bd$$

Jestliže chceme výsledný výraz realizovat pouze s pomocí obvodů NAND, musíme dále použít De Morganovo pravidlo k další úpravě výsledného logického výrazu. Po úpravách dostaneme:

$$f = \overline{\bar{a}c \cdot a\bar{b} \cdot \bar{b}c \cdot \bar{a}bd}$$

Logické schéma vytváříme postupně. Pomocí invertorů nebo hradel NAND zapojených tak, aby realizovaly funkci invertoru, zajistíme negaci jednotlivých proměnných. (V uvedeném příkladu jsou to proměnné $\bar{a}, \bar{b}$). Prostřednictvím členů NAND s vhodným počtem vstupů realizujeme vnitřní negované logické součiny. (V příkladu jsou to negované součiny $\bar{a}c, a\bar{b}, \bar{a}bd$):

*Poslední výstupní člen NAND s vhodným počtem vstupů provede negovaný součin členů vytvořených podle předchozího bodu. Výsledné logické schéma je na **Obr. 9**.*



Obr. 9 Zapojení obvodu realizujícího funkci f

Úloha minimalizace není jednoznačná. V uvedeném příkladu bylo například možno místo pokrývání jedniček v Karnaughově mapě pokrývat nuly a výslednou proměnnou invertovat.

V praxi je návrhář zpravidla omezen ještě dalšími faktory, jako je maximální počet logických úrovní, kterými se v obvodu může signál šířit, maximálním možným počtem vstupů jednotlivých hradel, maximálním možným větvením signálu atd. S výhodou naopak v některých případech může využít tzv. skupinové minimalizace, kdy pro větší počet výstupů využije společnou část obvodu. V současnosti se pro návrh obvodů používá návrhových systémů, které automaticky vygenerují vhodnou vnitřní strukturu zapojení obvodů.

Př. 9 Minimalizace pomocí mapy

Použijte Karnaughovu mapu pro minimalizaci Booleových výrazů následujících funkcí zadaných logickým výrazem a pravdivostní tabulkou:

$X = \bar{a}\bar{b} + a\bar{b}\bar{c} + a\bar{b}c + abc$

$Y = \bar{a}\bar{b}\bar{c} + \bar{a}b\bar{c}d + a\bar{c}\bar{d} + a\bar{b}cd$

Logická funkce je zadána v pravdivostní Tab. 12:

Tab. 12 Logická funkce zadaná pravdivostní tabulkou

a	b	c	d	Z
0	0	0	0	1
0	0	0	1	0
0	0	1	0	X
0	0	1	1	0
0	1	0	0	1
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	X
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	X

3.2.5 Minimalizace logických funkcí metodou Quine–McCluskey

Minimalizace logických funkcí pomocí Karnaughových map (KM) je relativně rychlá a jednoduchá metoda, kterou lze provádět ručně na papíře. Tato metoda funguje velmi dobře pro 4 a méně proměnných. Minimalizaci pomocí KM lze použít i pro 5 proměnných, ale je méně přehledná a snadno se v ní udělá chyba. Alternativou, která je ekvivalentem minimalizace pomocí KM a umožňuje minimalizaci funkce o více než 5 proměnných je využití Quine–McCluskey (QM) algoritmu (někdy nazýván metoda přímých implikantů). QM je metoda určená pro minimalizaci booleovských funkcí, která byla vyvinuta W. V. Quinem a E. J. McCluskym v roce 1956. Tento algoritmus je prováděn pomocí tabulek, může tedy být prováděn ručně na papír, ale je také vhodný pro algoritmické zpracování pomocí počítače.

V následujícím textu si minimalizaci pomocí QM algoritmu vysvětlíme na jednoduchém příkladu.

Př. 10 Minimalizace Quine-McCluskey

Zadaní: Nalezněte MNDF následující funkce

$$\boldsymbol{f(d,c,b,a) = \sum(1,4,7,8,9,10,11,12,14,15)}$$

Nyní je třeba sestavit tabulky, s jejíž pomocí budeme zadanou funkci minimalizovat. V následujících krocích (a - i) je popsán postup pro sestavení těchto tabulek.

a) *Jako první si vytvoříme pravdivostní tabulku zadané logické funkce s tím, že vynecháme ty řádky, kde je hodnota funkce nulová.*

Tab. 13 Quine-McCluskey, krok a

Index	d	c	b	a
1	0	0	0	1
4	0	1	0	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
14	1	1	1	0
15	1	1	1	1

b) *Jako další si vytvoříme tabulku, kde rozdělíme jednotlivé řádky do skupin podle počtu jedniček*.

Tab. 14 Quine-McCluskey, krok b

Počet jedniček	Index	d	c	b	a
1	1*	0	0	0	1
	4*	0	1	0	0
	8*	1	0	0	0
2	9*	1	0	0	1
	10*	1	0	1	0
	12*	1	1	0	0
3	7*	0	1	1	1
	11*	1	0	1	1
	14*	1	1	1	0
	15*	1	1	1	1

c) *Sestavíme tabulku, kde se pokusíme spojit dvojice řádků, které se liší pouze v jednom bitu, tento bit nahradíme znakem "-„. To které dvojice jsme při porovnání použili, si v předchozí tabulce vyznačíme *.*

Tab. 15 Quine-McCluskey, krok c

Index	dcba
1-9	-001
4-12	-100
8-9*	100-
8-10*	10-0
8-12*	1-00
9-11*	10-1
10-11*	101-
10-14*	1-10
12-14*	11-0
7-15*	-111
11-15*	1-11
14-15*	111-

d) *Nyní budeme stejný postup aplikovat na novou tabulku, znak – je třeba chápat jako novou hodnotu. Použité řádky si opět označíme *. (Při porovnávání stačí nalézt řádky se znakem – na stejné pozici). V případě duplicity některé řádky vyškrtneme.*

Tab. 16 Quine-McCluskey, krok d

Index	dcba
8-9 10-11	10--
8-10 9-11	10--
8-10 12-14	1--0
8-12 10-14	1--0
10-11 14-15	1-1-
10-14 11-15	1-1-

e) *Do další tabulky zapíšeme řádky, které jsme nepoužili při žádném porovnání (řádky, které nejsou označeny *) a řádky, které nám zbyly v předchozí tabulce. A vytvoříme tzv. tabulku pokrytí.*

Tab. 17 Quine-McCluskey, krok e

Index	1	4	7	8	9	10	11	12	14	15
1-9	X				X					
4-12		X						X		
7-15			X							X
8-9 10-11				X	X	X	X			
8-10 12-14				X		X		X	X	
10-11 14-15						X	X		X	X

f) *V tabulce pokrytí najdeme ty sloupce, ve kterých je označené jenom jedno políčko (ty to sloupce se označují jako tzv. nesporné implikanty) a tyto sloupce škrtneme. Od těchto sloupců škrtneme celý řádek, a pokud při tom škrtneme další označená políčka, tak škrtneme i další sloupce (ty, které daná políčka obsahují).*

Tab. 18 Quine-McCluskey, krok f

Index	Výraz	1	4	7	8	9	10	11	12	14	15
1-9	abc	X				X					
4-12	$\bar{a}\bar{b}c$		X						X		
7-15	$a\bar{b}\bar{c}$			X							X
8-9 10-11	$\bar{c}d$				X	X	X	X			
8-10 12-14	$\bar{a}d$				X		X		X	X	
10-11 14-15	bd						X	X		X	X

g) *Získáme tabulku, kde ve sloupcích jsou všechny jedničkové stavy (tedy stavy, které musíme pokrýt) a v řádcích jsou všechny přímé implikanty (tedy všechny skupiny sousedních stavů)*

Tab. 19 Quine-McCluskey, krok g

Index	Výraz	8	10	11	14
8-9 10-11	$\bar{c}d$	X	X	X	
8-10 12-14	$\bar{a}d$	X	X		X
10-11 14-15	bd		X	X	X

h) *Vyškrtnutí sloupce, který má hvězdičku ve stejných řádcích jako jiný sloupec nebo má nějaké hvězdičky navíc (dominance sloupce), v příkladu 10. sloupec. Nebo vyškrtnutí řádku, který je podmnožinou jiného řádku (dominance řádku), v tomto příkladu se nevyskytuje.*

Tab. 20 Quine-McCluskey, krok h

Index	Výraz	8	11	14
8-9 10-11	$\bar{c}d$	X	X	
8-10 12-14	$\bar{a}d$	X		X
10-11 14-15	bd		X	X

i) *Sestavení výsledného logického výrazu. Použijeme tří přímých implikantů získaných v bodě f, zbylé termy vybereme z předešlé tabulky dle kritérií minimality (minimální počet termů, minimální počet nezávislých proměnných v každém termu, minimální počet negovaných proměnných v každém termu).*

Výsledek:

$$f(d,c,b,a) = a\bar{b}\bar{c} + \bar{a}\bar{b}c + abc + bd + \bar{a}d$$

nebo:

$$f(d,c,b,a) = a\bar{b}\bar{c} + \bar{a}\bar{b}c + abc + bd + \bar{c}d$$

Dle kritérií minimality byl vybrán term bd, který neobsahuje žádnou negaci. Zbylé dva termy mají stejnou délku a obsahují stejný počet negací, můžeme tedy vybrat kterýkoli z nich (proto máme dva výsledky).

3.2.6 PŘÍKLADY K PROCVIČOVÁNÍ

Př. 11

Pomocí algebraických úprav nalezněte MNDF funkce $\boldsymbol{f = a\bar{b}c + a(b + \bar{c}) + (a + b)(\bar{a} + c)}$

Př. 12

Upravte logický výraz vyjadřující funkci f tak, aby tato funkce byla popsána pomocí minimálního počtu hradel NAND: $\boldsymbol{f = a(\bar{b} + \bar{c}) + abc + abcd}$

Př. 13

Využitím zákonů Booleovy algebry minimalizujte logický výraz $\boldsymbol{(a + b + \bar{c})(a + \bar{a}b + \bar{a}d)}$

Př. 14

Pomocí algebraických úprav nalezněte MNDF funkce $\boldsymbol{f = a(\bar{b} + \bar{c}) + a\bar{b}c + abcd}$

Př. 15

Pomocí algebraických úprav nalezněte MNDF funkce $\boldsymbol{f = \bar{a}d + \bar{b}cd + a\bar{b}(c + d) + \bar{b}\bar{c}\bar{d}}$

Př. 16

Navrhněte schéma zapojení, které bude realizovat dekodér z BCD kódu do kódu sedmisegmentového displeje. Úlohu řešte pouze pro horní vodorovný segment displeje. Proveďte minimalizaci zapojení. Realizujte pomocí hradel NAND.

Př. 17

Navrhněte schéma zapojení, které bude pro 4 vstupní proměnné realizovat prahovou funkci s prahem ≥ 11. K realizaci použijte hradel NAND, minimalizaci proveďte pomocí úpravy algebraického výrazu.

Př. 18

Popište prahovou funkci čtyř proměnných s prahem 12 (vyjádřeno dekadicky) pomocí UNDF a UNKF. Oba logické výrazy minimalizujte a převeďte do tvaru, vhodného pro realizaci pomocí logických členů NAND a NOR.

Př. 19

Navrhněte obvod hlídače liché parity 3 bitové informace pomocí hradel NAND.

Př. 20

Navrhněte schéma zapojení, které bude doplňovat paritní bit liché parity ke tříbitové informaci. (vstupní proměnné: a, b, c výstupní proměnná: p). K realizaci použijte hradel NAND, minimalizaci proveďte pomocí úpravy algebraického výrazu.

Př. 21

Navrhněte schéma zapojení, které bude doplňovat majoritní bit ke tříbitové informaci. (vstupní proměnné: a, b, c, výstupní proměnná: p). K realizaci použijte hradel NAND, minimalizaci proveďte pomocí úpravy algebraického výrazu.

Př. 22

Na Karnaughově mapě pro tři proměnné a b c vyznačte sousední pole k poli reprezentujícímu term $\bar{a}\bar{b}\bar{c}$

4 TECHNOLOGIE VÝROBY ČÍSLICOVÝCH OBVODŮ

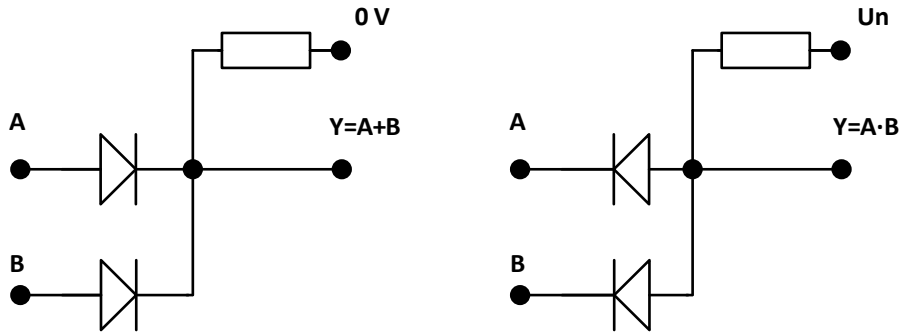
Číslicové elektronické obvody jsou vyrobeny tak, aby jejich prostřednictvím bylo možno realizovat logické funkce. Snahou výrobců je co nejvíce se přiblížit ideální funkci obvodů, pro kombinační obvody to znamená nastavit na výstupu obvodu co nejdříve výsledek logické operace se vstupními proměnnými. Jednotliví výrobci obvodů značí integrované obvody podle jednotného schématu.

4.1 ZKOUMANÉ VLASTNOSTI

Míru přiblížení se k ideálním vlastnostem můžeme zkoumat pomocí různých měřítek. Zpravidla se zajímáme o to, jak daný obvod zatěžuje předchozí obvody, logicky jej napájející (vstupní charakteristika), dále se zajímáme o průběh zpracování vstupního signálů (převodní charakteristika a zpoždění signálu při průchodu obvodem) a o zatížitelnost obvodu dalšími obvody (zatěžovací charakteristika). Důležitými vlastnostmi obvodu je také jeho spotřeba, a to jak v klidovém stavu tak při přepínání mezi stavy. Další vlastností je odolnost proti rušení a u složitějších obvodů existence hazardů. Jelikož výrobci zpravidla neudávají přesně všechny charakteristiky, používají se další míry, charakterizující obvody:

- Vstupní větvení udává hypotetický počet standardních vstupů, které by zatěžovaly předchozí výstupy obvodů stejně jako uvažované hradlo.
- Výstupní větvení udává počet standardních vstupů hradel dané technologie, které je možno připojit k výstupu daného hradla.
- Logický zdvih (viz Obr. 2) udává rozdíl napětí pro log. 1 a log. 0. Hodnota logického zdvihu ovlivňuje odolnost proti rušení.
- Zpoždění signálu při průchodu jedním hradlem je nejsledovanějším parametrem číslicových obvodů, neboť tento parametr přímo ovlivňuje rychlost výpočtů prováděných pomocí číslicových obvodů.

4.2 DIODOVÁ LOGIKA



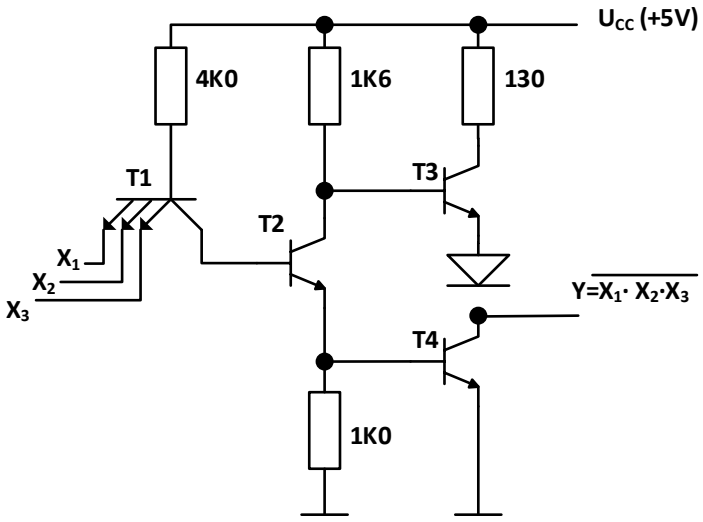
Obr. 10 Příklad diodové logiky, vytvoření logického součtu a logického součinu

Pro pochopení principu číslicových obvodů si nejprve uvedeme příklad diodové logiky, která se v moderních obvodech již nepoužívá, ale je velmi názorná. Zkoumejme nejprve vytvoření logického součinu pomocí diodové logiky. Na rozdíl od příkladu uvedeného na Obr. 10 budeme vytvářet logický součin signálů s logickými úrovněmi log. 1 a log. 0 reprezentovaných napěťovými úrovněmi Un a 0 V. Obvod realizující funkci log. součinu je na Obr. 10.

Předpokládejme nejprve, že napětí na vstupech A a B je log. 1, potom na výstupu Y bude napětí rovno Un, což odpovídá log. 1. V případě, že alespoň na jednom ze vstupů bude log. 0, bude na výstupu Y vzhledem k otevření příslušné diody napětí odpovídající log. 0. Obdobně je možno sestrojit obvod realizující logický součet. (Obr. 10). Diodová logika má podstatná omezení. Na odporech vznikají úbytky napětí, které způsobují degradaci signálu a omezují řazení počtu stupňů diodové logiky na max. dva za sebou. Dále v této logice není možno provést negaci signálu. Tyto problémy řeší diodově-tranzistorová logika, která využívá spojení diodové logiky s invertujícím tranzistorovým zesilovačem. Tato logika je však už zastaralá, její dynamické parametry jsou nevyhovující.

4.3 LOGIKA TTL

Hradla TTL používají více emitorového tranzistoru k vytvoření logické funkce (transistor T1 na Obr. 11), dvojčinného invertujícího stupně s budičem (T2, T3, T4) k úpravě výstupního signálu. Existují různé technologické varianty těchto obvodů: H-rychlá, N-normální, L-nízká spotřeba, S-se Schotkyho diodami (zabraňují saturaci tranzistorů), LS, ALS nízká spotřeba + Schotkyho diody. Z charakteristik hradel TTL uvádíme vstupní charakteristiku (Obr. 12), převodní charakteristiku **Obr. 13**), zatěžovací charakteristiku (**Obr. 14**) a odběrovou charakteristiku (**Obr. 15**). Konkrétní průběh charakteristik je závislý na teplotě. Na **Obr. 16** je znázorněna základní představa o statické odolnosti obvodu TTL proti rušení na vstupu. Odolnost proti rušení je zajištěna pro rušivé signály maximálně do amplitudy 0,4 V. Vyšší amplituda rušivého signálu může způsobit zákmit logické hodnoty na výstupu.



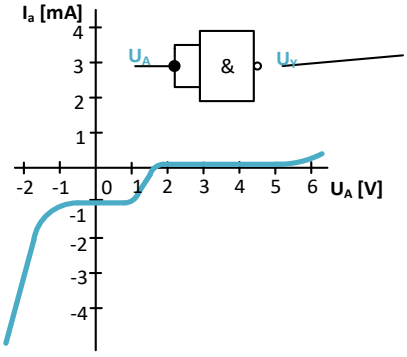
Obr. 11 Třívstupové hradlo NAND v technologii TTL

Na Obr. 11 je vyobrazeno třívstupové hradlo NAND v technologii TTL. Je-li některý ze vstupů v log 0, je T1 otevřen, T2 a T4 uzavřen a T3 otevřen. Jsou-li všechny vstupy v log 1, T1, T3 jsou uzavřeny, T2 a T4 otevřeny.

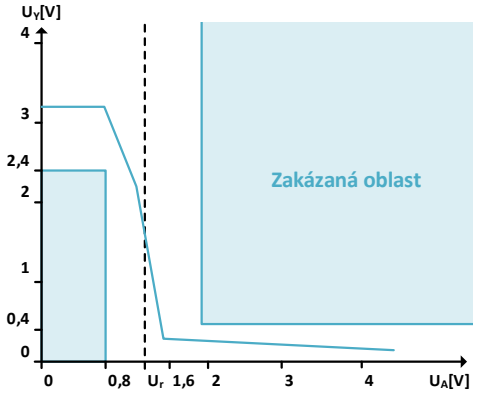
Významným bodem vstupní charakteristiky (**Obr. 12**) je hodnota 1,5 V vstupního napětí pro nulový vstupní proud. Tato hodnota se ustálí na nezapojených vstupech hradla.

Podstatné hodnoty napětí UA a UY na grafu převodní charakteristiky (Obr. 13) jsou:

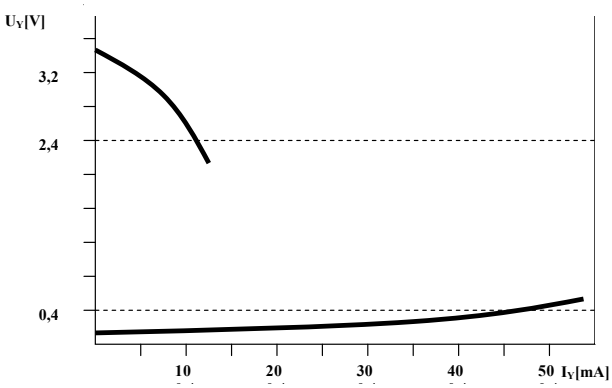
- 0,8 V – max. hodnota UA odpovídající log. 0,
- 2,0 V – min. hodnota UA odpovídající log. 1,
- 2,4 V – max. hodnota UY odpovídající log. 0
- 0,4 V – min. hodnota UY odpovídající log. 1
- Ur – překlápěcí úroveň vstupního napětí



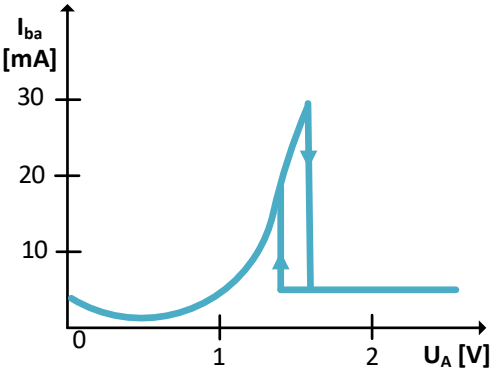
Obr. 12 Vstupní charakteristika hradla TTL



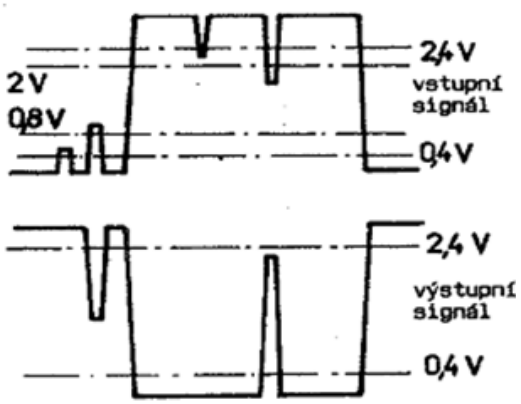
Obr. 13 Převodní charakteristika invertoru TTL



Obr. 14 Zatěžovací charakteristiky hradla TTL . výstupu nastavený do log. 1 a do log. 0.



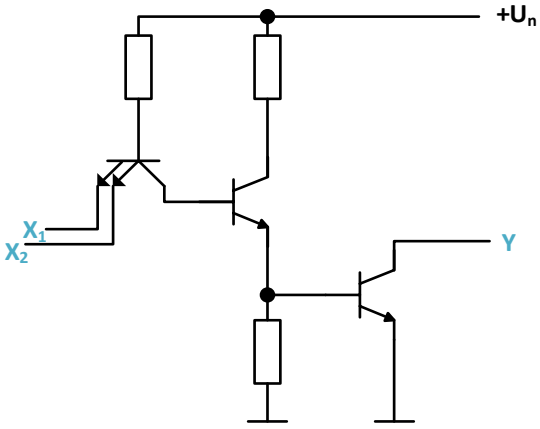
Obr. 15 Odběrová charakteristika invertoru TTL. UA je vstupní napětí na invertoru, ICC proud odebíraný z napájecího zdroje



Obr. 16 Odolnost proti statickému rušení invertoru TTL

4.3.1 Obvody s otevřeným kolektorem

Zapojení TTL hradel s otevřeným kolektorem může být stejné jako u běžného hradla (Obr. 11) s tím rozdílem, že ve schématu je vynechán tranzistor T3 a odpor 130 Ohmů. Zapojení obvodu s otevřeným kolektorem je vyobrazeno na Obr. 17. Chybějící tranzistor pak musí být nahrazen externím odporem nebo dalším hradlem. Výhoda obvodů s otevřeným kolektorem spočívá v tom, že umožňuje spojení výstupů několika hradel. Takto spojené výstupy realizují funkci logického součinu, kterému se běžně říká „montážní součin“. Na rozdíl od obvodů s otevřeným kolektorem je spojení výstupů běžných hradel nepřípustné, neboť může dojít k jejich zničení.

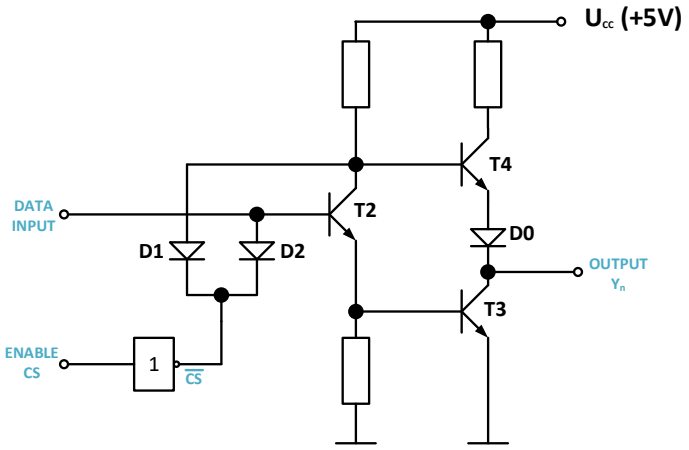


Obr. 17 TTL obvod s otevřeným kolektorem

4.3.2 Hradla s třetím stavem

Třístavová hradla pracují ve dvou základních režimech: v normálním režimu hradlo provádí požadovanou logickou funkci stejně jako hradlo standardní. V režimu vysoké impedance (třetí stav) hradlo vykazuje „nekonečnou“ výstupní impedanci. K řízení těchto dvou režimů se používá pomocného logického vstupu ovládajícího přídavné obvody, které umožňují současně

uzavřít transistory T3 a T4 (stav vysoké impedance výstupu). Obvody se využívají při návrhu sběrnic, protože umožňují obousměrný tok signálu po signálových vodičích, nebo odpojení zařízení, která nejsou adresovaná.



Obr. 18 Tří stavové hradlo

4.3.3 OŠETŘENÍ NEPOUŽITÝCH VSTUPŮ

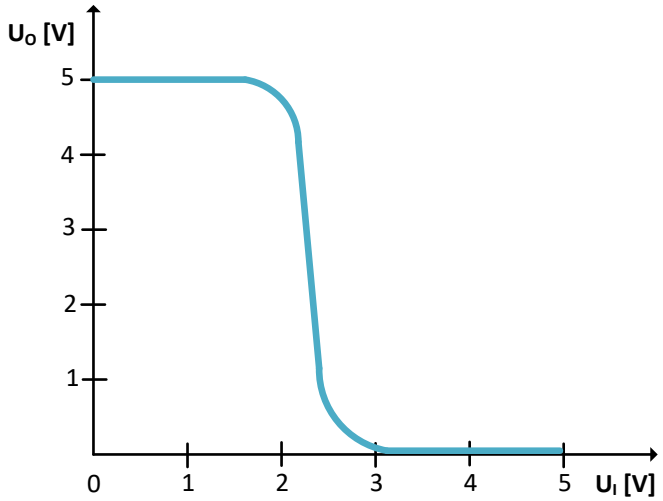
Ze vstupní charakteristiky vyplývá, že není vhodné ponechávat u logiky TTL nezapojené vstupy (nezapojený vstup je zpravidla interpretován jako log. 1, napětí, které se ustálí na tomto vstupu, leží v zakázaném pásmu a existuje nebezpečí špatné interpretace tohoto napětí obvodem). U členů NAND je možno nepoužité vstupy paralelně připojit ke vstupům použitým nebo je připojit na úroveň H. U členů NOR je třeba nepoužité vstupy připojit na úroveň L.

4.4 CMOS TECHNOLOGIE

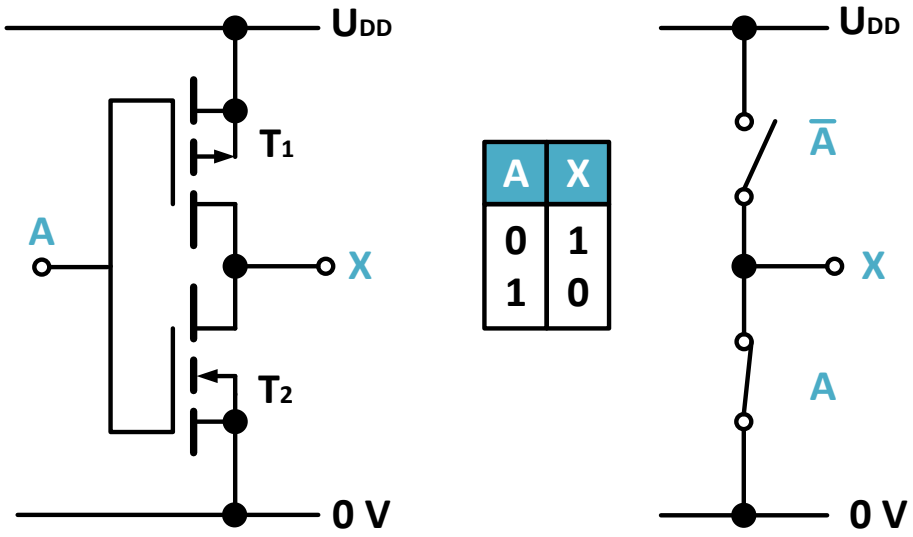
V současnosti je CMOS technologie nejběžnější technologií při realizaci číslicových obvodů. CMOS obvody mají téměř nulovou klidovou spotřebu, odběrová charakteristika však vykazuje relativně velké odběrové maximum při překlápění mezi 0 a 1. Velkou výhodou této technologie je to, že umožňuje vysokou hustotu integrace. Nevýhodou je to, že maximální počet vstupů do jednoho hradla je zpravidla roven 2. Větší počet vstupů do jednoho hradla je nevýhodný z hlediska dynamických vlastností takovéhoto zapojení a bývá nahrazován kaskádou dvouvstupových hradel. Dnešní CMOS obvody jsou již dostatečně rychlé, pro většinu aplikací jsou plnohodnotnou náhradou za obvody TTL. Integrované obvody CMOS jsou označovány jako „unipolární“. Základní jednotku tvoří komplementární tranzistory MOS. Hodnotu napájecího napětí je možno volit v rozsahu od cca 1,5 V do 15 V. Volba napájecího napětí ovlivňuje zejména tyto parametry: rychlost obvodu, šumovou imunitu a spotřebu z napájecího zdroje. U obvodů CMOS je třeba vždy připojit vstupy obvodu na výstup jiného obvodu, na napájecí napětí nebo na zem. Obvody je třeba budit signály, které mají dostatečně strmé náběžné a sestupné hrany, neboť při pomalejších změnách prudce vzrůstá spotřeba obvodu. Klidová spotřeba je velmi nízká (v každé cestě mezi napájecími vstupy hradla je alespoň jeden z tranzistorů uzavřen). Spotřeba obvodu je tedy úměrná frekvenci změn vstupních signálů. Jestliže obvody nejsou chráněny substrátovými diodami proti přepětí na vstupu, je třeba zabránit vzniku a uplatnění statické elektřiny, která může zničit obvod.

Maximální výstupní větvení je větší než 100, tzn. na jeden výstup hradla je možno připojit více než 100 vstupů dalších hradel. Tato možnost je dána tím, že vstupní proud do tranzistoru FET je zanedbatelný. Velká vstupní kapacita obvodů ovšem zhoršuje dynamické vlastnosti.

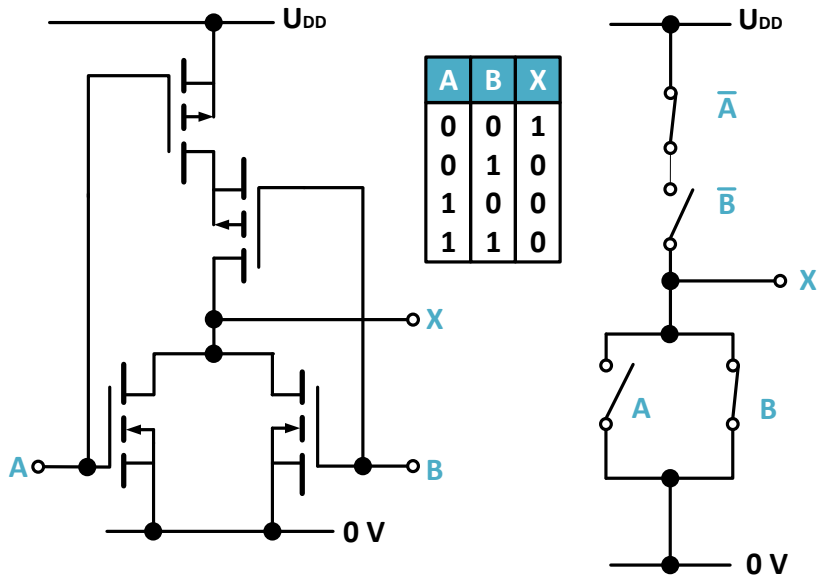
Na dalších obrázcích jsou vyobrazena hradla vyrobená technologií CMOS. Na Obr. 20 představuje invertor, kde při přivedení napětí UDD na oba transistory se T1 otevře, T2 zůstane uzavřen. Při přivedení napětí 0 V se otevře T2 a uzavře T1. Na **Obr. 21** můžeme vidět hradlo NOR a na **Obr. 22** hradlo NAND.



Obr. 19 Převodní charakteristika invertoru CMOS

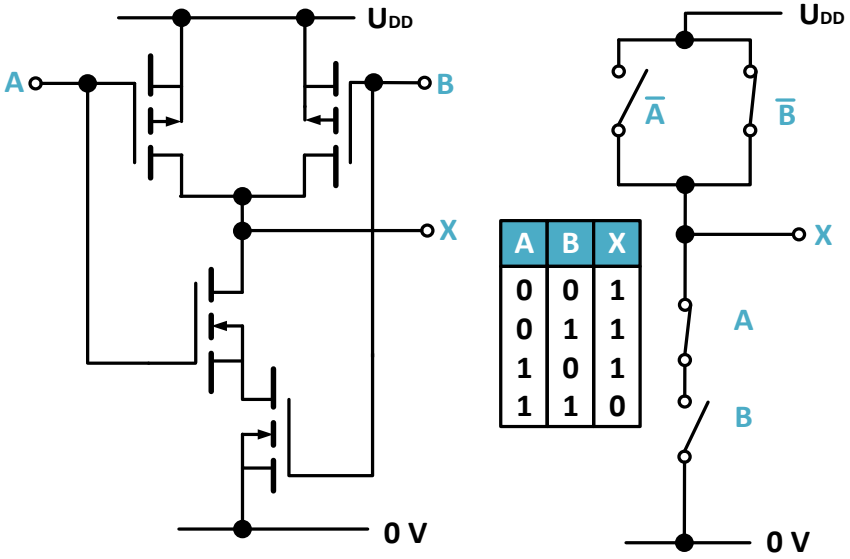


Obr. 20 Invertor v CMOS technologii



A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Obr. 21 Hradlo NOR v CMOS technologii



A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

Obr. 22 Hradlo NAND v CMOS technologii

4.5 DALŠÍ TECHNOLOGIE VÝROBY INTEGROVANÝCH OBVODŮ

V současné době prochází technologie výroby IO rychlým vývojem, o čemž svědčí prudké zvyšování výkonu výpočetní techniky. Nejrychlejší obvody jsou realizovány v současnosti technologií ECL (zpoždění signálu při průchodu hradlem <1 nanosekunda), probíhají pokusy s vytvářením rychlých obvodů v supravodivých materiálech (zpoždění v řádu pikosekund).

4.6 SLUČITELNOST JEDNOTLIVÝCH TECHNOLOGIÍ

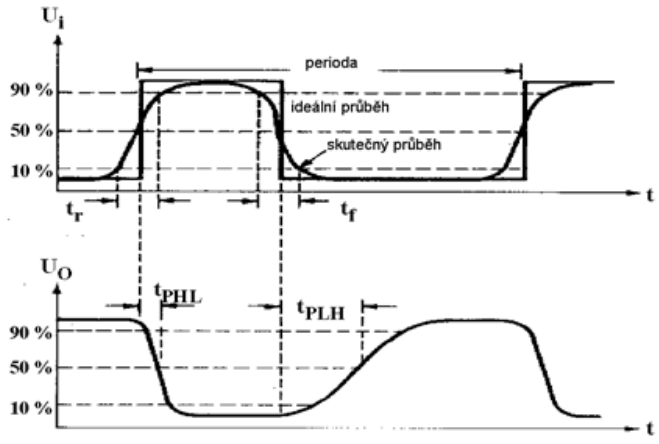
Obecně můžeme říci, že není možno kombinovat v jednom zařízení různé technologie bez přizpůsobení signálů. Např. při porovnání převodní charakteristiky TTL a CMOS vidíme, že rozhodovací úrovně obou těchto technologií jsou vůči sobě navzájem posunuty tak, že obvody CMOS interpretují log. 1 TTL jako log. 0. Proto používáme obvody různých technologií členy, které upravují příslušné úrovně napětí a impedance.

Tab. 21 Srovnání základních parametrů různých technologií výroby číslicových obvodů

Technologie	TTL			CMOS		ECL	
Parametry	74 LS	74 AS	74 ALS	74 C	74 HC	10K	100K
Napájecí napětí,V	5	5	5	5	5	-5,2	-4,5
Maximální úroveň log. 0 na výstupu, V	0,5	0,5	0,5	0,4	0,4	-1,7	-1,7
Minimální úroveň log. 1 na výstupu, V	2,7	2,7	2,7	4,2	4,2	-0,9	-0,9
Minimální úroveň log. 1 na vstupu, V	2,0	2,0	2,0	3,5	3,5	-1,2	-1,2
Maximální úroveň log. 0 na vstupu, V	0,8	0,8	0,8	1,0	1,0	-1,4	-1,4
Logický zdvih, V	2,0	2,0	2,0	3,8	3,8	0,8	0,8
Ztrátový výkon na hradlo, mW	2	20	1	0	0	24	40
Zpoždění signálu, ns	10	1,5	4	30	10	2	0,75
Výstupní větvení	10	10	10	>100	>100	10	10

Parametry pro CMOS technologie byly získány pro zatěžovací proudy 4 mA (log. 0) a 2 mA (log. 1). V Tab. 21 vidíme, že k jednotlivým výstupům hradel můžeme připojit pouze omezený počet dalších vstupů. Jelikož toto omezení není možno vždy dodržet, vytvářejí se tzv. budiče. Tyto obvody nevykonávají žádnou logickou funkci, slouží pouze jako neinvertující zesilovače signálu, tzn., zvyšují výstupní větvení.

Návrhář musí zvolit takovou technologii používaných obvodů, která zaručí, že prahové úrovně napětí a zaručené logické úrovně jsou dostatečně od sebe vzdáleny tak, aby žádný z uvedených šumů nezpůsobil nežádoucí změnu stavu zařízení.



Obr. 23 Zpoždění signálu při průchodu hradlem

4.7 Rušení v číslicových systémech

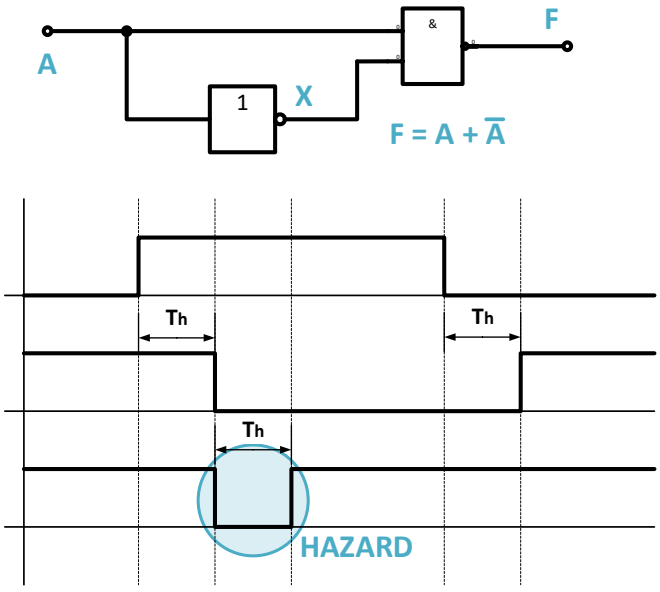
Velkou výhodou číslicových systémů ve srovnání s analogovou technikou je velká tolerance vůči šumu. Odolnost jednotlivých komponent systému je základním faktorem, který ovlivňuje spolehlivost systému. Šum může pocházet v číslicovém systému z více zdrojů: Termální (Johnsonův) šum je způsoben rezistory, výstřelový (shot) šum je způsoben náhodným přenesením náboje mezi tranzistory, 1/f šum je zpravidla způsoben náhodnými variacemi rozšíření nosičů náboje mezi jednotlivými částmi zařízení, interferenční šum souvisí s přenosem náboje od blízkých zařízení kapacitní, nebo indukční vazbou. Při konstrukci obvodu je třeba dbát na to, aby celkový šum byl tak nízký, že neovlivní správnou funkci obvodu.

4.8 Zpoždění a hazardy

Obr. 24 ukazuje typickou závislost průběhu výstupního signálu na vstupním při periodické změně vstupního signálu. Na obrázku si můžeme všimnout, že skutečný průběh signálu není obdélníkový, při průchodu signálu hradlem dochází ještě k jeho další deformaci a že reakce obvodu na změnu vstupu je zpožděná. Průměrná hodnota zpoždění se udává jako veličina tp, kde tp =1/2(tpLH+tpHL). Různě velká zpoždění signálu, který se šíří více rekonvergentními cestami v obvodu může způsobit hazardy, tj. krátké zákmity výstupní logické hodnoty do stavu opačného, než je předpokládaná výstupní logická hodnota. Příklad jednoduchého obvodu s hazardem je na Obr. 24.

Na Obr. 24 vidíme časovou analýzu hazardu v chování obvodu realizujícím funkci F.

Ideální funkce obvodu je konstantní (obvod by bylo možno nahradit signálem log. 1). Reálný časový průběh signálu F má krátký zákmit do log. 0 způsobený nestejným zpožděním signálů v jednotlivých větvích obvodu. Tento zákmit nazýváme hazardem.



Obr. 24 Časová analýza hazardu

5 REALIZACE LOGICKÝCH FUNKCÍ KOMBINAČNÍMI OBVODY

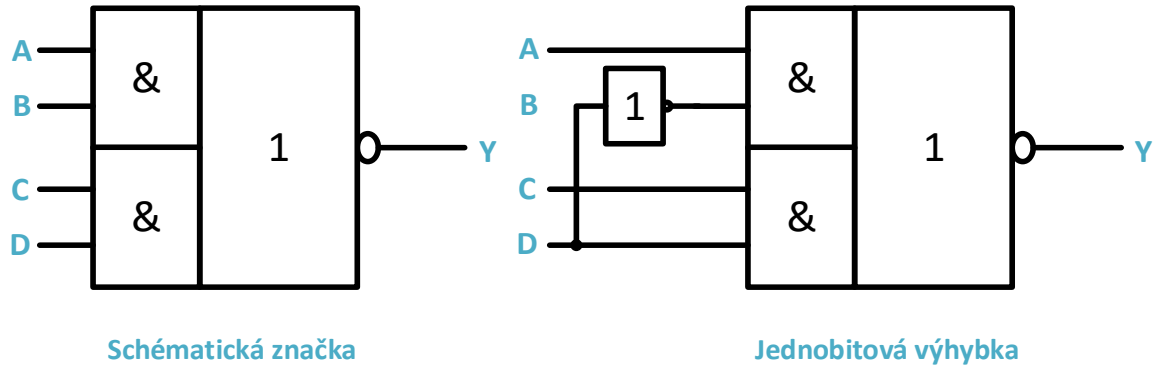
Kombinační funkce můžeme realizovat pomocí základních hradel (NAND, NOR, XOR, NOT,…). V jednotlivých technologiích byly vytvořeny řady základních obvodů, tak aby bylo s jejich pomocí možno snadno realizovat logické obvody. Druhou možností je využít k vytvoření vlastní logické funkce standardních obvodů s vyššími funkčními celky. Tento způsob je výhodný zvláště u složitějších funkcí, kde bychom obtížně sestavovali každý obvod „od začátku" ze základních hradel. Třetí možností je využít programovatelných logických obvodů, o této možnosti bude pojednáno v samostatné kapitole.

5.1 REALIZACE OBVODU POMOCÍ ZÁKLADNÍCH HRADEL

Vzhledem k tomu, že pomocí obvodů NAND můžeme realizovat libovolnou logickou funkci, je jako základní hradlo použit zpravidla 2-8 vstupový obvod NAND pro technologii TTL, 2 vstupový NAND pro technologii CMOS. V tomto případě se více vstupová hradla nahrazují kaskádou dvouvstupových hradel. Existuje více požadavků, které jsou kladeny na výsledný tvar výrazu. Při návrhu obvodů je třeba také uvažovat omezení dané maximálním počtem hradel, kterými se může signál v obvodu šířit (existuje věta, která říká, že každý logický výraz je možno upravit tak, aby počet logických úrovní byl menší nebo roven třem), někdy je třeba zabránit možnosti vytváření hazardů. U složitějších obvodů je třeba dbát na to, aby navržený obvod byl snadno testovatelný.

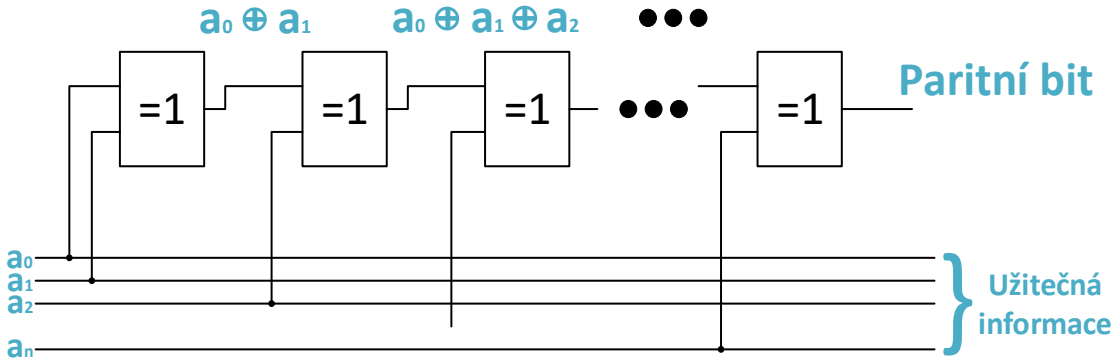
5.2 REALIZACE SLOŽITĚJŠÍCH OBVODŮ POMOCÍ STANDARDNÍCH FUNKČNÍCH CELKŮ

V návrhu kombinačních obvodů se často opakují některá zapojení. Ukázalo se, že je vhodné tato zapojení předpřipravit ve formě samostatného integrovaného obvodu a ušetřit tak návrháři práci s jejich návrhem. Jedním z těchto zapojení je AND-OR-INVERT (Obr. 25). Používá se s výhodou jako základní konstrukční zapojení v zákaznických obvodech v technologii CMOS. Funkce obvodu je zřejmá ze schematické značky: vytvoří se log. součin ze vstupů A a B a současně ze vstupů C a D, z těchto součinů se vytvoří invertovaný součet. Tento obvod je možno využít pro konstrukci logické výhybky. Jednobitová výhybka umožňuje výběr mezi signály A a B pomocí řídícího vstupu V, na výstupu Y se objeví ta informace, která je přivedena na vybraný vstup. Vícebitová výhybka může být vytvořena z jednobitových výhybek tak, že řídící vstup V je společný pro všechny jednobitové výhybky.



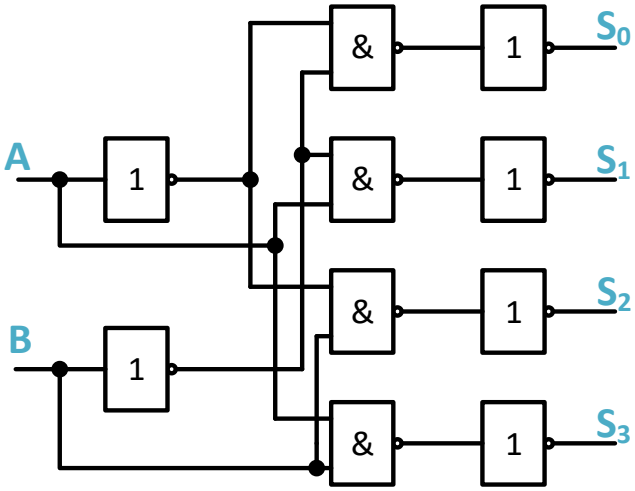
Obr. 25 Obvod AND-OR-INVERT. schématická značka a příklad použití pro konstrukci jednobitové výhybky

Užitečným obvodem je obvod generování, resp. kontroly parity (**Obr. 26**). Parita může být sudá nebo lichá. Jestliže počet jedniček ve slově, které obsahuje informační a paritní bity je lichý, jedná se o paritu lichou, v opačném případě o paritu sudou. **Obr. 26** můžeme chápat jednak jako generátor sudé parity, jednak jako obvod kontroly parity, kdy na výstupním bitu bude hodnota 1 v případě, že parita užitečné informace je lichá.



Obr. 26 Generování sudé parity z n bitové informace

Důležitými obvody jsou kodéry a dekodéry. Pod pojmem kodér rozumíme obvod, který převádí informaci z méně komprimovaných kódů (např. kód 1 z N) do kódů komprimovanějších např. do kódu binárního. Obvody provádějící opačnou funkci říkáme dekodéry. Nejčastěji převádíme informaci z binárního kódu nebo z kódu BCD do kódu 1 z N. Příklad dekodéru z binárního dvoubitového kódu do kódu 1 ze 4 je na Obr. 27. V praxi se ovšem používají dekodéry složitější. Například dekodér 7442, což je dekodér z kódu BCD do kódu 1 z 10 a dekodér 74154 což je dekodér z binárního kódu do kódu 1 ze 16 (**Obr. 28**).

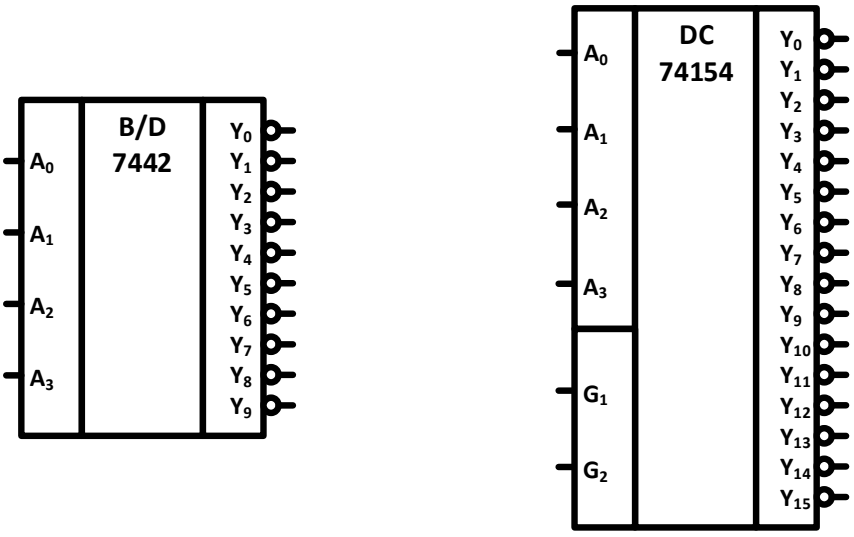


Obr. 27 Principiální schéma dekodéru

Na **Obr. 28** můžeme vidět schematické značky dekodéru z kódu BCD do kódu 1 z 10 (obvod číslo 7442) a dekodéru z binárního 4bitového kódu do kódu 1 ze 16 (obvod číslo 74154). Vstupy G1 a G2 u obvodu 74154 umožňují rozšíření funkce dekodéru na více vstupních proměnných propojením více shodných obvodů. Je-li G1 a současně G2 rovno 0, dekodér nastavuje výstupy podle odpovídající pravdivostní tabulky.

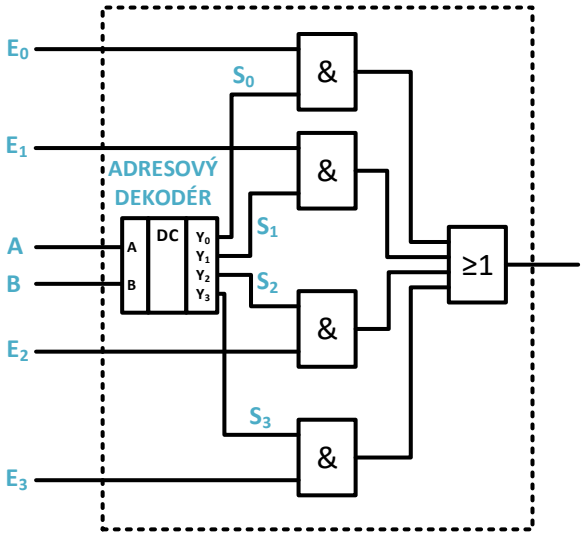
Multiplexor je ve své podstatě výhybkou, která přepíná více než dva vstupy. Principiálně si můžeme jeho činnost popsat pomocí **Obr. 29**. Skládá se z adresového dekodéru, který převede binární informaci o adrese do kódu 1 ze 4 a z hradlovacích obvodů, které uvolní cestu na výstup pouze tomu vstupnímu signálu, jehož adresa je vybrána.

Vnitřní schéma zapojení multiplexoru pomocí hradel (bez použití adresového dekodéru) je vyobrazeno na **Obr. 30**. V praxi ovšem používáme multiplexory s větším počtem adres (vstupů).

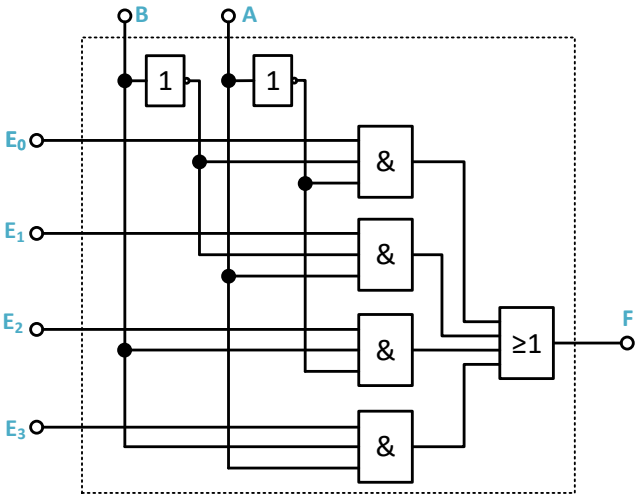


Obr. 28 Schématické značky dekodéru 7442 a 74154

Obecnou schématickou značku multiplexoru můžeme vidět na Obr. 31, příklad multiplexoru 74151 je na Obr. 32. Je-li S (Select) nastaven do log. 1, obvod není vybrán a na výstupu Q je nastavena konstantní hodnota, je-li S=0, obvod pracuje podle pravdivostní tabulky multiplexoru. Název multiplexor se používá i pro složitější obvody, které přepínají mezi více vícebitovými vstupy.



Obr. 29 Principiální schéma multiplexoru



Obr. 30 Skutečné schéma multiplexoru

5.3 Realizace logické funkce pomocí dekodéru a multiplexoru

Standardními funkčními bloky je možno realizovat různé logické funkce. Možnosti využití dekodéru a multiplexoru pro realizaci obvodu realizujícího danou funkci budeme demonstrovat na příkladu.

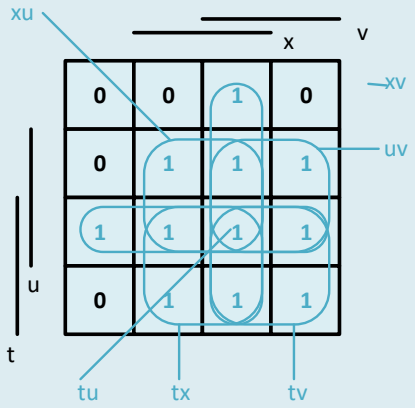
Př. 23 Zapojení alarmu pomocí hradel NAN

Realizujte obvod spuštění alarmu pro zabezpečenou místnost se 4 senzory.
T a U – tlakové senzory, V optický senzor, X - infračervený senzor. Alarm se aktivuje (alarm aktivní pro log. 1), jestliže alespoň dva ze senzorů T, U, V a X jsou aktivovány (na jejich výstupu je log. 1). Nyní si ukážeme řešení této úlohy s využitím hradel NAND, dekodéru a multiplexoru. Zadání úlohy odpovídá funkci majority ze čtyř proměnných, kdy připouštíme, že i polovina jedniček ve vstupním slově tvoří majoritu. Funkci majority můžeme popsat pomocí následující pravdivostní tabulky.

Tab. 22 Pravdivostní tabulka majoritní funkce 4 proměnných

index	t	u	v	x	s
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

Pro realizaci hradly NAND je zapotřebí z pravdivostní tabulky získat minimalizovanou logickou funkci, kterou je možné po úpravě (pomocí De Moraganovo pravidlo) zapojit pomocí hradel NAND. Pro získání minimální logické funkce využijeme Karnaughovy mapy. Karnaughova mapa pro minimalizaci Tab. 22 je vyobrazena na Obr. 31. Vyčtený minimální logický výraz (MNDF) dále upravíme pomocí De Morganova pravidla.



Obr. 31 Karnaughova mapa pro minimalizaci dané úlohy

MNDF:

$$s = vx + ux + uv + tv + tu$$

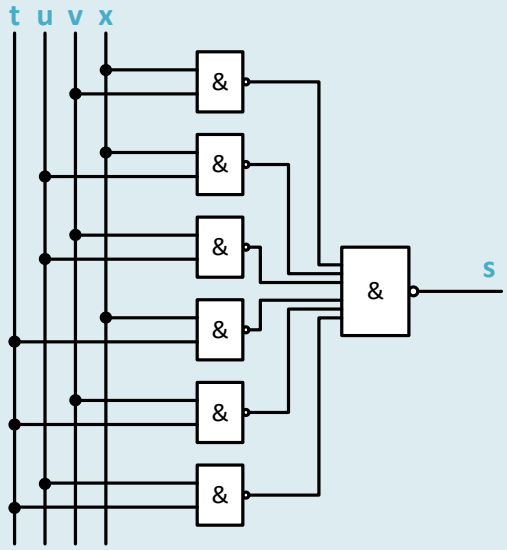
Úprava pomoví De Morganova pravidla:

$$s = \overline{\overline{(vx + ux + uv + tx + tv + tu)}}$$

Výraz upravený pro realizaci hradly NAND:

$$s = \overline{\overline{vx} \cdot \overline{ux} \cdot \overline{uv} \cdot \overline{tx} \cdot \overline{tv} \cdot \overline{tu}}$$

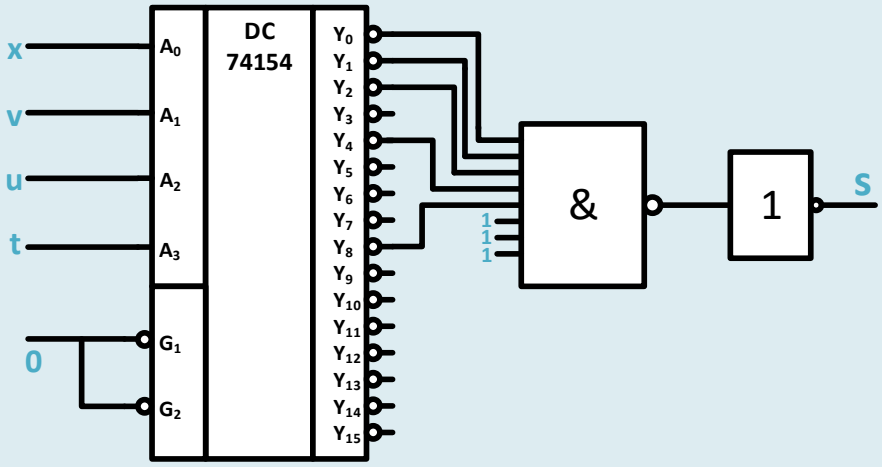
Výsledný logický výraz nyní realizujeme hradly NAND, přičemž každý negovaný součin v tomto výrazu představuje jedno hradlo NAND (tj. 6 dvouvstupových hradel a jedno šestivstupové hradlo). Logický obvod realizující zadanou funkci je vyobrazen na Obr. 32.



Obr. 32 Realizace úlohy pomocí hradel NAN

Př. 24 Realizace alarmu pomocí dekodéru

Další možností jak realizovat danou úlohu, je použití dekodéru. V tomto případě postupujeme tak, že vybereme jednotlivé řádky tabulky s jednotkovou funkční hodnotou a logický součet těchto řádků bude tvořit výslednou funkci. Jelikož čísla řádků jsou volena tak, že odpovídají binárnímu zakódování jednotlivých proměnných, můžeme k sestavení obvodu použít dekodér. Dekodér totiž přiřazuje každému zakódovanému číslu řádku tabulky jeden výstup. V případě, že je na vstupu nastaveno nějaké číslo, bude aktivní příslušný výstupní vodič. Výstupní vodiče odpovídající řádkům s nenulovou funkční hodnotou můžeme logicky sečíst obvodem OR a získáme signál S, který aktivuje alarm.
Jelikož v příkladu použijeme dekodér 74151, který má výstupy negovány, musíme místo součtu výstupních vodičů pomocí obvodu OR provést negovaný součin obvodem NAND (viz De Morganova pravidla). Jelikož v pravdivostní tabulce je více než osm řádků s jedničkovou funkční hodnotou S, není možno realizovat výsledný negovaný součin pomocí jednoho hradla NAND.



Obr. 33 Realizace dané úlohy s využitím dekodéru

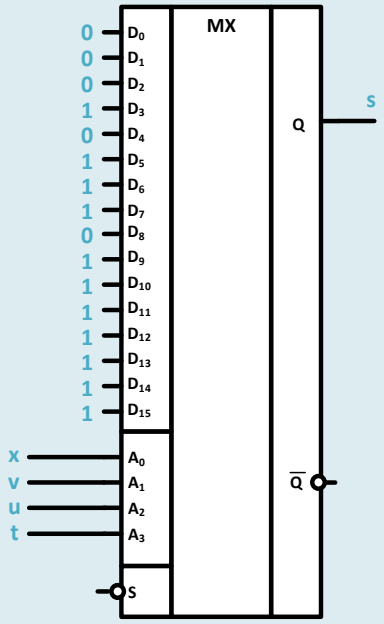
Proto jsme vytvořili obvod pro realizaci funkce S', kde S' je inversí funkce S. Funkci S jsme potom získali tím, že jsme funkční hodnotu S' invertovali pomocí invertoru. Toto zapojení můžeme vidět na následujícím obrázku (Obr. 33).

Př. 25 Realizace alarmu pomocí multiplexoru

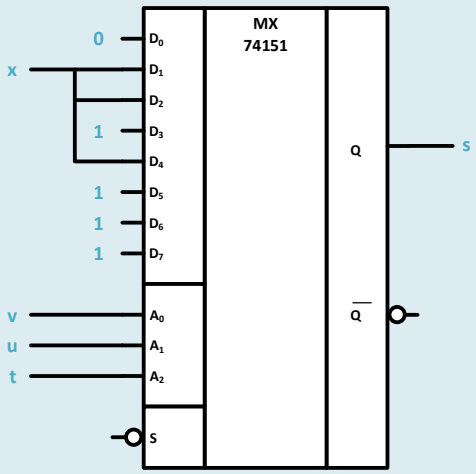
*Třetí možností jak danou úlohu realizovat, je využití multiplexoru. Jelikož máme 4 vstupní proměnné (viz **Tab. 23**) můžeme pro realizaci využít 16 vstupového multiplexoru, kde na adresové vstupy přivedeme naše vstupní proměnné (čidla t, u, v, x) a na datové vstupy multiplexoru přivedeme požadované výstupní hodnoty pro všechny kombinace vstupních proměnných. Realizace zadaného příkladu tímto způsobem je vyobrazena na **Obr. 34**. Tento způsob řešení je však dosti nehospodárný, a proto pro realizaci použijeme 8mi vstupový multiplexor. Toto řešení je vyobrazeno na **Obr. 35**.*
*V tomto případě použijeme jako adresu pouze tři ze vstupních proměnných a 4. vstupní proměnou (v přímém nebo negovaném tvaru) přivádíme spolu s log. 1 a log. 0 na datové vstupy multiplexoru v závislosti na vztahu mezi použitou vstupní proměnnou (v tomto případě proměnná X) a požadovanou hodnotou výstupy pro všechny kombinace zbylých vstupních proměnných (naznačeno v **Tab. 24**, která je součástí **Obr. 35**).*

Tab. 23 Pravdivostní tabulka pro realizaci úlohy s využitím 16 vstupového multiplexoru

index	t	u	v	x	s
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1



Obr. 34 Realizace dané úlohy s využitím 16 vstupového multiplexoru



Obr. 35 Realizace dané úlohy s využitím 8 vstupového multiplexoru

Tab. 24 Pravdivostní tabulka pro realizaci úlohy s využitím 8 vstupového multiplexoru

index	a	b	c	d	s	
0	0	0	0	0	0	0
1	0	0	0	1	0	
2	0	0	1	0	0	X
3	0	0	1	1	1	
4	0	1	0	0	0	X
5	0	1	0	1	1	
6	0	1	1	0	1	1
7	0	1	1	1	1	
8	1	0	0	0	0	X
9	1	0	0	1	1	
10	1	0	1	0	1	1
11	1	0	1	1	1	
12	1	1	0	0	1	1
13	1	1	0	1	1	
14	1	1	1	0	1	1
15	1	1	1	1	1	

5.4 ÚLOHY K PROCVIČENÍ LÁTKY

V následujících úlohách sestavte pravdivostní tabulku slovně popsaných logických funkcí, minimalizujte výraz pomocí Karnaughových map a realizujte zapojení obvodu pomocí hradel NAND. Alternativně vyřešte s pomocí dekodéru a multiplexoru.

Př. 26 Spouštění elektromotoru

Navrhněte pomocí hradel NAND, multiplexoru nebo dekodéru.
Před zapnutím trojfázového elektromotoru /nulové otáčky/, je nutné připojit kartáčky a zařadit spouštěcí odpor. Po spuštění je nutno vyřadit spouštěcí odpor a odpojit kartáčky. Navrhněte logický obvod, který vyšle výstražný signál, jestliže v klidovém stavu nejsou zapojeny kartáčky a zařazen odpor nebo při běhu motoru jsou zapojeny kartáčky nebo zařazen odpor, nebo se po zapnutí motor nerozběhne.

Log. proměnné: Zapnutý motor, nenulové otáčky, zařazený odpor, zapnuté kartáčky.
Log. funkce: Výstražný signál, Ovládání světel automobilu

Př. 27 Světla automobilu

Navrhněte pomocí hradel NAND, multiplexoru nebo dekodéru.
V automobilu je možno zapnout tato světla: parkovací, tlumená, dálková a mlhová. Platí tato pravidla: Při zapnutí tlumeného, dálkového nebo mlhového světla se musí automaticky rozsvítit světlo parkovací. Mlhová světla svítí pouze tehdy, když nesvítí světla dálková. Při současném požadavku na rozsvícení tlumených a dálkových světel se rozsvítí světla dálková. Navrhněte logický obvod, který pro libovolnou kombinaci tlačítek zajistí správné rozsvícení světel i se zřetelem na bezpečnost silničního provozu.

Log. proměnné: tlačítka parkovacích, tlumených, dálkových a mlhových světel.
Log. funkce: napětí na vodičích vedoucích k jednotlivým žárovkám parkovacích, tlumených, dálkových a mlhových světel.

Př. 28 Nápojový automat

Navrhněte pomocí hradel NAND, multiplexoru nebo dekodéru.
Automat obsahuje otvor pro vhození malé či velké mince a dvě tlačítka pro volbu sodovky nebo limonády. Po vhození malé mince a stisknutí tlačítka sodovka automat nalije sodovku, po vhození velké mince a stisknutí tlačítka limonáda nalije limonádu. Při chybné volbě automat vrátí minci. Automat nesmí šidit ani majitele ani zákazníky.

Log. proměnné: vhozena velká mince, vhozena malá mince, zvolena sodovka, zvolena limonáda
Log. funkce: napětí na elektromagnetu pro nalití sodovky, limonády a pro vrácení mince.

Př. 29 Ochrana parního kotle

Navrhněte pomocí hradel NAND, multiplexoru nebo dekodéru.
Parní kotel má čtyři hořáky, na každém z nich je čidlo, které signalizuje, zda plamen hoří či nehoří. Navrhněte logický obvod, který signalizuje poruchu, hoří-li tři nebo méně hořáků a uzavře přívod plynu, zhasnou-li dva nebo více hořáků.
Log. proměnné: Indikace hoření jednotlivých hořáků
Log. funkce: Poruchový signál, elektromagnet ovládající přívod plynu

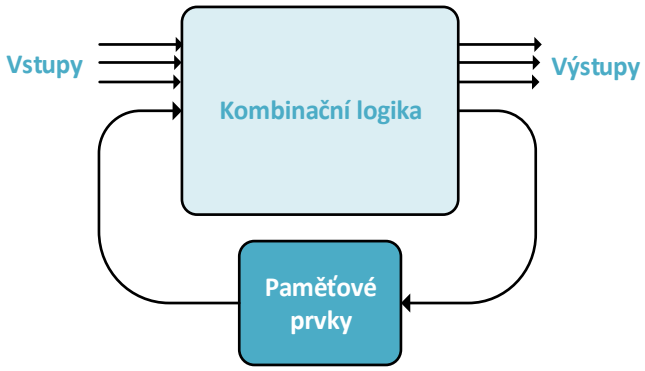
Př. 30 Jednobitová sčítačka

Navrhněte pomocí hradel NAND, multiplexoru nebo dekodéru.
Před zapnutím trojfázového elektromotoru /nulové otáčky/, je nutné připojit kartáčky a zařadit spouštěcí odpor. Po spuštění je nutno vyřadit spouštěcí odpor a odpojit kartáčky. Navrhněte logický obvod, který vyšle výstražný signál, jestliže v klidovém stavu nejsou zapojeny kartáčky a zařazen odpor nebo při běhu motoru jsou zapojeny kartáčky nebo zařazen odpor, nebo se po zapnutí motor nerozběhne.

Log. proměnné: Zapnutý motor, nenulové otáčky, zařazený odpor, zapnuté kartáčky.
Log. funkce: Výstražný signál, Ovládání světel automobilu

6 SEKVENČNÍ OBVODY

Sekvenční obvod je takový obvod, jehož výstupy jsou určeny hodnotou vstupů a vnitřním stavem. Na Obr. 36 vidíme, že při konstrukci sekvenčních obvodů bychom měli dodržet pravidlo, které neumožňuje vkládat zpětné vazby do kombinačního obvodu, bez toho, aniž by tyto vazby vedly přes paměťové členy, tj. obvody, které jsou schopny definovaným způsobem uchovávat informace. Tento přístup k sekvenčním obvodům usnadňuje návrh, neboť zjednodušuje analýzu chování obvodu. V této kapitole se budeme zabývat zvlášť jednotlivými typy paměťových prvků, tj. klopnými obvody a pamětmi, základními sekvenčními automaty tj. čítači a ukážeme možnost jednoduchého návrhu obecných automatů.



Obr. 36 Obecné blokové schéma sekvenčního obvodu

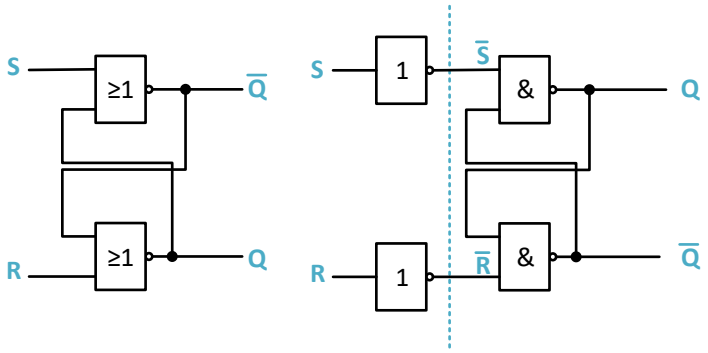
6.1 KLOPNÉ OBVODY

Klopný obvod (KO) je elektronický obvod, který se skokové překlápí mezi dvěma napěťovými stavy. Klopné obvody rozdělujeme na bistabilní, monostabilní a astabilní. Astabilní obvody slouží ke generování periodického signálu, který může být využit například pro synchronizaci. Monostabilní obvody upravují zachycené impulsy na impulsy s předem stanovenou délkou. Bistabilními KO, na které se zaměříme v následující kapitole, jsou KO se dvěma stabilními stavy (tj. pro změnu stavu je zapotřebí vnější podmět) a používají se jako paměťové prvky (anglicky flip-flops). Tyto KO dělíme na asynchronní a synchronní. Synchronní obvody reagují na vstupní signály pouze v okamžicích, kdy je aktivní hodinový signál, povolující změnu stavu. Asynchronní obvody reagují na všechny změny vstupního signálu. Asynchronní obvody jsou jednodušší než synchronní, a proto se jimi budeme zabývat dříve.

6.1.1 RS KLOPNÝ OBVOD

Vstupní proměnné jsou R a S výstupní proměnná je Q a její negace. Hodnota Qt v pravdivostní tabulce (Tab. 25 a Tab. 26) odpovídá hodnotě výstupu Q v okamžiku před změnou jedné ze vstupních proměnných na hodnotu uvedenou v tabulce, Hodnota Qt+1 udává hodnotu výstupu Q po přivedení příslušných hodnot na vstupy R a S při původní hodnotě výstupu Qt.

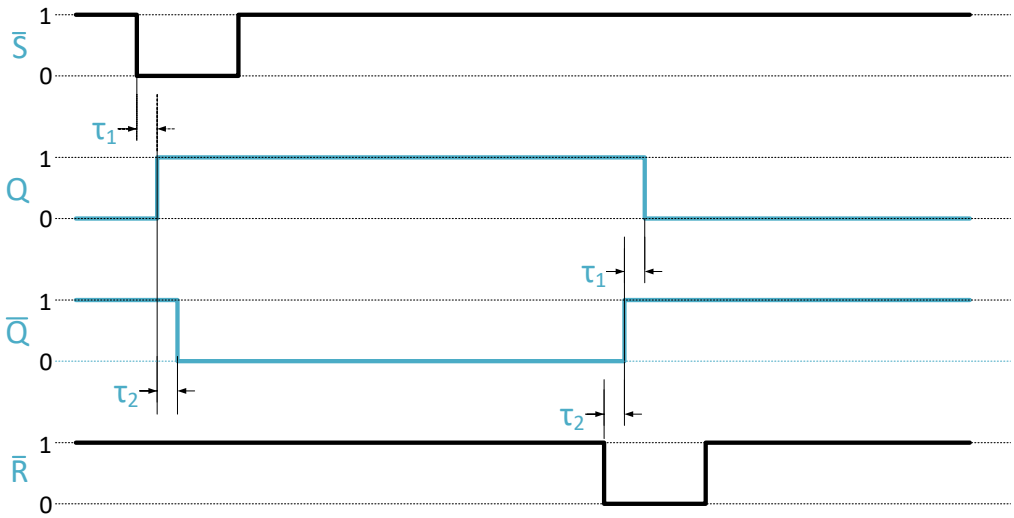
Obvod RS je paměťový prvek sloužící k zapamatování 1 bitu informace. Jestliže je aktivní vstup S (SET) zapisuje se do obvodu log. 1, je-li aktivní vstup R (RESET) zapisuje se do paměti log. 0. Není-li žádný ze vstupů aktivní, nemění se obsah paměti, pamatuje se předchozí stav. U obvodu tvořeného hradly NOR jsou aktivní úrovně signálů R a S logické 1. U obvodu tvořeného výhradně obvody NAND jsou aktivními úrovněmi signálů a úrovně log. 0. Proto se k tomuto obvodu mohou přidat vstupní invertory (viz Obr. 37), které změní aktivní úrovně tak, aby byly v souladu s běžnou konvencí. RS klopné obvody tedy v případě aktivního signálu R nulují výstup Q, v případě aktivního signálu S nastavují výstup a v případě obou vstupů neaktivních si pamatují předchozí stav (Tab. 25 a Tab. 26).



Obr. 37 RS klopný obvod tvořený hradly NOR a NAND

Je nutno zabránit tomu, aby byly současně aktivní oba vstupy, protože na výstupech se objeví zakázaný stav, kdy neplatí předpoklad o tom, že je inverzí signálu Q. Tohoto stavu je třeba se vyvarovat v zapojeních s obvody RS, neboť může vést k nedefinovanému následujícímu stavu.

Časový diagram RS klopného obvodu je na Obr. 38. Na diagramu je patrný vliv zpoždění signálu při průchodu jednotlivými hradly v okamžicích překlápění stavu klopného obvodu.



Obr. 38 Časový diagram RS obvodu tvořeného hradly NAND

Tab. 25 Pravdivostní tabulka RS KO tvořeného hradly NOR. Zakázané stavy jsou označeny x

R	S	Q_t	Q_{t+1}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0x
1	1	1	0x

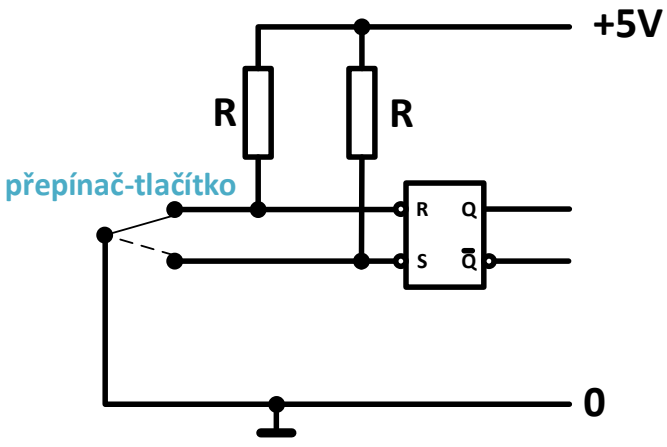
Tab. 26 Pravdivostní tabulka RS KO tvořeného hradly NAND. Zakázané stavy jsou označeny x.

$\overline{R}$	$\overline{S}$	Q_t	Q_{t+1}
0	0	0	1x
0	0	1	1x
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Jako příklad využití RS klopného obvodu si nyní ukážeme úlohu na ošetření tlačítka proti zákmitu.

6.1.2 Ošetřené tlačítko

Mechanický přepínač – tlačítko při sepnutí nebo rozepnutí obvodu způsobí několik zákmitů výstupního napětí. Jelikož elektronické obvody na tyto nežádoucí zákmity reagují, je třeba kritická tlačítka ošetřit. Ošetřené tlačítko (přepínač) lze získat z obvodu RS s negovanými vstupy. Schéma zapojení je na Obr. 39.

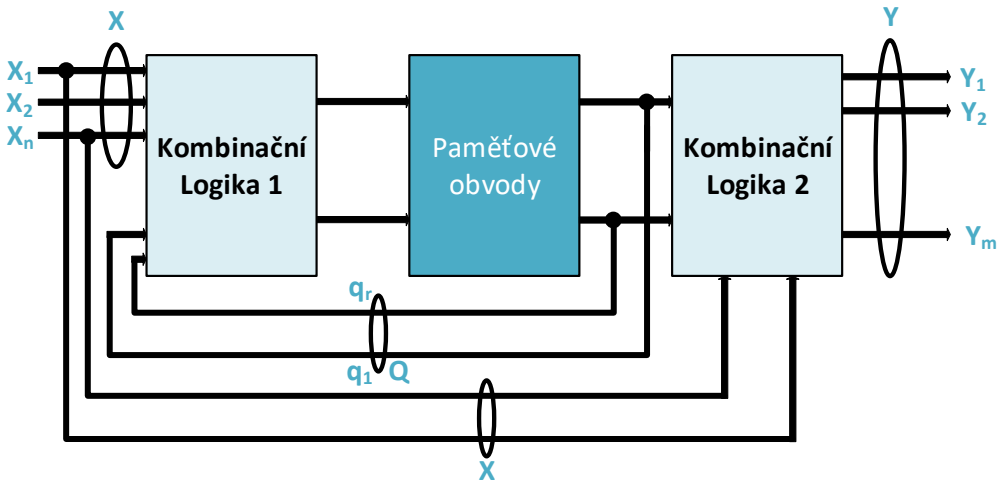


Obr. 39 Ošetřené tlačítko s RS KO

Odpor R volíme v řádu kΩ. V naznačené poloze přepínače je R = 0, S = 1, takže podle pravdivostní tabulky RS obvodu je Q = 0. Při přepínání, je-li kontakt v mezipoloze je R = 1, S = 1, takže se zachová předchozí stav Q = 0, při prvním styku s dolním kontaktem je R = 1, S = 0, obvod se překlopí a Q = 1. Odskočí-li kontakt, dojde k přerušení dotyku, nastaví se R = 1, S= 1 avšak stav Q = 1 zůstává zachován. Na výstupu Q tedy vznikne pouze jeden přechod z úrovně 0 na úroveň 1, tedy i při vícenásobném odskočení kontaktu dojde k vytvoření pouze jedné vzestupné hrany při stisknutí tlačítka. Analogicky lze analyzovat pohyb kontaktu směrem vzhůru, který vzniká při uvolnění tlačítka, kdy dojde k vytvoření pouze jedné sestupné hrany výstupního signálu.

6.2 NÁVRH ASYNCHRONNÍHO SEKVENČNÍHO OBVODU

Asynchronní sekvenční obvod obsahuje alespoň jednu zpětnou vazbu. Stav asynchronního obvodu tedy závisí nejen na hodnotách vstupních proměnných ale i na hodnotě minulého vnitřního stavu. Obecně je návrh asynchronního obvodu poměrně komplikovaným úkolem, neboť při návrhu je třeba respektovat všechny možné hazardy, které mohou vzniknout v kombinační části obvodu a je třeba zajistit stabilitu jednotlivých stavů tak, aby se projevily na výstupu obvodu.



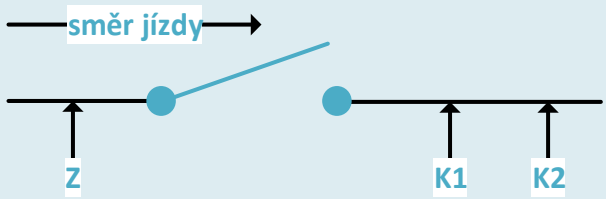
Obr. 40 Typy sekvenčních automatů – obecné schéma

Dále je třeba rozhodnout, jaký typ automatu chceme zkonstruovat, základní dva typy automatu jsou uvedeny na Obr. 40. Vektor X popisuje vstupní proměnné, vektor Y proměnné výstupní, vektor Q popisuje vnitřní stavy. Jestliže vstupy X ovlivňují pouze kombinační logiku kombinační logiku 1 a nikoliv kombinační logiku 2, jedná se o automat typu Moore. V opačném případě hovoříme o automatu typu Mealy.

Obrázek výše popisuje nejen asynchronní sekvenční obvod ale i obvod synchronní. Aby úloha syntézy obvodu byla jednodušší, nepředpokládáme v sekvenčním obvodu existenci žádné zpětné vazby, která v sobě neobsahuje paměťový prvek. Tento předpoklad nám umožní využít k návrhu standardní klopné obvody a usnadní analýzu chování obvodu. Postup návrhu jednoduchého asynchronního sekvenčního obvodu budeme ilustrovat na příkladu.

Př. 31 Železniční přejezd

Na železničním přejezdu jezdí vlaky jedním směrem, zleva doprava podle Obr. 41. Přejede-li lokomotiva bod Z, závory se spustí, je-li poslední vagón za bodem K1, závory se zvednou. Sestavte obvod, který ovládá signál pro spuštění závor.



Obr. 41 Železniční přejezd. Z, K1 a K2 jsou kontakty spínané vlakovou soupravou

Řešení je možné při použití třech spínačů v bodech Z, K1 a K2, které sepnou v případě přítomnosti některé části vlaku nad spínačem a pomocí obvodu, který vyhodnotí, ve kterém místě se nachází vlaková souprava. Akční veličinu, ovládající spouštění závor označíme jako Q. Je-li Q=1, závory se spustí. Jelikož pro správnou funkci závor postačuje jednobitová informace o tom, jestli se vlak nachází na místě, pro které je třeba spustit závory nebo ne, budeme tuto informaci uchovávat v jednom paměťovém prvku – v našem případě v RS klopném obvodu. Podmínky pro spuštění závor vyjádříme pravdivostní tabulkou Tab. 27:

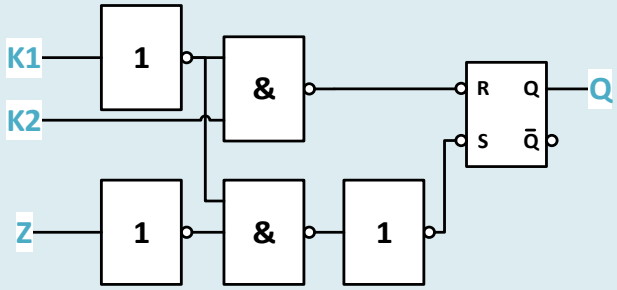
Tab. 27 Podmínky pro spuštění závor

Číslo řádku	Z	K1	K2	Q_t	Q_{t+1}	Funkce RS obvodu
1	0	0	0	0	0	paměť
2	0	0	0	1	1	paměť
3	0	0	1	X	0	nulování
4	0	1	X	X	1	nulování
5	1	X	X	X	1	nulování

*V souladu s předpoklady jsme pro výsledný signál Qt+1 určili, že v případě, když žádný z kontaktů nehlásí přítomnost vlaku, bude zapamatován předchozí stav, v případě, že alespoň jeden z kontaktů Z nebo K hlásí přítomnost vlaku, budou závory spuštěny nezávisle na předchozím stavu, a v případě, že K2 hlásí přítomnost vlaku a současně ostatní kontakty hlásí nepřítomnost vlaku, závory budou zvednuty. Jestliže porovnáme pravdivostní tabulku úlohy s pravdivostní tabulkou RS klopného obvodu (*Tab. 25 **a Tab. 26**) *vidíme, že RS klopný obvod můžeme ovládat tak, aby na jeho výstupu byla logická hodnota odpovídající signálu pro ovládání závor. Budící funkce RS obvodu potom budou realizovány následovně: na vstup S přivedeme hodnotu log. 1 v případě, že Z nebo K1 jsou rovny 1. De Morganovými pravidly upravíme výraz:*

$$S = Z + K1 = \overline{\bar{Z} \cdot \overline{K1}}$$

Při této hodnotě vstupu S budou závory spuštěny. Na vstup R přivedeme hodnotu log. 1 v případě, že platí současně, že K1 = 0 a K2 = 1. Při této hodnotě vstupu R budou závory vytaženy. Případ, kdy současně oba vstupy S i R jsou rovny jedné, nemůže nastat. Jestliže jsou všechny vstupní signály nulové, na S i R vstupy přivádíme 0. Při této hodnotě vstupů setrvávají závory v poloze dané předchozím nastavením. Výsledné schéma zapojení obvodu je na Obr. 42.



Obr. 42 Schéma obvodu pro ovládání závor

6.3 ÚLOHY NÁVRHU SEKVENČNÍCH OBVODŮ

Př. 32 Teplota v peci

Navrhněte sekvenční synchronní automat
Teplota v peci je snímána čidlem připojeným na analogově digitální převodník. Z jeho výstupu jsou odvozeny dvě logické proměnné H a D. Proměnná H nabývá log. 1, stoupne-li teplota nad stanovenou mez TH, proměnná D je v log.1 při poklesu teploty pod povolenou hodnotu TD. Navrhněte obvod, který zapne topení, klesne-li teplota pod TD a vypne jej při překročení teploty TH.
Log. proměnné: Teplota pod TD, teplota nad TH
Log. funkce: Ovládání topení, Automatické startování automobilu

Př. 33 Elektromotor 1

Navrhněte sekvenční synchronní automat
Sestavte logický obvod, který vypne elektromagnet startéru, jestliže motor naskočil a nedovolí startovat při běžícím motoru. Startuje se stiskem tlačítka a startér běží tak dlouho, dokud motor nedosáhne předepsaných otáček, signalizovaných log.1 příslušného čidla. Výstupní logická proměnná ovládá elektromagnet startéru.

Log. proměnné: Tlačítko startéru, indikace běžícího motoru
Log. funkce: Ovládání elektromagnetu startéru, Spouštění elektromotoru

Př. 34 Elektromotor 2

Navrhněte sekvenční synchronní automat
Elektromotor se spouští tlačítkem Z a zastavuje tlačítkem V. Navrhněte logický obvod, který zajistí, že elektromotor běží po stisku tlačítka Z a zastaví se po stisku tlačítka V i v případě, že je současně stisknuto tlačítko Z. Výstupní logická proměnná ovládá stykač, hodnota log.1 = zapnuto.

Log. proměnné: Tlačítko <Z>apni, <V>ypni
Log. funkce: Ovládání stykače motoru, Regulace výšky hladiny

Př. 35 Hladina nádrže

Navrhněte sekvenční synchronní automat
Stav hladiny v nádrži je sledován třemi snímači A, B, C. Snímač A je nejvýše, Snímač C nejníže. Na výstupu snímače je log.1, jestliže není ponořen. Voda je čerpána dvěma čerpadly - hlavním a vedlejším. Klesne-li hladina pod B, zapne se hlavní čerpadlo; klesne-li pod C zapne se i pomocné čerpadlo. Obě čerpadla se vypnou, stoupne-li hladina nad snímač A. Sestavte logický obvod, který zajistí automatické zapínání a vypínání čerpadel.

Log. proměnné: Čidlo úrovně hladiny A, B, C
Log. funkce: Ovládání Hlavního a Pomocného čerpadla, Automatické závory

Př. 36 Zabezpečení železniční tratě

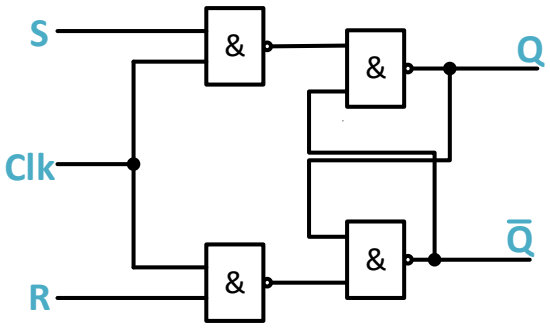
Navrhněte sekvenční synchronní automat
Na trati jezdí vlaky jen jedním směrem. Před závorami je kontakt S, za nimi v dostatečné vzdálenosti kontakt Z. Kontakt je sepnut, je-li nad ním vlak a v tom případě je na jeho výstupu log.1. Navrhněte logický obvod, který spustí závory, najede-li lokomotiva nad kontakt S a zvedne je, jakmile vlak najede na kontakt Z.

Log. proměnné: Nájezdový kontakt Z a S.
Log. Funkce: Ovládání závor

6.4 SYNCHRONNÍ KLOPNÉ OBVODY

Asynchronní klopné obvody reagují okamžitě po změně vstupních proměnných, což může vést k obtížím při návrhu sekvenčních obvodů, neboť obvody připojené na vstupu klopného obvodu mohou vlivem hazardů způsobit nechtěné překlopení obvodu. Při návrhu asynchronních obvodů je tedy třeba důsledně všechny možné hazardy odstranit a zajistit stabilitu jednotlivých stavů. Takovýto způsob návrhu je obtížný, pro složitější obvody je prakticky neřešitelný, a proto se více než asynchronních obvodů v praxi využívá obvodů synchronních, které reagují na vstupní signály pouze v těch okamžicích, kdy jsou všechny logické hodnoty na vstupu ustáleny. Výstupní hodnota synchronních obvodů je tedy určena přesně bez ohledu na možné hazardy v předcházejících obvodech.

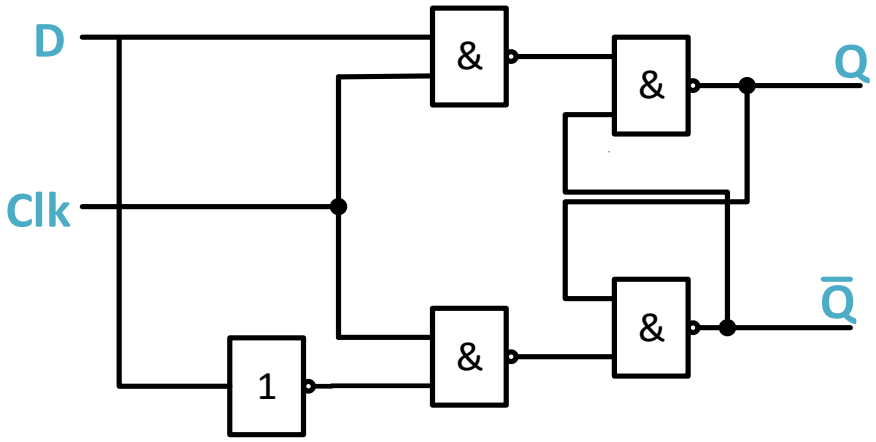
6.4.1 JEDNOSTUPŇOVÉ KLOPNÉ OBVODY



Obr. 43 Synchronní RS klopný obvod

Na Obr. 43 je upraven RS klopný obvod tak, aby pracoval jako synchronní. Obvod reaguje na vstupy R a S pouze tehdy, když vstup T je nastaven do log. 1. Vidíme, že synchronní obvody mají navíc oproti asynchronním obvodům další synchronizační vstup, který umožňuje znecitlivět ostatní vstupy až do doby, kdy jsou zajištěny podmínky pro správnou funkci obvodu (předcházení hazardům). Tento vstup bývá označen zpravidla zkratkou T, C nebo CLK, slovně jej popisujeme jako hodinový vstup. Hodinové vstupy bývají buzeny zpravidla periodickým signálem, tvořeným krátkými impulsy a nazývaným hodinový signál.

Další vylepšení synchronního RS klopného obvodu je uvedeno na Obr. 44. Aby bylo zabráněno vzniku zakázaného stavu, je signál R vytvořen ze signálu S (D) pomocí invertoru. Je vidět, že vstupní hodnoty 11 a 00 RS klopného obvodu nemohou nastat, obvod reaguje na hodnotu 0 i 1 na vstupu D a v případě, že na vstupu C je nastavena log.1, zapíše vstupní hodnotu z D na výstup.



Obr. 44 Hladinový klopný obvod typu D

Pravdivostní tabulka jednostupňového D klopného obvodu je v Tab. 28.

Tab. 28 Pravdivostní tabulka jednostupňového D KO

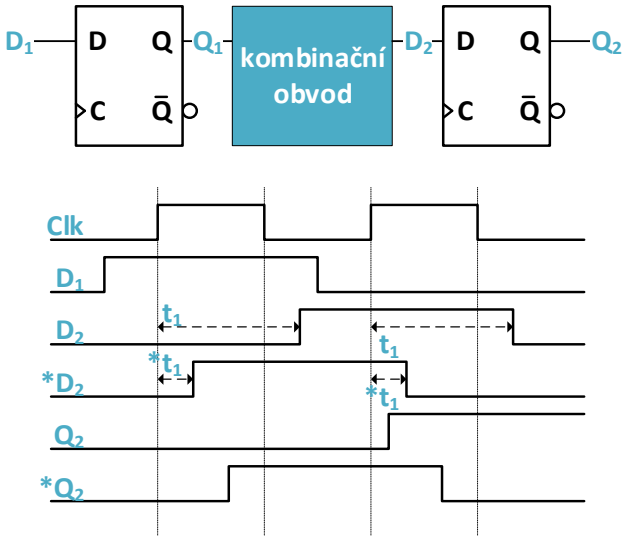
D	C	Q_{t+1}
0	0	Q_t
1	0	Q_t
0	1	0
1	1	1

Jednostupňový D klopný obvod je základním prvkem pro konstrukci statických pamětí. U těchto pamětí je k D klopným obvodům připojena dekódovací logika, která umožňuje adresaci. Jestliže je klopný obvod adresován a přijde povolení zápisu (signál C), pak se do obvodu zapíše hodnota na vstupu D, jestliže povolení zápisu nepřijde, pak paměť pracuje v režimu čtení a hodnota uložená v klopném obvodu je přepnuta na výstup z paměti. Synchronní obvody, které jsme doposud popsali, patří k těm nejjednodušším. Tyto obvody není možno využít při návrhu složitějších sekvenčních schémat, neboť zde dochází k hazardním stavům, jak je dokumentováno na příkladu zobrazeném na Obr. 45. Výstupní hodnota Q2 je ovlivněna rychlostí šíření signálu v kombinační logice a uvnitř prvního klopného obvodu. Rozdílné zpoždění signálu (znázorněno jako t1 a t1*

způsobí, že průběh vstupního signálu D2 může být změněn na D2* a tudíž i časový průběh výstupního signálu Q2 se změní na Q2*.

6.4.2 DVOUSTUPŇOVÉ KLOPNÉ OBVODY

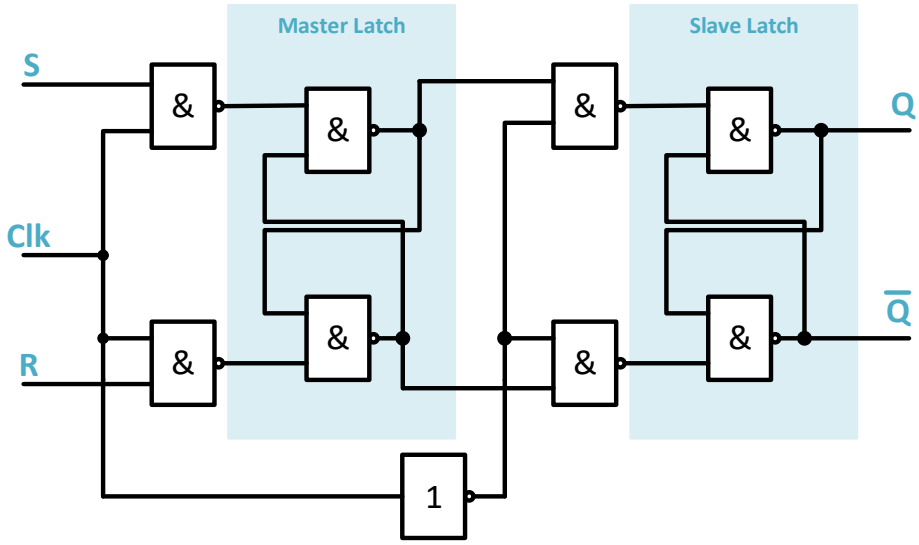
Při návrhu se většinou využívají dvoustupňové (Master-Slave) obvody (Obr. 46). Tyto obvody zaručují kromě definovaného okamžiku zápisu do klopného obvodu i definovaný okamžik rozšíření vnitřního stavu na výstup obvodu. Tato vlastnost je nezbytná pro konstrukci většiny sekvenčních obvodů, neboť umožňuje konstruovat zařízení tak, že v daný okamžik reagují pouze určené klopné obvody. Reakce synchronního Master-Slave obvodu po příchodu hodinového impulsu sestává ze dvou na sebe navazujících fází: zapamatování informace ze vstupu v prvním klopném obvodu a zpracování informace a její přenos na výstup v obvodu druhém.



Obr. 45 Problém hazardu vznikajícího ve vícestupňovém sekvenčním obvodu.

Na Obr. 46 je vyobrazeno ideové schéma master-slave RS klopného obvodu. První stupeň obvodu je řízen hodinovým signálem T, aktivním v log. 1, druhý stupeň je řízen invertovaným hodinovým signálem T.

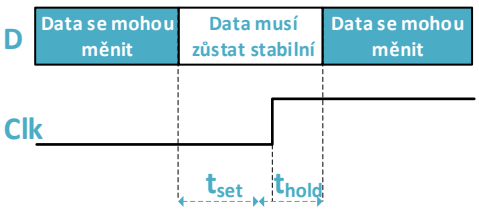
Počátek fáze zapamatování je vždy řízen hodinovým impulsem. Fáze přenosu může buď následovat bezprostředně po fázi zapamatování, nebo může být řízena hodinovým impulsem. Hodinové impulsy jsou buď přivedeny dvěma zvláštními vodiči ze zdroje hodinových signálů, nebo se uvnitř obvodu odvodí z náběžné a sestupné hrany jednoho vnějšího hodinového signálu.



Obr. 46 Master-slave RS klopný obvod

Používají se čtyři způsoby řízení synchronního sekvenčního obvodu:

- Řízení hladinou hodinového signálu (např. Obr. 44). Při hodinovém signálu v log. 1 je obvod transparentní – hodnoty vstupu se okamžitě přenášejí na výstup. Obvod si zapamatuje poslední hodnotu při aktivním hodinovém vstupu.
- Řízení hranou hodinového signálu (master-slave obvody Obr. 46). U obvodů řízených hranou se uvádějí parametry tset a thold, které uvadějí po jakou dobu před a po hraně hodinového signálu musí být data stabilní. Oba parametry jsou vyznačeny na Obr. 47. Ovody řízené hranó se dále dělí:
  - Náběžná hrana – změna výstupní hodnoty při změně hodin z 0 na 1.
  - Sestupná hrana – změna výstupní hodnoty při změně hodin z 1 na 0.

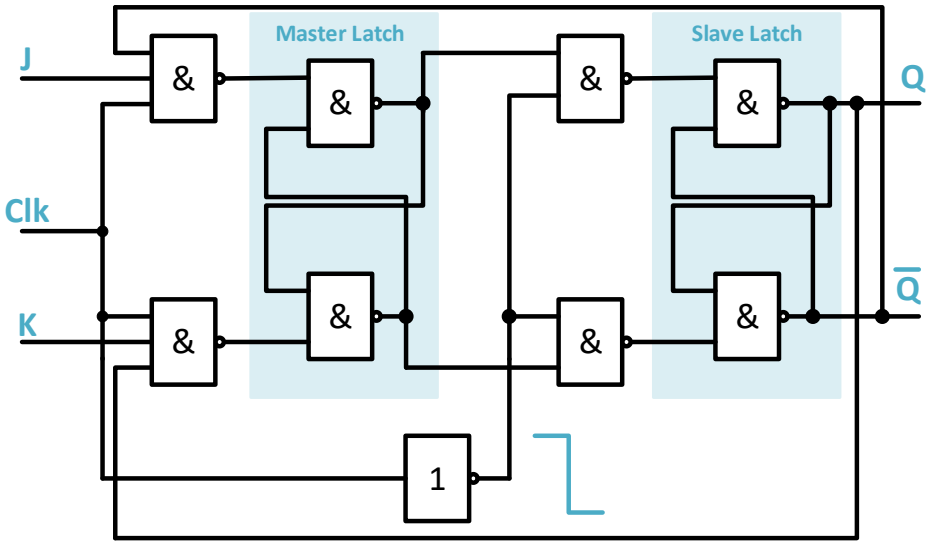


Obr. 47 Význam parametrů tset a thold

6.5 JK KLOPNÝ OBVOD

Principiální schéma JK klopného obvodu můžeme odvodit ze schématu Master-slave RS klopného obvodu, jak je to ukázáno na Obr. 46. Přidáním zpětné vazby z výstupů Q a Q/ na vstupní členy NAND (Obr. 48) je možno změnit chování obvodu na JK klopný obvod. Vstup J odpovídá vstupu S, K odpovídá vstupu R.

JK klopný obvod nemá žádný zakázaný stav, jestliže na oba vstupy J i K přivedeme v okamžiku aktivní úrovně hodinového signálu aktivní úroveň (log. 1) obvod se překlopí do opačného stavu, než byl před příchodem hodinového pulsu. Synchronní obvody jsou většinou konstruovány tak, že umožňují asynchronní nastavení do stavu log. 1 na výstupu (signál S) a asynchronní nulování (signál R). Pravdivostní tabulka JK klopného obvodu je uvedena v Tab. 29. Obvod JK je vhodný zvláště pro konstrukci čítačů vzhledem ke své schopnosti invertovat stav při příchodu logických hodnot 11 na vstupy JK.



Obr. 48 Master-Slave JK klopný obvod

Tab. 29 Pravdivostní tabulka JK klopného obvodu s asynchronním nastavením a nulováním

Režim	Vstupy					Výstupy		Poznámky
	S	R	C	J	K	Q_{t+1}	$\bar{Q}_{t+1}$	
Asynchronní	0	1	X	X	X	1	0	nastavení do 1
	1	0	X	X	X	0	1	nastavení do 0
	0	0	X	X	X	1*	1*	nestabilní stav
Synchronní	1	1	⊓	0	0	Q_t	$\bar{Q}_t$	beze změny
	1	1	⊓	1	0	1	0	nastavení do 1
	1	1	⊓	0	1	0	1	nastavení do 0
	1	1	⊓	1	1	$\bar{Q}_t$	Q_t	inverze stavu
	1	1	0	X	X	Q_t	$\bar{Q}_t$	beze změny

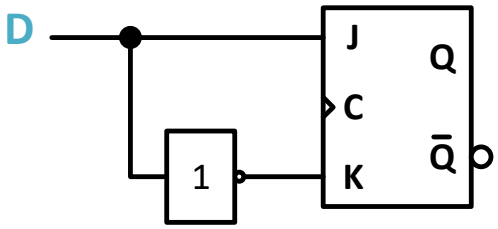
6.6 D A T KLOPNÉ OBVODY

D klopný obvod vznikne propojením a invertováním jednoho ze vstupů (vstup K) u JK klopného obvodu (Obr. 49). Pravdivostní tabulka Master-slave D klopného obvodu je shodná s pravdivostní tabulkou jednostupňového D klopného obvodu Tab. 28. D klopný obvod se používá zvláště pro konstrukci registrů a posuvných registrů.

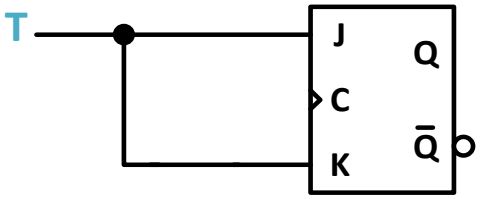
T klopný obvod vznikne z JK klopného obvodu spojením jeho vstupů (bez invertování). Úprava JK obvodu je naznačena na Obr. 50 a příslušná pravdivostní tabulka je v Tab. 30. Použití T klopného obvodu je časté zvláště při konstrukci čítačů.

Tab. 30 Pravdivostní tabulka T klopného obvodu

T	Q_t	Q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0



Obr. 49 D klopný obvod

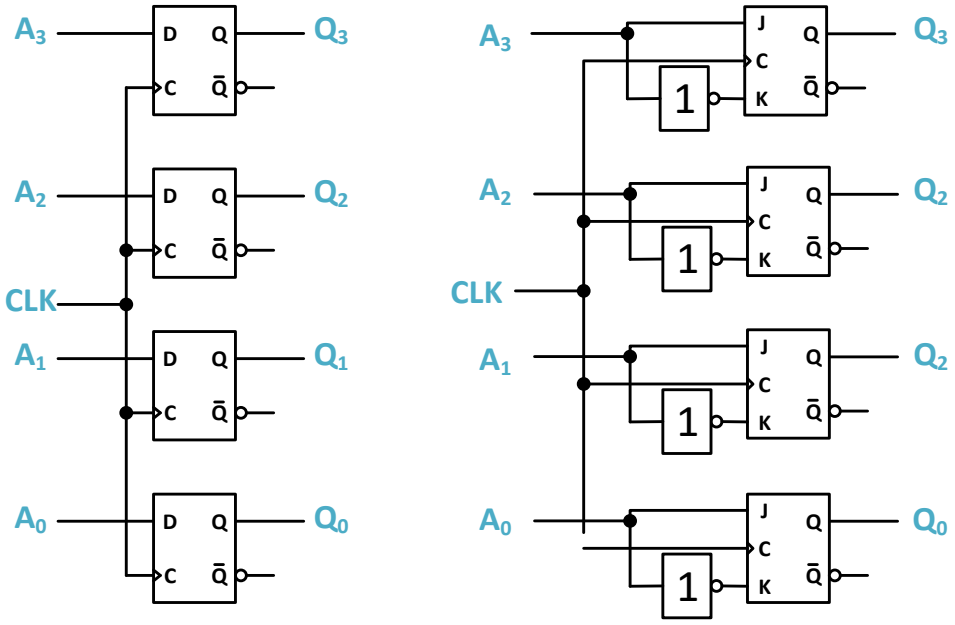


Obr. 50 T klopný obvod

7 VYŠŠÍ KONSTRUKČNÍ CELKY S KLOPNÝMI OBVODY

7.1 REGISTRY

N bitový registr je logický obvod s hodinovým vstupem, n informačními vstupy a s n výstupy. Může též obsahovat nulovací vstup, popřípadě nastavovací vstup. Hodinový impuls zajistí přenos okamžitých hodnot z informačních vstupů na výstup. Není-li přítomen, obsah výstupu se nemění. Registr lze realizovat jak členy D, tak členy JK. Principiální schéma čtyřbitového registru z obvodů D je na Obr. 51.

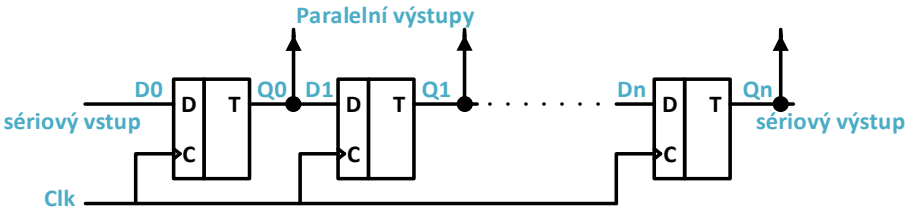


Obr. 51 Čtyřbitový registr tvořený hranovými klopnými obvody typu D a JK

Registr se mj. využívá k přenosu informace mezi dvěma kombinačními obvody. Registr zajistí, že po příchodu dostatečně zpožděného hodinového impulsu se zapamatují ustálené hodnoty výstupů předchozích částí obvodu.

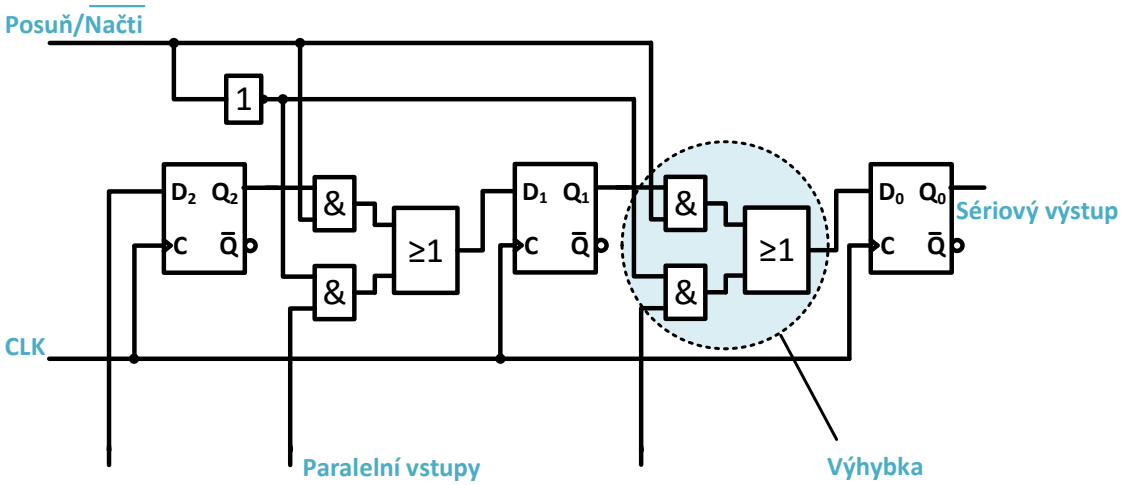
7.2 POSUVNÉ REGISTRY

Posuvný n bitový registr obsahuje hodinový vstup C, jeden informační vstup a jeden nebo dva informační výstupy (Obr. 52). Zpravidla bývá navržen tak, aby jej bylo možno jedním signálem nulovat. Posuvný registr obsahuje zpravidla D klopné obvody hranového typu a jeho funkce je následující: Po příchodu hodinového impulsu se obsah vstupu Di přenese na výstup Qi pro všechna i = 0, 1, …, n. Tato funkce současně posune informace o jedno místo vpravo nebo vlevo v závislosti na směru šíření signálu.



Obr. 52 Posuvný registr

Pomocí posuvných registrů lze snadno binárně dělit a násobit. Posun obsahu o jedno místo vpravo je podíl po dělení původního obsahu dvěma (se ztrátou zbytku po dělení), posun o jedno místo vlevo je výsledkem násobení původního obsahu dvěma. Posuvných registrů je více typů, liší se v počtu vstupů a výstupů. Běžně se využívá posuvných registrů se sériovým vstupem a výstupem a současně s paralelními vstupy nebo výstupy. Takovýto registr má dvě funkce, mezi kterými se přepíná pomocí zvláštního řídícího vstupu.



Obr. 53 Posuvný registr s paralelním zápisem

V základním režimu slouží jako běžný paralelní registr, ve druhém režimu slouží k sériovému zapsání, popřípadě přečtení obsahu registru pomocí jednoho vodiče. Využití takovéhoto registru je tam, kde je třeba paralelní informace přenést pomocí jednoho vodiče. Příklad zapojení je na Obr. 53. Vstup pro řízení funkce je nazván Posun/$\overline{\text{Načti}}$. Paralelní data je možno zapsat, jestliže je tento signál roven log. 0. Sériová data se zapisují prostřednictvím vstupu D3, sériové čtení dat je umožněno prostřednictvím výstupu Q0.

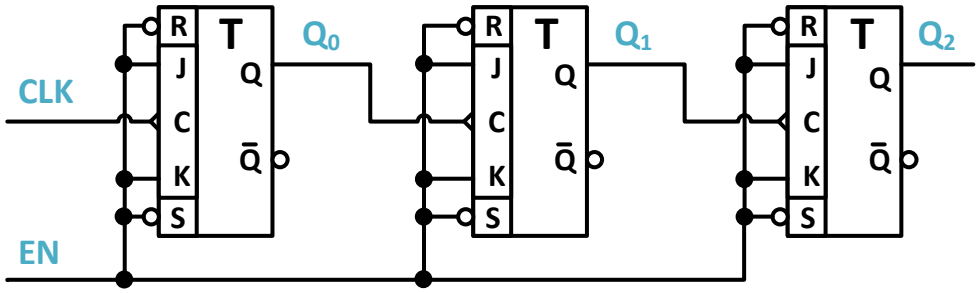
Pomocí posuvného registru s vhodně volenými zpětnými vazbami je možno vytvořit zvláštní čítače, které mají minimální hardwarové nároky na kombinační logiku mezi jednotlivými klopnými obvody a které jsou výhodné tam, kde vyžadujeme vysokou frekvenci hodinových pulsů. Specializované posuvné registry umožňují posun informace nalevo i napravo a využívají se ve výpočetní technice.

7.3 Čítače

Zjednodušeně řečeno, čítač je logický obvod, který zjišťuje počet došlých hodinových impulsů. Každý hodinový impuls změní jeho obsah, nejčastěji jej zvýší (sníží) o jednotku. Čítač však může čítat i jiným způsobem než v přirozeném pořadí binárního kódu (Grayův kód, binárně dekadický kód, další lineární i nelineární kódy). Čítače můžeme dělit podle počtu bitů, podle maximální frekvence hodinového signálu, podle typu kódu, který je čítačem generován, podle možnosti změny směru čítání nahoru nebo dolů nebo podle možností nastavení různých předvoleb.

Čítače se využívají jako základní konstrukční prvek pro sekvenční automaty různých typů. Tvoří například jádro řadiče procesoru (viz dále), využívají se pro měření kmitočtu a periody, vytváření časových základen osciloskopů, řízení multiplexorů, k dělení frekvence různých signálů atd. Čítače mohou být asynchronní nebo synchronní. Toto označení neznamená, že asynchronní čítače nemají hodinový (synchronizační) vstup, ale že jednotlivé stupně asynchronního čítače jsou synchronizovány jiným než hodinovým signálem. Principiální schéma asynchronního čítače je na Obr. 54. Vzhledem k nastavení vstupů J, K a R jednotlivé klopné obvody mají funkci děliče hodinové frekvence dvěma.

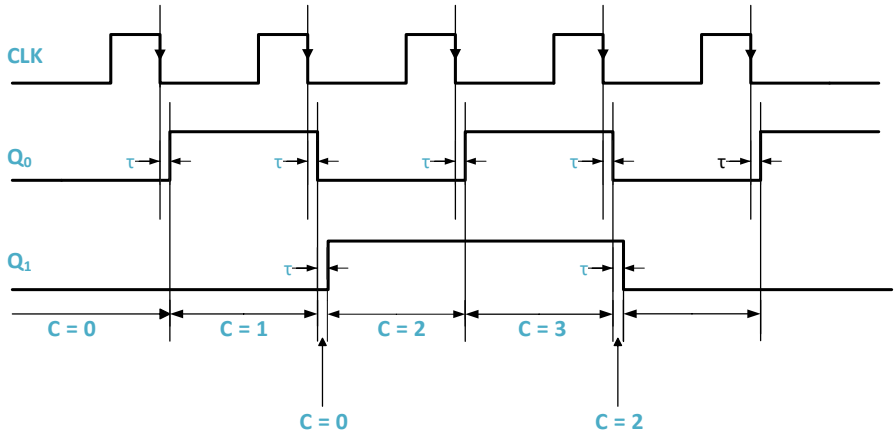
7.3.1 Asynchronní čítač



Obr. 54 Tříbitový asynchronní čítač tvořený JK klopnými obvody. Q_0 tvoří nejnižší řád.

Každý obvod překlopí do opačného stavu při příchodu sestupné hrany hodinového impulsu. Jelikož hodinové impulsy pro vyšší řády jsou odvozeny vždy ze stavu řádu nižšího, dochází na klopných obvodech k vytváření posloupnosti binárního kódu, přičemž nejnižší řád je nalevo. Asynchronní čítač je jednoduchý, není třeba konstruovat zvláštní obvody, které generují přenos z nižších do vyšších řádů. Nevýhodou tohoto zapojení je ta skutečnost, že jednotlivé bity čítače jsou nastavovány do správné kódové hodnoty v nestejných okamžicích. Tuto skutečnost dokumentuje časový diagram na Obr. 55.

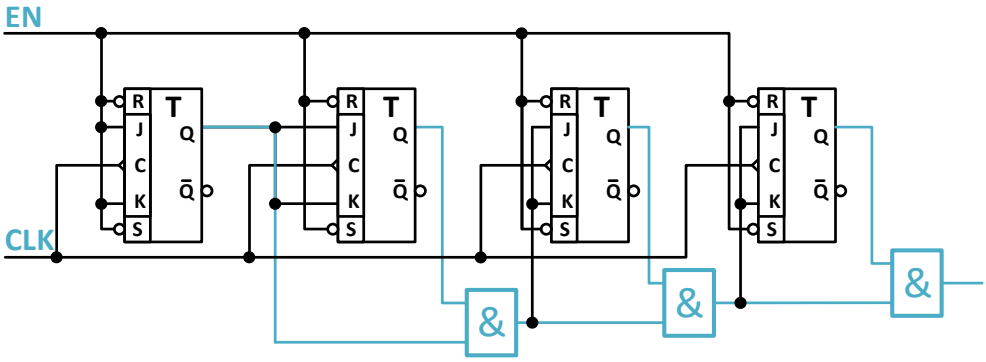
Na diagramu vidíme, že okamžik nastavení výstupu Q0 je zpožděn oproti sestupné hraně hodinového impulsu o dobu τ, výstup Q1 je zpožděn o dobu 2τ atd. Při větší délce čítače může dojít k tomu, že doba šíření signálu do nejvyššího řádu je delší, než je perioda hodinového signálu a čítač tudíž nebude zobrazovat správné hodnoty generovaného kódu.



Obr. 55 Časový diagram asynchronního čítače. Na diagramu je vyznačeno zpoždění signálu τ při průchody jednotlivými stupni čítače

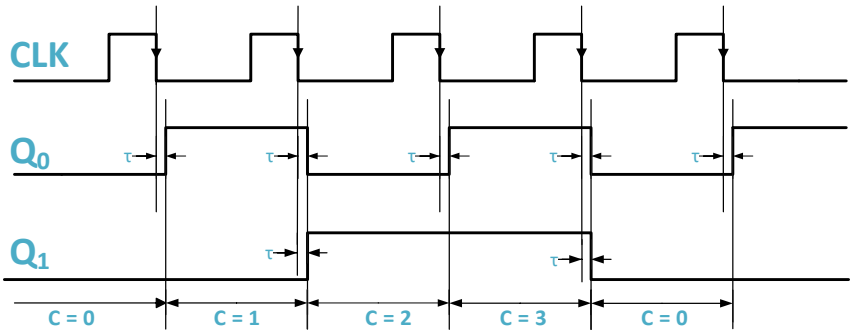
7.3.2 SYNCHRONNÍ ČÍTAČ

Principiální schéma čtyřbitového synchronního čítače je na Obr. 56. Z obrázku je patrné, že hodnota vyšších bitů čítače se změní při příchodu hodinového impulsu pouze tehdy, když v předchozích řádech byl po minulém hodinovém impulsu nastaven stav tvořený samými jedničkami.



Obr. 56 Synchronní čtyřbitový čítač tvořený T klopnými obvody

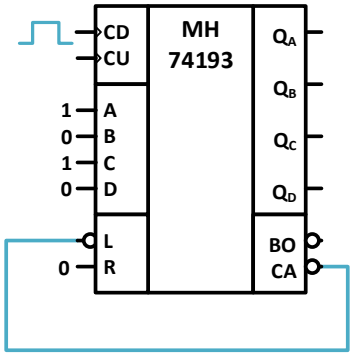
Výhodou synchronního čítače je to, že všechny bity čítače jsou nastavovány ve stejný okamžik. Přenos do vyššího řádu je nastavován kombinačními obvody, jejichž výstup se ustálí po nastavení nějakého stavu ještě před příchodem dalšího hodinového impulsu. Časový diagram je znázorněn na Obr. 57.



Obr. 57 Časový diagram dvou bitů synchronního čítače. Na diagramu je vyznačeno zpoždění signálu τ při průchody jednotlivými stupni čítače.

7.3.3 Zkrácení cyklu čítače

Úplným cyklem čítače rozumíme vzestupné nebo sestupné čítání v celém rozsahu čítače (např. 0 až 15 u čítače 74193). Neúplný cyklus je čítání v rozsahu N_1, N_2, kde $N_1 \geq 0$ a $N_2 \leq 15$. Např. pro $N_1 = 5$, $N_2 = 7$ bude na výstupu posloupnost čísel 5, 6, 7, 5, 6, ...



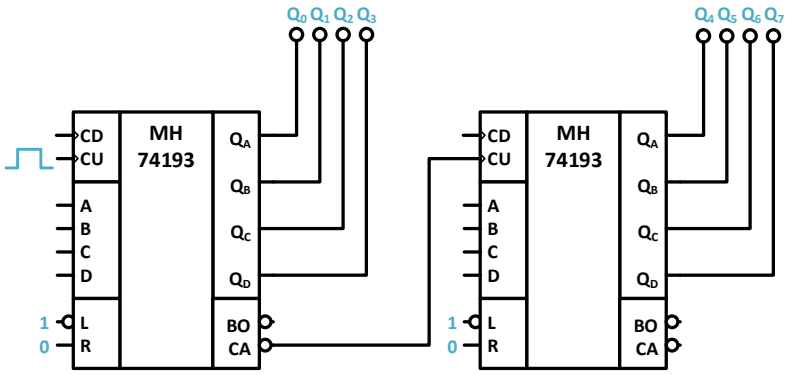
Obr. 58 Zkrácení cyklu čítače

Snadno lze realizovat neúplný cyklus pro čítání vpřed od N do 14 podle Obr. 58. Číslo N nazýváme předvolba. Předvolbu nastavíme na informačních vstupech A až D, nejnižší řád na svorce A. Na obr. B 7.5 je N = 5. Je-li na výstupu číslo 14, nastaví vzestupná hrana následujícího hodinového impulsu na výstupu číslo 15, ale sestupná hrana aktivizuje vstup L, výstup se přepíše číslem N a vzápětí přestane být výstup CA a tím i vstup L aktivním. Během hodinového impulsu je na výstupu číslo 15, ale po jeho ukončení je na výstupu číslo N.

7.3.4 Rozšířeni rozsahu čítače

Obvykle je rozsah jednoho čítače malý, pro zvětšení rozsahu se přidají další čítače, tím získáme vícestupňový čítač. V praxi se často používá dekadických čítačů 74192, pak každý stupeň odpovídá jednomu desítkovému řádu. Zapojení dvoustupňového čítače pro čítání vpřed je na Obr. 59. Hodiny se přivádějí na vstup prvního čítače. Jde-li o dekadický čítač, jsou na vstupu prvního

čítače jednotky a na výstupu druhého desítky. Spojením vstupů L je možnost předvolby na informačních vstupech obou čítačů.



Obr. 59 Rozšíření cyklu čítače

8 ARITMETICKÉ OBVODY

Obvody, zpracovávající aritmetické operace mezi vstupními daty mohou být jak sekvenční tak i pouze kombinační. V této kapitole uvedeme obvody a postupy využívané pro realizaci operace sčítání, odčítání, násobení a dělení.

Předpokládáme, že vstupní a výstupní data jsou vyjádřena v binární soustavě buď s pevnou, nebo pohyblivou řádovou čárkou.

V pevné řádové čárce bude číslo A rovno součtu součinů:

$$A = a_n 2^n + a_0 2^0 + a_{-1} 2^{-1} + \cdots + a_{-m} 2^{-m}$$

kde a_i je binární číslice nabývající dvou hodnot: 0 a 1. Číslo bude vyjádřeno pomocí příslušného počtu binárních číslic a_n … a_0, a_{-1} … a_{-m}. Mezi řádem „0“ a „-1“ píšeme „desetinnou“ čárku. Záporná čísla zobrazujeme pomocí některého z těchto kódů: přímý kód, kód s posunutou nulou, kód jednotkového doplňku a kód dvojkového doplňku.

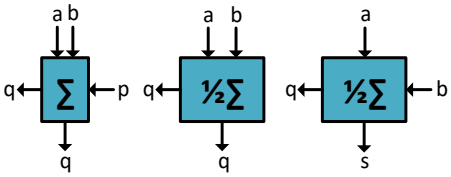
Pro zobrazení reálných nebo příliš velkých celých čísel je výhodné použít pohyblivou řádovou čárku. Potom číslo A bude ve tvaru $A = M.2^E$, kde M je mantisa a E exponent. Přesnost čísla A závisí na počtu číslic mantisy, rozsah zobrazení závisí na počtu číslic exponentu. Reálná čísla jsou zobrazována buď v jednoduché přesnosti pomocí 32 bitů (1 bit znaménko mantisy, 23 bitů mantisa v přímém kódu, 8 bitů exponent v kódu s posunutou nulou) anebo v dvojnásobné přesnosti 64 bitů (znaménko, 52 bitů mantisa, 11 bitů exponent). Pro vyčíslení výsledků aritmetických operací v pohyblivé řádové čárce jsou využity základní obvody, které zpracovávají aritmetické operace v pevné řádové čárce. Pro sčítání/odčítání je třeba srovnat exponenty a sečíst/odečíst mantisy. Pro násobení je třeba sečíst exponenty a vynásobit mantisy. Pro dělení je třeba odečíst exponenty a vydělit mantisy. Výsledek každé operace je třeba normalizovat, tj. vyjádřit takovým výrazem $M.2^E$, který zajistí na nejvýznamnějším bitu mantisy hodnotu log. 1.

8.1 SČÍTAČKY

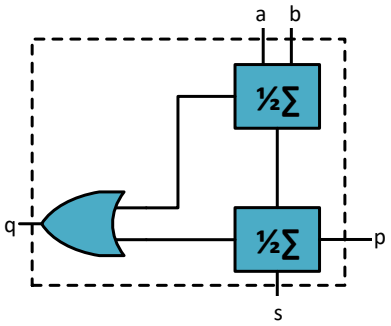
V tomto odstavci se budeme zabývat sčítáním operandů v pevné řádové čárce, pro sčítání v pohyblivé čárce je ke sčítačkám doplnit přídavné obvody realizující srovnání exponentů.

8.1.1 SČÍTAČKY PRO NEZÁPORNÁ ČÍSLA

U sčítaček nezáporných čísel rozlišujeme přenos do řádu i, který budeme označovat p_i přenos z řádu i, q_i přenos z posledního řádu q*(přenos anglicky nazýváme carry). K přeplnění sčítačky dojde tehdy, když q* =1. Přenos q* nazýváme přeplněním (overflow). Případné přeplnění je třeba při sčítání monitorovat a jeho výskyt ošetřit jako chybový výsledek.

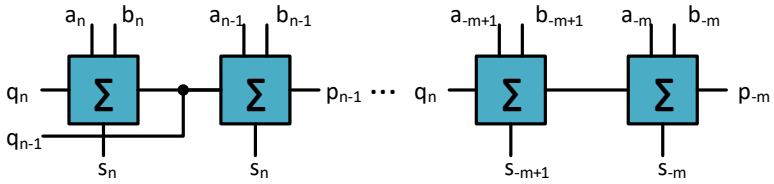


Obr. 60 Půlsčítačka a úplná sčítačka



Obr. 61 Úplná jednobitová sčítačka

Základním blokem sčítačky je úplná jednobitová sčítačka Obr. 60. Úplná jednobitová sčítačka se skládá ze dvou polosčítaček a obvodu logického součtu. Polosčítačka vytváří na výstupu s XOR ze vstupních bitů. Výstup q polosčítačky odpovídá logickému součinu vstupních proměnných polosčítačky. Propojením polosčítaček podle Obr. 60 získáme na výstupu s součet mod 2 vstupních proměnných, na výstupu q získáme logickou hodnotu odpovídající výrazu q = ab + ap + bp. Porovnáme-li výstupní hodnoty úplné jednobitové sčítačky s hodnotami, které odpovídají hodnotám součtu a přenosu do vyššího řádu v jednom řádu při sčítání binárních čísel, vidíme, že kaskádní spojení úplných jednobitových sčítaček bude realizovat operaci sčítání binárních čísel (Obr. 62).



Obr. 62 Paralelní sčítačka se sériovým přenosem

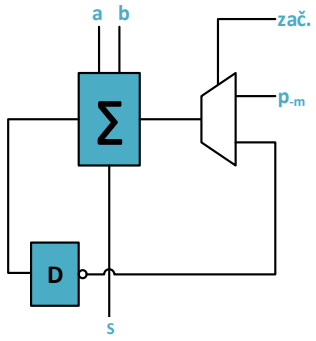
Př. 37 Binární sčítání

Sečtěte čísla 11_{10} a 6_{10}.

01011_2	11_{10}
00110_2	06_{10}
10001_2	17_{10}

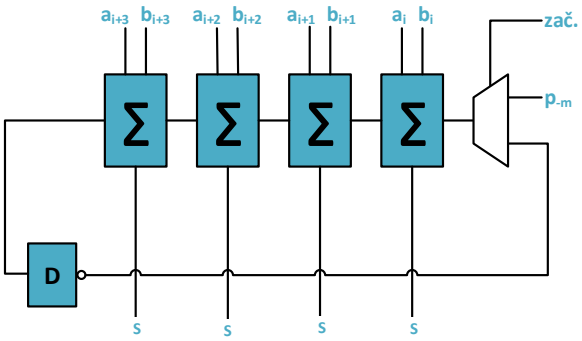
Vícebitové paralelní sčítačky vytvářejí vzhledem ke zřetězení jednobitových sčítaček značné zpoždění výstupu oproti některým vstupním proměnným. Je při tom otázkou, jak minimalizovat zpoždění sčítačky, popřípadě jak přizpůsobit sčítačku taktu hodinových pulsů, které synchronizují případné vstupní a výstupní registry. Maximální redukce nejdelšího zpoždění sčítačky je dosažena u sčítačky s predikcí přenosu. Tato sčítačka je vybavena rozsáhlým obvodem, který samostatně paralelním způsobem vypočítává přenosy mezi jednotlivými řády sčítačky a tím

dochází ke zkrácení nejdelšího zpoždění v obvodu. Toto schéma zde nebudeme rozebírat, řešení sčítačky s predikcí přenosu je možné najít např. v [1].



Obr. 63 Sériová sčítačka

Hardwarově nejjednodušší sčítačkou je sčítačka sériová. Její výhodou kromě jednoduchého zapojení je malé zpoždění v kombinační logice a tím i možnost vyšší frekvence taktování. Principiální schéma je uvedeno na Obr. 63. Sčítačka vyžaduje současné přivádění odpovídajících sečítaných bitů vícebitového čísla od nejnižšího řádu po nejvyšší, přičemž v klopném obvodě D se vždy uloží výstupní přenos, který je použit jako vstupní přenos pro následující zpracovávaný řád sčítání. Výhybka ve zpětné vazbě sčítačky přepíná mezi vnějším vstupním přenosem, který se může uplatnit u nejnižšího řádu sčítání a přenosy uchovávanými v klopném obvodě. Vzhledem k tomu, že sčítání dvou m-bitových čísel trvá u sériové sčítačky m hodinových taktů, je možné zvolit jako kompromisní řešení sério-paralelní sčítačku (Obr. 64).



Obr. 64 Sérioparalelní sčítačka

8.1.2 ODEČÍTÁNÍ

Pro odečítání s výhodou využijeme sčítačku popsanou v předchozím odstavci doplněnou o případné obvody vytvářející jednotkový nebo dvojkový doplněk z menšitele.

Jednotkový doplněk kladného čísla je číslo samo, ze záporného čísla získáme doplněk tím, že invertujeme všechny bity absolutní hodnoty čísla. Odečtení provedeme jako součet jednotkových doplňků kladného a záporného čísla a přičtením výstupního přenosu do nejnižšího řádu (kruhový přenos). Výsledek odečítání bude v kódu jednotkového doplňku.

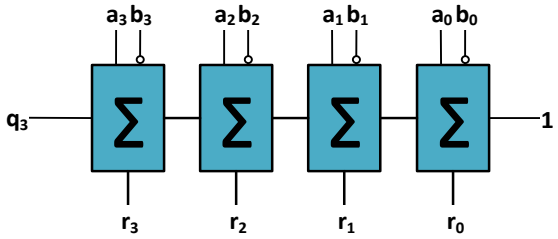
Př. 38 Odečítání s jednotkovým doplňkem

S pomocí jednotkového doplňku odečtěte číslo 6_{10} od čísla 11_{10}

01011_2	11_{10}
$\underline{11001_2}$	$\underline{-6_{10}}$
$1+00101_2$	5_{10}

Nevýhodou využití jednotkového doplňku je dvojí obraz nuly. Získáme kladnou a zápornou representaci nuly, z nichž jedna nemá využití.

Při použití dvojkového doplňku změníme u menšitele nuly na jedničky (a naopak) a přičteme jedničku, výstupní přenos se nevyužije. Předpokládejme, že od čísla A odečítáme číslo B.¨



Obr. 65 Odečítačka

Odečítání provedeme ve sčítačce, kterou doplníme invertory na vstupech bitů operandu B a vstupním přenosem do sčítačky, který zajistí přičtení jedničky do nejnižšího řádu (Obr. 65). Výsledek odečítání bude opět v kódu dvojkového doplňku.

Př. 39 Odečítání s dvojkovým doplňkem

S pomocí dvojkového doplňku odečtěte číslo 6_{10} od čísla 11_{10}

01011_2	11_{10}
$\underline{11010_2}$	$\underline{-6_{10}}$
$1/\ 00101_2$	5_{10}

Při odečítání se místo pojmu výstupní přenos q* používá pojem výpůjčka v* (borrow).

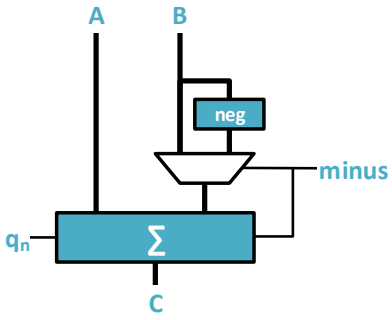
Platí $\overline{q*} = v*$

$q^*=1 \leftrightarrow v^*=0 \leftrightarrow A-B \geq 0$

$q^*=0 \leftrightarrow v^*=1 \leftrightarrow A-B<0$

Je-li výsledek záporný, jeho absolutní hodnotu je možné získat opět dvojkovým doplňkem.

Porovnáme-li schéma paralelní sčítačky s odečítačkou vidíme, že oba obvody můžeme sloučit do univerzálního schématu sčítačky-odečítačky (Obr. 66). Ze schématu je vidět, že sčítačka-odečítačka vytváří dvojkový doplněk operandu B a standardně sčítá dva operandy, které v případě, že jsou v kódu dvojkového doplňku, mohou být kladné i záporné, výsledek bude vždy správný, jestliže nedojde k jejímu přeplnění. Z tohoto důvodu je možné uvažovat o sčítačce jako o sčítačce-odečítačce pracující s operandy v kódu dvojkového doplňku.



Obr. 66 Sčítačka-odečítačka v dvojkové doplňku

8.1.3 Přeplnění sčítačky-odečítačky

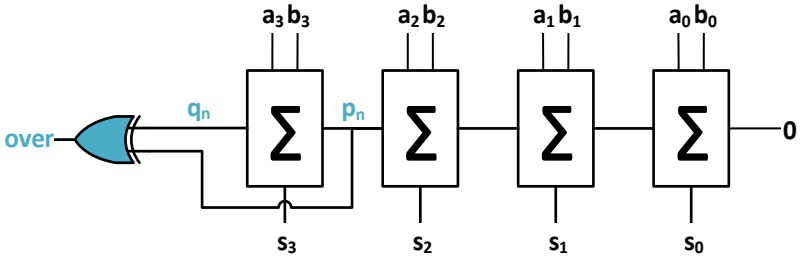
Přeplnění (overflow) není shodné s výstupním přenosem nebo výpůjčkou. Dochází k němu tehdy, když se sečtením dvou kladných čísel na výstupech vytvoří číslo záporné popřípadě, nebo když součet dvou záporných čísel vede na kladný výsledek. Tato situace nastává tehdy, když dvojkový doplněk součtu operandů je mimo povolený rozsah pro daný počet bitů. Vzhledem k tomu, že tuto skutečnost není možné odhadnout předem, je třeba provést výpočet a na základě vlastností výsledku rozhodnout, jestli k přeplnění došlo nebo ne.

Tab. 31 Pravdivostní tabulka úplné jednobitové sčítačky

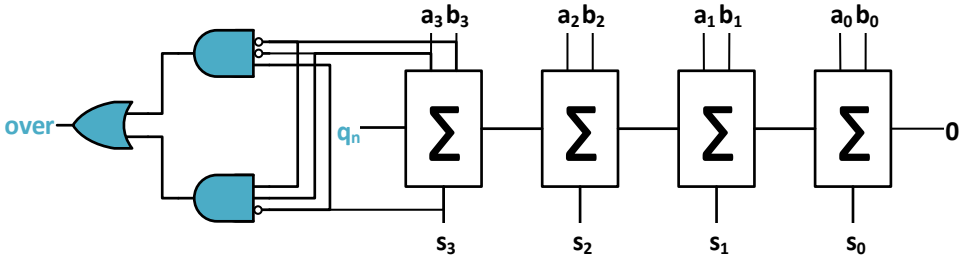
Si	a	b	p	q	S
S_0	0	0	0	0	0
S_1	**0**	**0**	**1**	**0**	**1**
S_2	0	1	0	0	1
S_3	0	1	1	1	0
S_4	1	0	0	0	1
S_5	1	0	1	1	0
S_6	**1**	**1**	**0**	**1**	**0**
S_7	1	1	1	1	1

V Tab. 31 je znázorněna pravdivostní tabulka úplné jednobitové sčítačky v nejvyšším řádu sčítačky-odečítačky. Barevně jsou vyznačeny ty řádky tabulky, které representují přeplnění. Pro stavový index

S1 jsou vstupní bity a, b, p rovny 0,0,1 výstupy q a s jsou rovny 0 a 1. Znamená to, že dvě kladná čísla jsou sečtena a výstupní přenos signalizuje kladný výsledek, ale nejvyšší bit součtu je roven 1 tzn. výsledek je záporný. Podobně pro S6 jsou vstupy a, b, p rovny 1,1,0, výstupy q a s jsou 1 a 0. Výstupní přenos signalizuje záporný výsledek, ale výsledek je kladný. Nutnou a postačující podmínkou přeplnění je tedy vztah p≠q anebo vztah $\bar{a}\bar{b}s + ab\bar{s} = 1$. Schéma sčítačky–odečítačky indikující přeplnění je na Obr. 67 a Obr. 68.



Obr. 67 Generování příznaku přeplnění s pomocí XOR



Obr. 68 Generování příznaku přeplnění s pomocí struktury AND-OR

8.2 NÁSOBENÍ VE DVOJKOVÉ SOUSTAVĚ

Násobičky jsou často užívanými obvody, jelikož pomocí nich je možné realizovat filtry signálů při jejich zpracování. Rozdělujeme je na sériové a paralelní.

8.2.1 SÉRIOVÁ NÁSOBIČKA

Při sériovém násobení využíváme faktu, že v binární soustavě je aritmetické násobení dvou bitů rovno jejich logickému součinu. Operandy A a B a výsledek násobení C vyjádříme pomocí následujících výrazů:

$$A = \sum_{i=0}^{m} a_i z^i$$

$$B = \sum_{j=0}^{m} b_j z^j$$

$$C = A \cdot B = \sum_{j=0}^{m+n+1} c_j z^j$$

$$C = \sum_{j=0}^{n} A b_j z^j$$

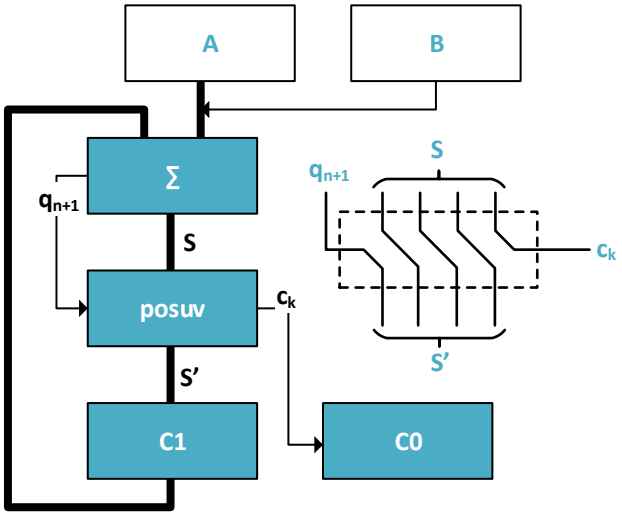
Poslední z uvedených výrazů je využitelný pro sériové násobení čísla A koeficienty z čísla B a postupnému přičítání výsledků k odpovídajícím řádům výsledku.

Př. 40 Ruční násobení v binární soustavě

Znásobte čísla 7_{10} a 5_{10}

A:	111_2	07_{10}
B:	101_2	05_{10}
	111	
	000	
	111	
C:	100011_2	35_{10}

Násobení provádíme pomocí operací logického součinu přičítání a posunů v registrech C0 a C1. Počáteční hodnota registrů C0 a C1 je nulová. V prvním kroku k registru C0 přičteme výsledek logického součinu LSB čísla B a všech bitů čísla A. Výsledek posuneme o jednu pozici doprava ve spojeném posuvném registru C0C1. Proces opakujeme pro další bity čísla B. Po třech cyklech se vytvoří v registrech C0C1 výsledek C.



Obr. 69 Sériová násobička

Tab. 32 Operace násobení

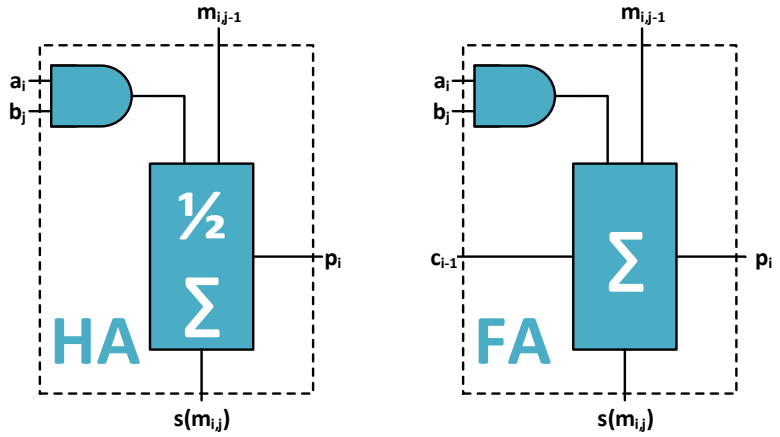
operace	C1	C0
přičtení 111	0111	000
Posuv	0011	1000
přičtení 000	0011	100
Posuv	0001	110
přičtení 111	1000	110
Posuv	0100	011

8.2.2 PARALELNÍ NÁSOBIČKA

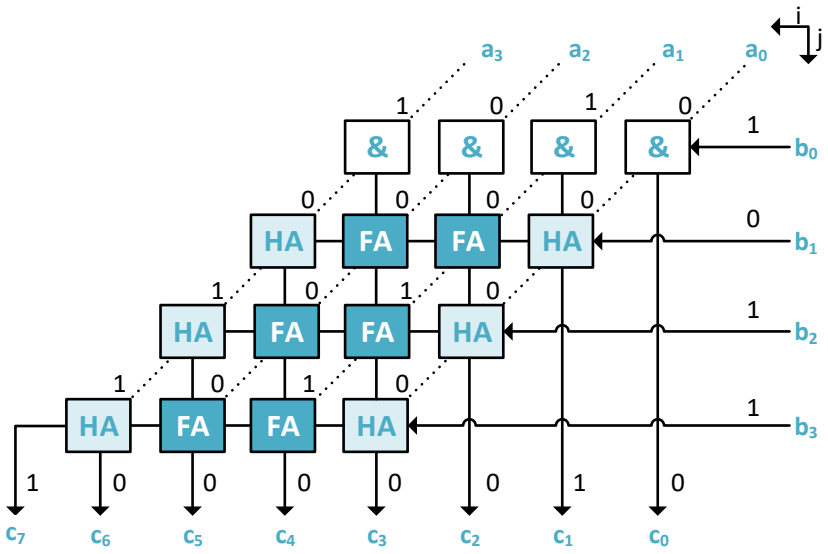
Násobení se provádí během jediného taktu hodin. Využívá se při tom následujícího rozkladu výsledku C:

$$C = \sum_{i=0}^{m} \sum_{j=0}^{n} a_i b_j 2^{i+j}$$

Výraz $a_i.b_j$ ve dvojkové soustavě realizujeme jako log. součin, 2^{i+j} representuje posuv o i+j míst doleva. Příklad realizace 4bitové paralelní násobičky je uveden v Obr. 71. Při realizaci násobičky pomocí logických obvodů vzniká pravidelná struktura se součinovými hradly (v Obr. 71 označeno &) a úplnými jednobitovými sčítačkami (na Obr. 70 označeny jako FA) a polosčítačkami (HA tamtéž).



Obr. 70 Funkční bloky paralelní násobičky



Obr. 71 Paralelní násobička

8.3 DĚLENÍ VE DVOJKOVÉ SOUSTAVĚ

Zatímco násobení v sériové násobičce bylo prováděno pomocí operací sčítání a posunu, dělení se provádí pomocí operace odčítání a posunů. Dělení vyžaduje složitější řízení výpočtu, a proto jej není vhodné provádět pomocí kombinačního obvodu. Výjimku tvoří dělení číslem 2m, neboť to odpovídá pouze posunu řádové čárky a tyto posuny se kombinačním obvodem snadno provedou. Rozlišujeme algoritmy dělení s návratem přes nulu a bez návratu přes nulu.

Při dělení s návratem přes nulu postupujeme obdobně jako při dělení dekadických čísel algoritmem známým z primárního vzdělávání. Po srovnání řádových čárek odečteme od dělence dělitele, v případě nezáporného zbytku získáme nejvyšší bit výsledku roven jedné, v opačném

případě nule. Při záporném zbytku provedeme rekonstrukci dělence tzv. návratem, který spočívá ve zpětném přičtení původně odečtené hodnoty. Od upravené hodnoty dělence odečteme dělitele na o jeden řád posunuté pozici. Provádíme vyčíslení podílu popř. návrat. Opakujeme do vyčerpání platných řádů, popřípadě podle počtu vyžadovaných platných bitů.

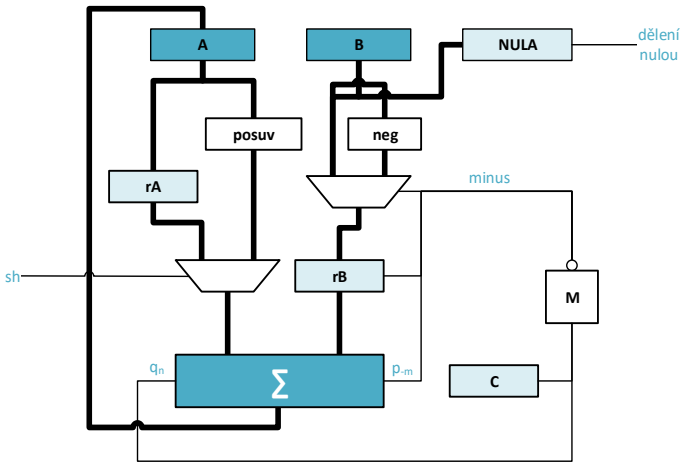
Př. 41 Dělení ve dvojkové soustavě

Vydělte čísla $0{,}101_2$ a $0{,}110_2$

	1	0	1	:	1	1	0	=	0	,	1	1	0
-	1	1	0										
		-	1						0				
+	1	1	0										
	1	0	1										
	-	1	1	0									
		1	0	0							1		
		-	1	1	0								
			1	0	0							1	
			-	1	1	0							
				-	1	0							0
			+	1	1	0							
				1	0	0							

odečtení dělitele
záporný zbytek, 1. bit podílu
návrat
zrekonstruovaná hodnota dělence
odečtení dělitele posunutého o jeden řád doprava
kladný zbytek, 2. bit podílu
odečtení dělitele posunutého o jeden řád doprava
kladný zbytek, 3. bit podílu
odečtení dělitele posunutého o jeden řád doprava
záporný zbytek 4. bit podílu
návrat
zbytek

Při dělení bez návratu přes nulu není prováděna rekonstrukce dělence při záporném zbytku, ale rovnou je k tomuto zbytku přičtena hodnota dělitele posunutého o příslušný počet řádů odpovídající vyčíslovanému bitu podílu. Existuje celá řada různých způsobů urychlujících provádění dělení binárních čísel. Například je možné využívat iteračních postupů, které při kombinaci tabelování vhodných počátečních hodnot iterace umožní získat podíl za menší počet taktu, než odpovídá počtu bitů dělence. Je třeba si uvědomit, že operace dělení je mnohem složitější na HW realizaci (např. Obr. 72) a dobu výpočtu než operace násobení, a proto je vhodné minimalizovat její provádění pro aplikace vyžadující rychlou odezvu.

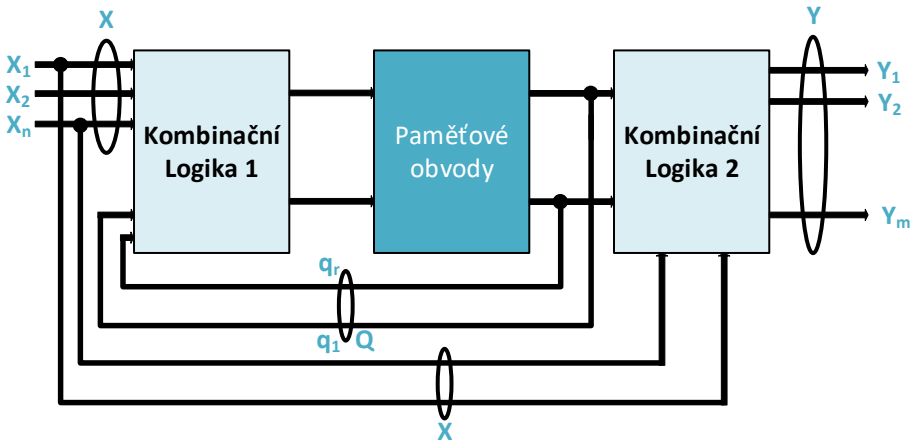


Obr. 72 Sekvenční dělička

9 NÁVRH SYNCHRONNÍCH SEKVENČNÍCH OBVODŮ

9.1 SEKVENČNÍ AUTOMATY

Obecně všechny systémy, kde počítáme s časem, jakožto s faktorem podmiňujícím chování jsou sekvenční. V praxi tuto sekvenčnost chování zanedbáváme a hovoříme například o kombinačních obvodech, kde při jejich zjednodušeném popisu nehovoříme o zpoždění průchodu signálu obvodem. Časovou podmíněnost výstupu však nelze zanedbat u obvodů, kde výstupy jsou podmíněny okamžitým vnitřním stavem. Sekvenční obvod (systém) je určen množinou vstupních symbolů (X) tzv. vstupní abecedou, množinou výstupních symbolů (Y) tzv. výstupní abecedou, množinou vnitřních stavů (Q), přechodovou funkcí (δ) a výstupní funkcí (λ). Přechodová funkce určuje, za jakých podmínek přechází automat mezi jednotlivými stavy a výstupní funkce určuje, jaké výstupní symboly jsou přiřazeny jednotlivým stavům a vstupům. Jestliže požadujeme, aby automat byl na začátku své funkce v definovaném stavu (Q0) je informace o tomto stavu také vyžadována pro popis chování automatu. Z kombinačního obvodu vytvoříme obvod sekvenční tak, že do obvodu zařadíme zpětnou vazbu, která hodnoty z výstupů nebo hodnoty některých vnitřních proměnných přivede k dalšímu zpracování některým kombinačním podobvodem, který je ve schématu obvodu blíže ke vstupům. Takto vytvořený systém může trpět tím, že zpracování signálů v různých částech kombinační sítě netrvá stejnou dobu, a tudíž mohou vznikat různé hazardní jevy, které se obtížně předvídají a odstraňují. Z těchto důvodů jsou zpětné vazby v systému doplněny o synchronní klopné obvody, které zabezpečí, že systém reaguje na vstupy vždy bezhazardně. Obecné schéma automatu je na Obr. 73.



Obr. 73 Obecné schéma sekvenčního automatu

Při využití popisu automatu pomocí uspořádané pětice <X, Y, Q, δ, λ> můžeme rozlišit různé typy automatů.

Jestliže některá funkce nebo některé proměnná chybí, hovoříme o speciálních automatech:

Automat, který je definován čtveřicí < Y, Q, Γ, Φ> bude automatem autonomním. Za tento automat můžeme považovat například zdroj hodinových impulsů. Takový automat po zapnutí napájecího zdroje pracuje bez vstupních řídících signálů.

Automatem bez vnitřních stavů by byl kombinační obvod, a proto se jím zde nebudeme zabývat.

Automat bez výstupu se nazývá automatem Medveděvovým. Takový automat nemá praktické využití, protože jeho chování je z vnějšku nepozorovatelné.

Automat, který má všechny proměnné a funkce může být buď typu Moore nebo Mealy. Přehled typů automatů nalezneme v Tab. 33.

Tab. 33 Typy sekvenčních automatů

	Typ automatu	Určení automatu	Poznámka
1	Automat kombinačního typu – kombinační logický systém	$A = \{X, Y, \delta=F\}$	Automat má jeden konstantní stav (vnitřní).
2	Medvěděvův "konečný" automat – automat bez výstupního zařazení	$A = \{X, Q, \delta\}$ $Q_t = \delta(X_t, Q_{t-1})$	Výstupní chování automatu nelze pozorovat.
3	Mooreův automat	$A = \{X, Y, Q, \delta, \lambda_0\}$ $Q_t = \delta(X_t, Q_{t-1})$ $Y_t = \lambda_0(Q_t)$	Výstupní stav je závislý na vnitřním stavu automatu. Převoditelný na Mealyho.
4	Autonomní automat	$A = \{Y, Q, \delta, \lambda\}$ $Q_t = \delta_0(Q_{t-1})$ $Y_t = \lambda_0(Q_t)$	Konstantní chování z hlediska "vstupů".
5	Mealyho automat	$A = \{X, Y, Q, \delta, \lambda\}$ $Q_t=\delta(X_t, Q_{t-1})$ $Y_t=\lambda(X_t, Q_{t-1})$	Nejobecnější automat, výstupní stav je závislý na vnitřním stavu a na vstupu, převoditelný na Moora.
6	Stochastický, pravděpodobnostní automat	$A = \{X, Y, Q, \delta, \lambda\}$ $\delta:P_\delta\{Q_j / Q_i, X_1\}$ $\lambda:P_\lambda\{Y_m / Q_i, X_1\}$	Přechodové a výstupní funkce jsou pravděpodobnostmi P_δa P_λ. Realizace je deterministická.

Chování automatu typu Moore je dáno následujícími rovnicemi:

$Q_t = \delta(X_t, Q_{t-1})$ přechodová funkce
$Y_t = \lambda_0(Q_t)$ výstupní funkce

Uvedené rovnice říkají, že stav v časovém okamžiku t je určen vstupem v tomtéž čase a předchozím stavem (čas t-1), výstup v čase t je funkcí současného stavu. Jelikož funkce λ je kombinační funkcí, platí, že pro různé výstupy musí existovat různé vnitřní stavy.

Chování automatu typu Mealy je dáno následujícími rovnicemi:

$Q_t = \delta(X_t, Q_{t-1})$ přechodová funkce
$Y_t = \lambda_0(X_t, Q_{t-1})$ výstupní funkce

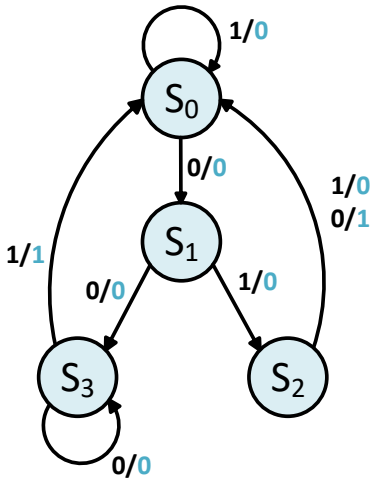
Automat typu Mealy je tedy podobný automatu Moorovu s tím rozdílem, že výstup je funkcí aktuálního vstupu a stavu v předchozím taktu. Různým výstupům tedy nemusí odpovídat různé vnitřní stavy.

Z praktického hlediska existují dva základní rozdíly mezi oběma automaty:

- Automat typu Mealy je zpravidla jednodušší než automat Mooreův, protože pro vytvoření požadované výstupní sekvence není třeba navrhovat pro každou hodnotu požadovaného výstupu, který má být odezvou na vstupní sekvenci, odpovídající vnitřní stav.
- Automat typu Mealy reaguje na vstupní sekvenci dříve než automat typu Moore. Tato vlastnost může být nevýhodou, protože okamžik, kdy automat zareaguje na změnu vstupu, není přesně definován, záleží na vnějším vstupním signálu, který nemusí být přesně synchronizován s hodinami automatu.

9.2 Popis chování automatů

Sekvenční automaty lze zadávat několika způsoby. Nejčastěji se popisují pomocí grafu přechodů a tabulky přechodů a výstupů. Graf přechodů umožňuje snazší pochopení funkce navrhovaného automatu, a proto je vhodné jej využít ve fázi návrhu chování automatu. Tabulka je základem pro další syntézní kroky, a proto je vhodné ji využít v těch případech, kdy manuálně provádíme celou syntézu. Vlastní syntéza automatu je poměrně složitou úlohou zvláště pro automaty, které mají velký počet vnitřních stavů. Některé automatizované návrhové systémy však tuto úlohu řeší autonomně na základě zadaného grafu přechodu, návrhář tedy nemusí další syntézu provádět.

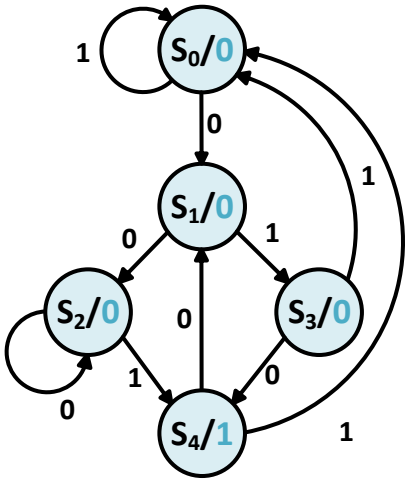


Obr. 74 Graf přechodů Mealyho automatu

Graf přechodu automatu typu Mealy je na Obr. 74. Uzly grafu odpovídají stavům (Q), hrany grafu možným přechodům mezi stavy (δ) a ohodnocení hran odpovídá vstupním podmínkám nutným pro přechod mezi stavy (X) a odpovídajícímu výstupu (Y). Odpovídající tabulka přechodů a výstupů je v **Tab. 34**.

Tab. 34 Tabulka přechodů a výstupů Mealyho automatu

Stav Q_t	Vstup N	Budoucí stav Q_{t+1}	Výstup Z
S0	0	S1	0
	1	S0	0
S1	0	S2	0
	1	S3	0
S2	0	S2	0
	1	S0	1
S3	0	S0	0
	1	S0	1



Obr. 75 Graf přechodů Mooreova automatu

Graf přechodu automatu typu Moore je na Obr. 75. V uzlech grafu jsou vepsány kromě pojmenování odpovídajícího stavu i hodnoty příslušného výstupu (Y). Odpovídající tabulka přechodů a výstupů je v Tab. 35.

Tab. 35 Tabulka přechodů a výstupů Mooreova automatu

Stav Q_t	Vstup N	Budoucí stav Q_{t+1}	Výstup Z
S0	0	S1	0
	1	S0	0
S1	0	S2	0
	1	S3	0
S2	0	S2	0
	1	S4	0
S3	0	S0	0
	1	S4	0
S4	0	S1	1
	1	S0	1

Při zadávání sekvenčního systému se často stane, že tabulka přechodů a výstupů obsahuje vzájemně ekvivalentní stavy. Tyto stavy mají pro všechny vstupní symboly definovány přechody do stejného stavu (nebo do stavů o kterých víme, že jsou ekvivalentní) a zároveň všechny výstupní symboly jsou pro všechny přechody stejné. Ekvivalentní stavy můžeme nahradit jediným

representantem celé třídy ekvivalentních stavů. Dojde tak ke snížení počtu stavů automatu a tím často ke snížení počtu klopných obvodů automatu při zachování stejné funkčnosti.

9.3 Převody mezi automaty

Automaty Mealy a Moore jsou navzájem na sebe převoditelné, jestliže mají shodnou vstupní i výstupní abecedu. Znamená to, že každý z těchto automatů je možné nahradit typem druhým, přičemž pro libovolnou vstupní sekvenci dostaneme shodnou sekvenci výstupní jako u automatu původního.

Přeměna automatu Moore na automat Mealy:

Mějme automat A typu Moore charakterizovaný pětící <X, Y, Q, δ, λ>. Jestliže chceme vytvořit ekvivalentní automat typu Mealy můžeme využít stejnou množinu stavů Q a stejnou přechodovou funkci mezi stavy δ. Podle definice chování automatu vidíme, že výstupní funkce λ se musí lišit, protože má jiné vstupní parametry. Označme tedy výstupní funkci automatu Mealy symbolem λ´. Automat Mealy s označením A´ bude tedy charakterizován pětící <X, Y, Q, δ, λ´>. Aby oba automaty byly ekvivalentní (A=A´) musí platit:

$$\lambda'(X^t Q^{t-1}) = \lambda\big(\delta(X^t Q^{t-1})\big) = \lambda, Q^t = Y^t$$

V Tab. 36 a Tab. 37. jsou tabulky přechodů a výstupů obou ekvivalentních automatů. Vidíme, že přechody mezi stavy jsou shodné a výstupy automatu Mealy pro jednotlivé stavy a vstupy odpovídají výstupům automatu Moore pro stavy, do kterých by automat přešel v následujícím taktu při aplikování příslušného vstupu.

Tab. 36 Tabulka přechodů a výstupů Mooreova automatu (ekvivalentní s Mealym)

Přechody automatu			Výstup
Q/X	X1	X2	Z
Q1	Q3	Q1	Y3
Q2	Q1	Q2	Y1
Q3	Q2	Q3	Y2

Tab. 37 Tabulka přechodů a výstupů Mealyho automatu (ekvivalentní s Moorem)

Přechody automatu			Hodnota výstupu při vstupech X1 a X2	
Q/X	X1	X2	X1	X2
Q1	Q3	Q1	Y2	Y2
Q2	Q1	Q2	Y3	Y1
Q3	Q2	Q3	Y1	Y2

V případě, že přeměňujeme Mooreův automat na Mealyho může dojít k tomu, že bude obsahovat ekvivalentní stavy, takže nebude minimální. V tom případě je možné v dalším kroku provést redukci stavů.

9.3.1 Přeměna automatu Mealy na Moore

Automat typu Mealy může mít nedostatečný počet vnitřních stavů k tomu, aby bylo možné splnit podmínku ekvivalence. Tato podmínka je splnitelná tehdy, když pro všechny stavy platí, že do každého uzlu grafu přechodu vedou pouze hrany ohodnocené stejným výstupním symbolem.

Tato podmínka není splněna např. v příkladu na Obr. 74, kde do uzlu A vstupují hrany ohodnocené různě. Při vytváření ekvivalentního grafu automatu Moore nahradíme každý uzel, který nemá uvedenou vlastnost tolika uzly, kolika výstupními symboly jsou ohodnoceny hrany do něho vstupující. Uzly ohodnotíme příslušnými výstupními symboly, které odpovídají výstupním symbolům uvedeným u příslušných vstupujících hran. Nakonec připojíme vstupní a výstupní hrany těch uzlů, které jsme doplnili, tak, aby byla zachována návaznost vnitřních stavů automatu. Příklad automatu Moore, který je ekvivalentní Mealyho automatu z Obr. 74 je na Obr. 75.

9.4 Postup syntézy automatu

Postup návrhu automatu sestává z následujících kroků:

- Volba typu automatu
- Sestavení grafu přechodů automatu
- Sestavení tabulky přechodů a výstupů
- Redukce stavů
- Přiřazení stavů automatu vnitřním proměnným navrhovaného obvodu
- Sestavení budící tabulky pro zvolené typy klopných obvodů
- Návrh propojení obvodu

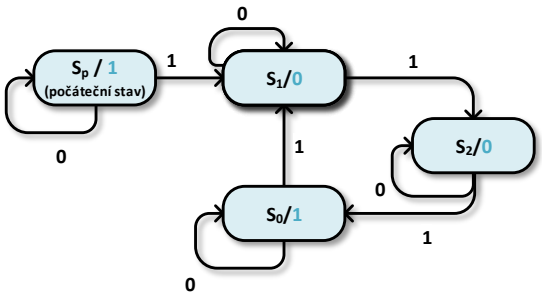
Volba typu automatu

Automat typu Mealy je konstrukčně jednodušší, Mooreův automat je lépe synchronizován. Volba typu tak závisí na podmínkách využití automatu, na součástkové základně, která bude použita pro konstrukci automatu.

Sestavení grafu přechodů automatu

Graf přechodu automatu sestavíme podle slovního zadání, které má charakterizovat požadované výstupní sekvence pro dané sekvence vstupů. Postup budeme demonstrovat na příkladu.

Slovní zadání: Sestavte čítač modulo 3. Čítač v případě vstupního signálu N rovném log. 1 cyklicky generuje na svém výstupu Y log. 1 v případě, že počet hodinových taktů od počátku činnosti je roven 0 modulo 3. V ostatních případech je na výstupu log. 0. Graf přechodů je vyobrazen na Obr. 76.



Obr. 76 Graf přechodů automatu pracujícího jako čítač modulo 3

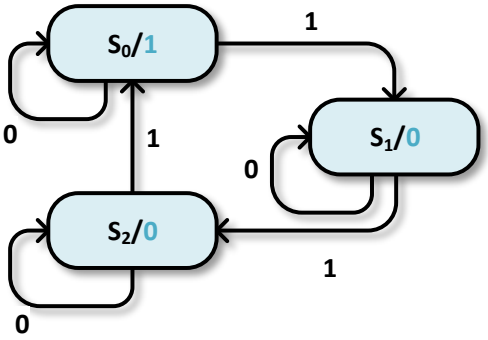
Na grafu přechodu vidíme 4 stavy, z nichž jeden je počáteční a dalšími třemi bude automat cyklicky procházet, při průchodu stavem S0 bude na výstupu generovat log. 1.

Sestavení tabulky přechodů a výstupů

Tabulka odpovídající grafu přechodu automatu je v Tab. 38.

Tab. 38 Tabulka přechodů a výstupů automatu pracujícího jako čítač modulo 3

Stav Q^t	Vstup N	Budoucí stav Q^{t+1}	Výstup Y
Sp	0	Sp	1
	1	S1	
S0	0	S0	1
	1	S1	
S1	0	S1	0
	1	S2	
S2	0	S2	0
	1	S0	



Obr. 77 Redukovaný graf přechodů automatu pracujícího jako čítač modulo 3

Redukce stavů

Tab. 38 obsahuje dva ekvivalentní stavy. Jedná se o stav počáteční a stav S0. Důvodem pro toto tvrzení je fakt, že oba stavy mají následující stavy stejné a zároveň mají přiřazené shodné výstupy. Z toho důvodu odstraníme stav nazvaný počáteční. Výsledný graf přechodu a tabulka přechodu a výstupů je na Obr. 77 resp. Tab. 39.

Tab. 39 Redukovaná tabulka přechodů a výstupů automatu pracujícího jako čítač modulo 3

Stav Q^t	Vstup N	Budoucí stav Q^{t+1}	Výstup Y
S0	0	S0	1
	1	S1	
S1	0	S1	0
	1	S2	
S2	0	S2	0
	1	S0	

Přiřazení stavů automatu vnitřním proměnným navrhovaného obvodu

Jednotlivé stavy automatu jsou representovány stavy klopných obvodů. Obecně platí, že je třeba použít tolik klopných obvodů, kolik jich je třeba k zakódování stavů vhodným kódem. Používá se binární kód, Grayův kód, kód 1 z n a další. Vhodnou volbou kódu je možné redukovat hardwarové nároky na realizace kombinačních funkcí λ a δ.

Ve sledovaném příkladu je možné využít 2 klopných obvodů a jako kód zvolit binární kód. Výsledné přiřazení hodnot klopných obvodů stavům je v Tab. 40 Tabulka přechodů a výstupů obsahující informaci o zakódovaných vnitřních stavech klopných obvodů je v Tab. 41.

Tab. 40 Přiřazení hodnot klopných obvodů vnitřním stavům automatu

Stav Q^t	Vnitřní proměnné	
	A	B
S0	0	0
S1	0	1
S2	1	0

Tab. 41 Tabulka přechodů a výstupů automatu s přiřazenými stavy

Q^t	A	B	N	Q^{t+1}	A	B	Y
S0	0	0	0	S0	0	0	1
			1	S1	0	1	
S1	0	1	0	S2	0	1	0
			1	S2	1	0	
S2	1	0	0	S2	1	0	0
			1	S0	0	0	

Sestavení budící tabulky pro zvolené typy klopných obvodů

Podle Obr. 73 výstup kombinační funkce δ logicky napájí vstupy klopných obvodů. Požadovaný stav klopných obvodů z Tab. 41 docílíme pouze tehdy, když na vstupy přivedeme takové hodnoty, které zaručí v následujícím taktu tento stav. V **Tab. 42** jsou uvedeny pro všechna možná překlopení různých klopných obvodů hodnoty vstupů, které je třeba nastavit.

Tab. 42 Budící tabulka klopných obvodů

Současný stav Q^t	Následující stav Q^{t+1}	D klopný obvod D	JK klopný obvod J	JK klopný obvod K	T klopný obvod T
0	0	0	0	X	0
0	1	1	1	X	1
1	0	0	X	1	1
1	1	1	X	0	0

Volba typu klopného obvodu je určena často součástkami, které má návrhář k dispozici. Obecně je možné říct, že realizace s JK klopnými obvody bývá náročnější na počet kombinačních funkcí, které je třeba vypočítat. Realizace s D klopnými obvody je výhodná spíše pro automaty, které klasifikují nějakou vstupní posloupnost, a realizace s T, popřípadě JK obvody je vhodná tehdy, když automat čítá nějaké vstupní události.

V našem příkladu ukážeme výsledné logické výrazy funkce δ pro všechny 3 případy realizace automatu: s klopnými obvody typu D, T a JK. Tab. 43, Tab. 44 a Tab. 45 doplňují předchozí tabulky o sloupce s budícími vstupy těchto klopných obvodů. Sloupce DA a DB odpovídají realizaci pomocí D klopných obvodů, sloupce JA, KA, JB a KB odpovídají obvodům JK a sloupce TA a TB obvodům T.

Tab. 43 Tabulka přechodů a výstupů s hodnotami buzení klopných obvodů typu D

Stav			Vstup	Budoucí stav			Buzení KO		Výstup
Q_t	A	B	N	Q_{t+1}	A	B	DA	DB	Y
S0	0	0	0	S0	0	0	0	0	1
			1	S1	0	1	0	1	
S1	0	1	0	S1	0	1	0	1	0
			1	S2	1	0	1	0	
S2	1	0	0	S2	1	0	1	0	0
			1	S0	0	0	0	0	

Tab. 44 Tabulka přechodů a výstupů s hodnotami buzení klopných obvodů typu JK

Stav			Vstup	Budoucí stav			Buzení KO				Výstup
Q_t	A	B	N	Q_{t+1}	A	B	JA	KA	JB	KB	Y
S0	0	0	0	S0	0	0	0	X	0	X	1
			1	S1	0	1	0	X	1	X	
S1	0	1	0	S1	0	1	0	X	X	0	0
			1	S2	1	0	1	X	X	1	
S2	1	0	0	S2	1	0	X	0	0	X	0
			1	S0	0	0	X	1	0	X	

Tab. 45 Tabulka přechodů a výstupů s hodnotami buzení klopných obvodů typu T

Stav			Vstup	Budoucí stav			Buzení KO		Výstup
Q_t	A	B	N	Q_{t+1}	A	B	TA	TB	Y
S0	0	0	0	S0	0	0	0	0	1
			1	S1	0	1	0	1	
S1	0	1	0	S1	0	1	0	0	0
			1	S2	1	0	1	1	
S2	1	0	0	S2	1	0	0	0	0
			1	S0	0	0	0	0	

Pro jednotlivé zvolené typy klopných obvodů tyto sloupce odpovídají funkčním hodnotám příslušných kombinačních obvodů funkce δ, přičemž odpovídajícími vstupními hodnotami jsou stavy klopných obvodů a hodnota vstupu v příslušném řádku. Popisy těchto kombinačních funkcí můžeme vyextrahovat do Karnaughových map nebo můžeme rovnou sestavit minimální logické výrazy, které representují. Příslušné logické výrazy jsou:

$$D_A = \bar{A}BN + A\bar{B}\bar{N}$$

$$D_B = \bar{A}\bar{B}N + \bar{A}B\bar{N}$$

$$J_A = \bar{A}BN$$

$K_A = N$

$J_B = \bar{A}\bar{B}N$

$K_B = N$

$T_A = \bar{A}BN + A\bar{B}N$

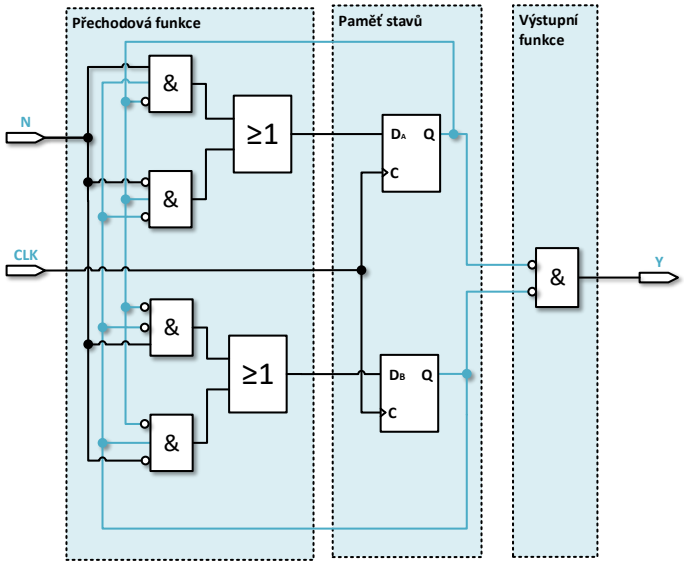
$T_B = \bar{A}\bar{B}N + \bar{A}BN$

$Y = \bar{A}\bar{B}$

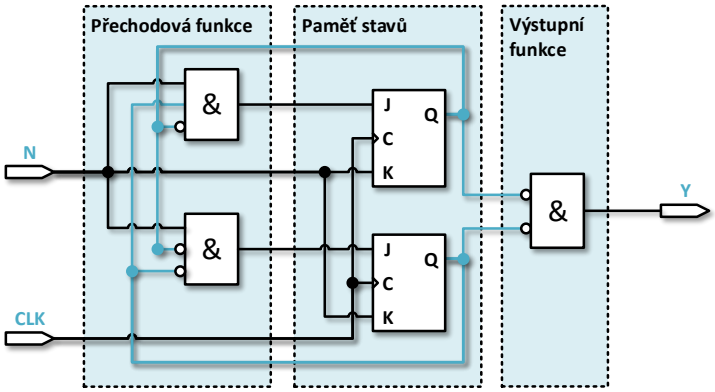
Tyto logické výrazy byly přímo vyčteny z Tab. 43 až Tab. 45.

Návrh propojení obvodu

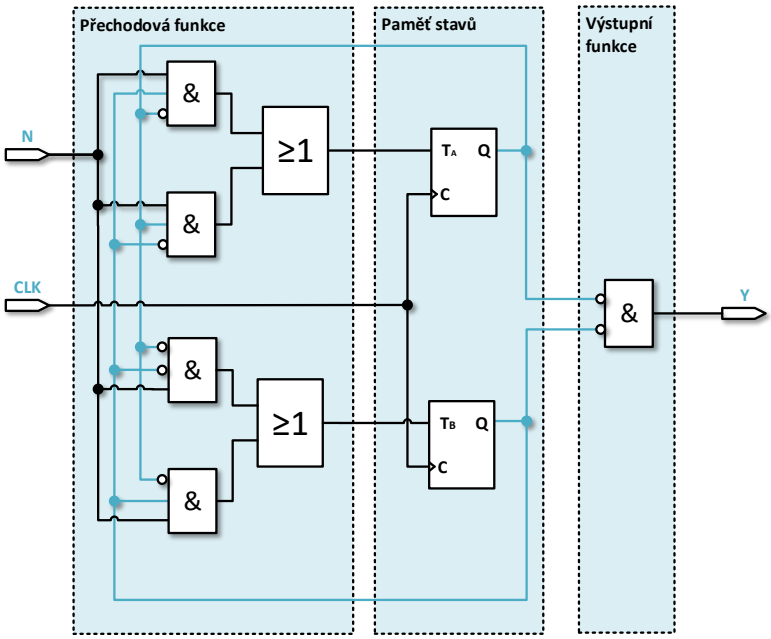
Realizace automatu pomocí obvodů JK, D a T jsou vyobrazeny na Obr. 78, Obr. 79 a Obr. 80. Pro zapojení jsme využili logických výrazů získaných z tabulek.



Obr. 78 Schéma zapojení automatu s využitím D klopných obvodů



Obr. 79 Schéma zapojení automatu s využitím JK klopných obvodů



Obr. 80 Schéma zapojení automatu s využitím T klopných obvodů

9.4.1 Možné problémy realizace automatů

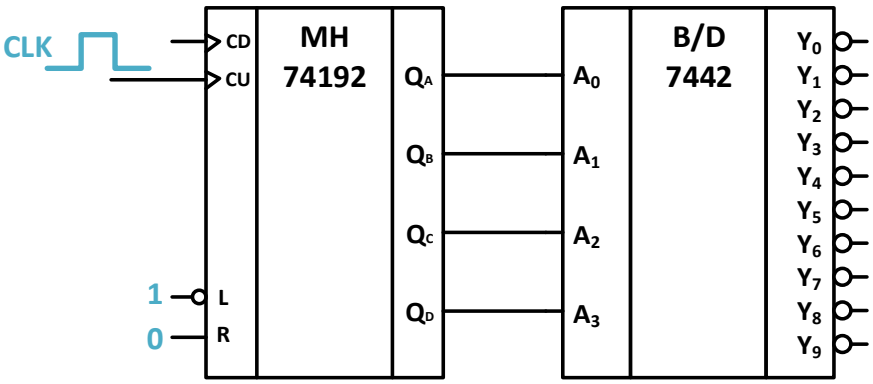
V ilustračním příkladu jsme minimalizovali počet stavů na 3. Vzhledem k tomu, že tento počet stavů lze zakódovat 2 klopnými obvody zůstává otázka, co způsobí 4. možný stav, který není uvažován pro funkci automatu. Do tohoto stavu může automat přejít například v okamžiku zapnutí napájecího napětí. Jestliže není přechodovou funkcí δ definován deterministický přechod z tohoto stavu do nějakého vnitřního stavu automatu, mohlo by se stát, že se automat zablokuje anebo bude generovat neplánovanou sekvenci výstupů. To lze řešit dvěma způsoby:

- vybavit automat asynchronním resetem a zajistit tak vždy při jeho aktivaci uvedení do předem definovaného stavu
- doplnit graf přechodu automatu o všechny další možné stavy a počítat tak s možností definovaného přechodu automatu z každého nevyužitého stavu.

Druhou možnost bychom v ilustračním příkladu využili tehdy, kdybychom neprováděli vyloučení ekvivalentního stavu „počáteční“ a definovali přechody z něj např. podle grafu na Obr. 76.

V dalším textu využijeme pro syntézu obvodu čítače s možností zkrácení cyklu, popř. s možností rozšíření rozsahu, ke kterému doplníme kombinační obvody tak, aby bylo realizováno požadované chování.

Výstupem čítače je stav vyjádřený hodnotou výstupních vodičů. Tento stav můžeme využít jako adresu, jejíž obsah se cyklicky mění, každý následující hodinový impuls zvýší (sníží) dekadickou hodnotu adresy o jednotku, popřípadě nastaví počáteční hodnotu cyklu. Je-li hodinový signál získáván z generátoru, obsah adresy se mění pravidelně, taktování čítače je však také možno provádět buď manuálně, nebo pomocí dvojkového výstupu nějakého čidla, které měří libovolnou fyzikální veličinu.



Obr. 81 Spojení čítače a dekodéru

Adresa může být zpracována v kombinačních obvodech tak, že každému stavu bude odpovídat nějaká kombinace výstupních proměnných. Tímto způsobem vytvoříme automat Mooreova typu [2], viz Obr. 81.

Př. 42 Rozsvěcení LED

Rozsvěcujte postupně 8 LED diod tak, aby se po rozsvícení a zhasnutí jedné diody rozsvítila další, atd. Rozsvěcení cyklicky opakujte.

Pro řešení předpokládejme použití čítače 74192, který cyklicky generuje 10 za sebou jdoucích adres. Výstup čítače je vhodné propojit s dekodérem 7442 (binárně desítkový dekodér). Každý následující hodinový impuls přenese log 0 na následující výstup dekodéru. Připojíme-li na výstupy dekodéru LED diody, jejichž druhý vývod připojíme přes rezistory na napětí +5V, pak se po zhasnutí předchozí rozsvítí následující dioda, tj. pohybuje se světelný bod. Rychlost pohybu se nastavuje změnou frekvence hodinového signálu. Jelikož LED diod je pouze 8 a za sebou generujeme vždy 10 adres, dochází k časové prodlevě na konci řady. V tomto případě je vhodné volit např. čítání vzad v neúplném cyklu od 8 do 1 a nepoužít výstupu dekodéru číslo 0.

Př. 43 Semafor

Sestavte obvod ovládající semafor pro auta a semafor pro chodce na řízeném přechodu pro chodce.

Postup: Nejprve připravíme pravdivostní tabulku kombinačních funkcí. Výstupní proměnné určíme následovně: Č – červená pro auta, Z – zelená pro auta, O – oranžová pro auta, ČCH – červená pro chodce, ZCH – zelená pro chodce. Vstupními proměnnými kombinačních funkcí budou tři výstupy šestnáctkového čítače: a, b c. Pravdivostní tabulka úlohy je v Tab. 46.

Tab. 46 Řízení světelné křižovatky

abc	Č	O	Z	ČCH	ZCH
000	1	0	0	0	1
001	1	0	0	0	1
010	1	0	0	0	1
011	1	1	0	1	0
100	0	0	1	1	0
101	0	0	1	1	0
110	0	0	1	1	0
111	0	1	0	1	0

V pravdivostní tabulce jsme zohlednili požadavek na nestejnou dobu nastavení jednotlivých barev na semaforu. Stavy semaforu, kdy je zelená pro auta a zelená pro chodce se třikrát za sebou v pravdivostní tabulce opakují. Jestliže použijeme k vytvoření signálů a, b, c čítač buzený hodinovým signálem, budou tyto stavy trvat třikrát déle než přechodové stavy, kdy svítí oranžová barva. Pomocí Karnaughovy mapy a De Morganových vztahů určíme logické výrazy pro jednotlivé výstupní proměnné, při úpravě s výhodou využíváme již určených výstupních proměnných (skupinová minimalizace):

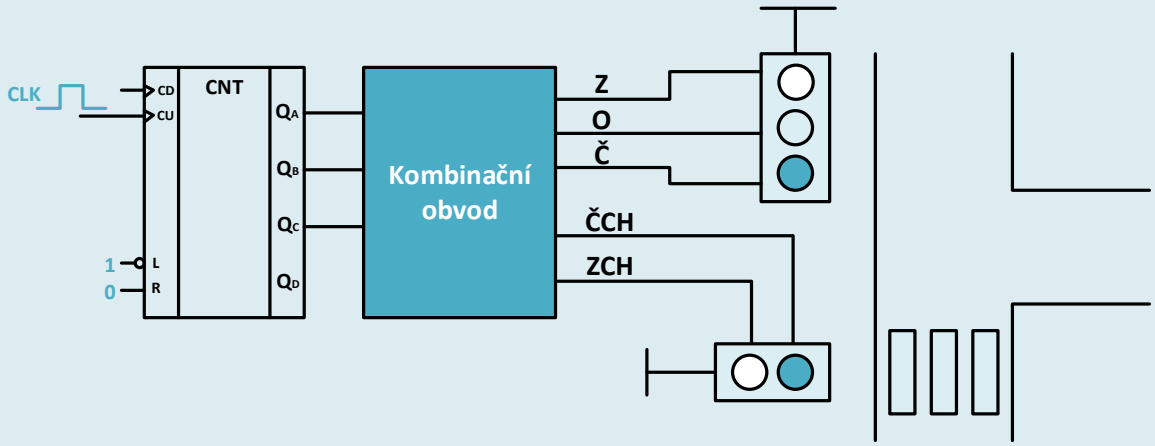
$\check{C} = a$

$O = ab$

$Z = \bar{a}b + \bar{b}c$

$\check{C}CH = c + ab = O + c$

$ZCH = \overline{\check{C}CH}$



Obr. 82 Zapojení obvodu řízení semaforů

Kombinační obvod, který sestavíme pomocí logických hradel, propojíme s výstupy čítače a se vstupy semaforů podle Obr. 82. Úlohu můžeme řešit také spojením čítače s dekodérem a přídavnými hradly.

9.4.2 Příklady k procvičování

Př. 44 Čítač impulsů

Pomocí čítače určete střední počet impulsů vznikajících při sepnutí a rozepnutí neošetřeného tlačítka a přepínače.

Postup:
1. Neošetřené tlačítko připojíme na hodinový vstup prvního stupně dvoustupňového desítkového čítače v režimu čítání vpřed. Výstup čítače připojíme na dvoumístný sedmisegmentový displej. Vstup R přivedeme též na tlačítko a tím zajistíme možnost nulování výstupu.
2. Ke každému stlačeni tlačítka (přepnutí přepínače) odečteme výstup čítače a pak jej vynulujeme. Určíme střední hodnotu a směrodatnou odchylku z 10 až 20 pokusů.

Př. 45 Blikání diodou

Sestavte sekvenční obvod, který zajistí předepsané cyklické zhasínání a rozsvěcení světelné diody. Obvod využívá multiplexoru.

Postup: Zapojíme čítač s multiplexorem tak, aby se umožnilo cyklické nastavování všech adres multiplexoru. Obsah informačních vstupů multiplexoru nastavujeme na přepínačích tak, že lze cyklus rozsvěcení diod měnit za provozu.

Př. 46 Porovnání slov

Navrhněte synchronní sekvenční obvod se vstupem x a s výstupem z.

Obvod porovnává bity vstupujícího slova. Výstup z = 1 pouze tehdy, když se na vstupu objeví posloupnost 111. Výstup z se navrátí do nuly zároveň s první 0 na vstupu x.

Př. 47 Generování posloupnosti

Navrhněte synchronní sekvenční obvod s jedním vstupem x a s jedním výstupem z.

Obvod po přivedení hodnoty log. 1 na vstup x vygeneruje na výstupu z posloupnost 1010 nezávisle na dalších hodnotách přivedených na x.

Př. 48 Porovnání slov 2

Navrhněte synchronní sekvenční obvod s 5 vstupy x1, x2, x3, x4 a x5 a s jedním výstupem z.

Obvod porovnává vstupující 5 bitová slova. V počátečním stavu je z = 0. Jestliže se objeví na vstupech kombinace 10101 automat začne generovat na výstupu z nekonečnou sekvenci 101010.

Př. 49 Porovnání slov 3

Navrhněte synchronní sekvenční automat typu Mealy se vstupem x a s výstupy y, z. Obvod porovnává bity vstupujícího slova.

Výstup z = 1 pouze tehdy, když se na vstupu objeví posloupnost 111 , výstup y =1 pro vstupní posloupnost 110.

Př. 50 Čítač jedniček

Navrhněte synchronní sekvenční obvod se vstupem x a s výstupem z. Obvod sčítá jedničky ve vstupující posloupnosti.

Od okamžiku, kdy je počet jedniček větší nebo roven 4 nastaví obvod výstup z = 1.

Př. 51 Porovnání slov 4

Navrhněte synchronní sekvenční obvod se dvěma vstupy x1, x2 a jedním výstupem z. Obvod porovnává dvě sériově vstupující slova.

Výstup z = 1 pouze tehdy, když se na obou vstupech zároveň objeví shodné hodnoty (x1 = x2) po dobu alespoň dva hodinové takty. Výstup z se navrátí do nuly zároveň s první rozdílnou dvojicí bitů na vstupech.

Př. 52 Porovnání slov 5

Navrhněte synchronní sekvenční obvod se třemi vstupy x1, x2, x3 a s jedním výstupem z. Obvod porovnává sériově vstupující 3 bitová slova.

Výstup z = 1 pouze tehdy, když se na všech vstupech zároveň objeví shodné bity (x1=x2=x3) po dobu alespoň 2 hodinových pulsů . Výstup z se navrátí do nuly v okamžiku, kdy podmínka shodných bitů přestane platit.

Př. 53 Čítání jedniček 2

Navrhněte synchronní sekvenční obvod se vstupem x a s výstupem z.

Obvod sčítá jedničky ve vstupující posloupnosti. Od okamžiku, kdy je počet jedniček větší nebo roven 3 nastaví obvod výstup z = 1.

Př. 54 Generování posloupnosti 2

Navrhněte synchronní sekvenční obvod se vstupem x a s výstupem z.

Obvod po přivedení hodnoty log. 1 na vstup x vygeneruje na výstupu z posloupnost 1010 nezávisle na dalších hodnotách přivedených na x.

Př. 55 Porovnání posloupnosti

Navrhněte synchronní sekvenční automat typu Moore se vstupem x a s výstupy y, z.

Obvod porovnává bity vstupujícího slova. Výstup z = 1 pouze tehdy, když se na vstupu objeví posloupnost 11 , výstup y =1 pro vstupní posloupnost 10.

10 PAMĚTI

Za paměť lze označit libovolné zařízení, které umožňuje uložení, uchování a znovunačtení dat. Paměti realizované s využitím křemíku (bez ohledu na technologii) nazýváme polovodičové paměti. Tyto paměti jsou dnes nedílnou součástí většiny elektronických zařízení. Nejjednodušší paměťové prvky jsou klopné obvody (viz kapitola Klopné obvody), které se používají k uchování jednobitové informace. Vedle procesoru je paměť součástí každého počítače. Informace je do paměti ukládána a z paměti vybírána po malých částech rozsahu několika bitů. Oblast paměti, která obsahuje tuto základní jednotku informace, nazveme buňkou. Každé buňce paměti je přiřazena určitá adresa a zavedením adresy na adresové vstupy paměti můžeme pracovat s obsahem příslušné buňky. Polovodičové paměti se rozdělují podle několika hledisek:

Podle závislosti na napájecím napětí

- Volatilní – po odpojení napájecího napětí ztrácí svůj původní obsah
- Nonvolatilní – jsou nezávislé na napájecím napětí

Podle technologie

- Bipolární – rychlé, vyšší příkon, nižší hustota integrace
- Unipolární – vyšší hustota integrace, v současnosti nejčastěji používané

Podle možností zápisu a čtení dat

- Paměti pouze pro čtení
- ROM (Read Only Memory) – obsah paměti je určen při výrobě, tento obsah není možné měnit
- PROM (Programmable ROM) – obsah paměti může být jednou naprogramován uživatelem
- Paměti převážně pro čtení
- EPROM (Erasable PROM) – elektricky programovatelná paměť s možností opakovaného programování (umožňuje pouze několik set cyklů). Vymazání paměti je prováděno UV záření přes okénko v pouzdru (vyšší cena pouzdra)
- EEPROM (Electrically Erasable PROM) – elektricky programovatelná a smazatelná paměť. Tento typ paměti umožňuje za provozu programovat a mazat některé buňky. Přepisování dat v paměti je prováděno ve speciálním režimu se zvýšeným napětím. Nástupcem pamětí EEPROM jsou rychlé paměti FLASH umožňující velmi rychlý přepis dat v normálním režimu.
- Paměti pro zápis a čtení
- RWM (Read Write Memory) – paměti umožňující libovolné množství čtení a zápisů (v libovolnou dobu stejnou rychlostí) za běžného provozu. Většinou se jedná o volatilní paměti, takže po odpojení napájení ztrácí svůj obsah. Po znovuzapojení obsahují náhodná data.

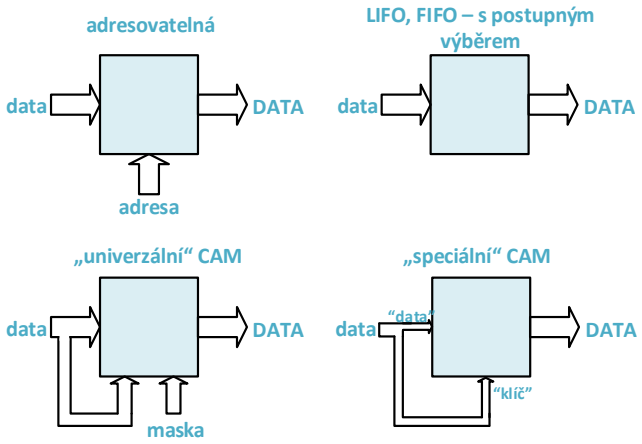
Podle přístupu k paměti (**Obr. 83**)

- RAM (Random Access Memory) – paměť s libovolným přístupem
- SAM (Seriál Access Memory) – přístupové adresy jsou generovány sekvenčně (např. vyrovnávací paměti grafických karet)
- CAM (Content Adressable Memory) – výběr adresy podle části uložené informace

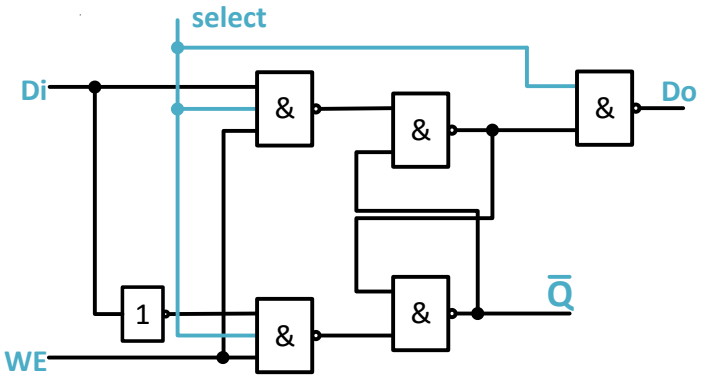
- LIFO (Last In First Out) – zásobník
- FIFO (First In First Out) – fronta

Podle způsobu uchování informace

- SRAM (Statické RWM – termín RWM se příliš nevžil proto název SRAM respektive DRAM) – paměťové buňky jsou založeny na jednostupňovém D klopném obvodu. Statická paměť má podstatně vyšší spotřebu při srovnatelné frekvenci zápisu dat (oproti dynamické paměti).
- DRAM (Dynamické RWM) – u těchto pamětí je informace uchovávána v podobě náboje u řídicí elektrody tranzistoru MOS. Velikost kapacity je v řádu desetin pF (s časem zaniká) informaci v paměti je tedy třeba periodicky obnovovat. Dynamická paměť má delší dobu přístupu k datům, snáze se však integruje (oproti statické paměti). Schéma statické paměťové buňky sestavené z hradel NAND můžeme vidět na Obr. 84.

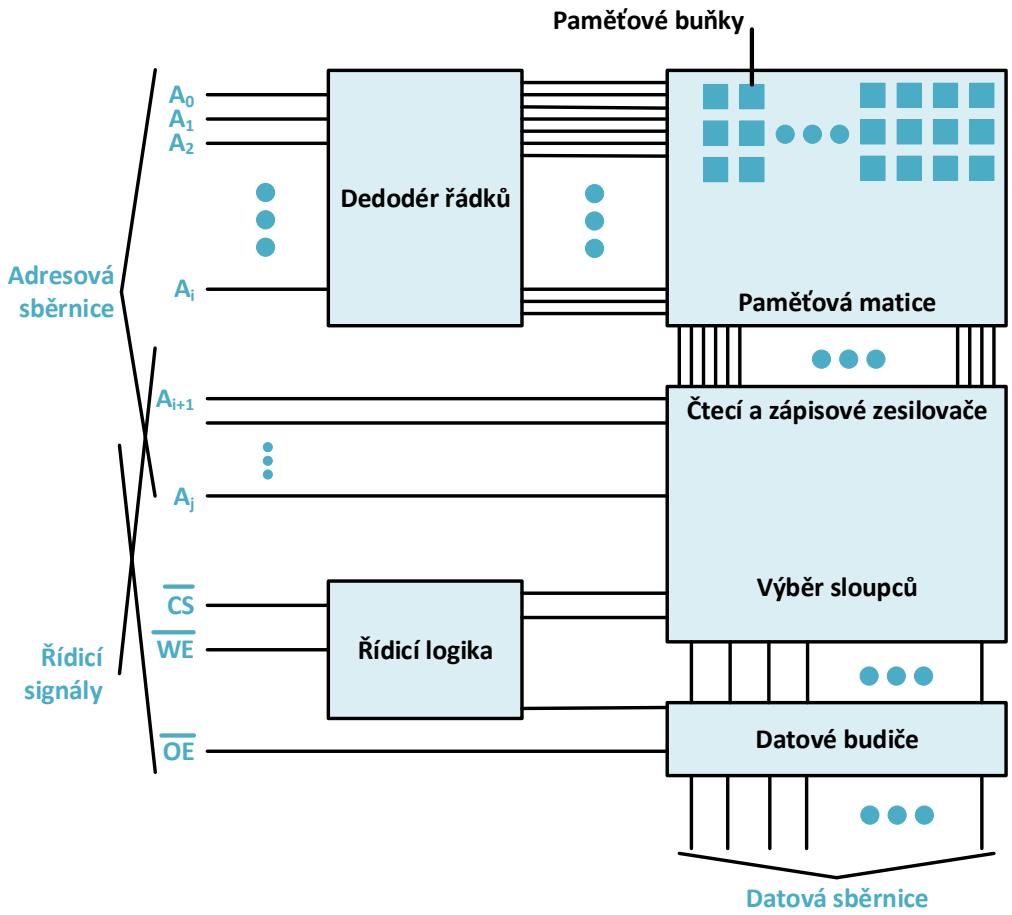


Obr. 83 Blokové naznačení některých přístupů k paměti



Obr. 84 Statická paměťová buňka

Organizaci paměti rozumíme počet bitů, který je k dispozici na dané adrese. Obsah paměti přiřazený adrese nazýváme slovo. Je-li N počet adresovatelných míst a n počet bitů ve slově, pak kapacita paměti je N slov a N.n bitů. Slova bývají jedno, osmi, šestnácti nebo 32bitová. Ač polovodičové paměti jsou značně různorodé jejich vnitřní strukturu lze obecně popsat (viz Obr. 85).



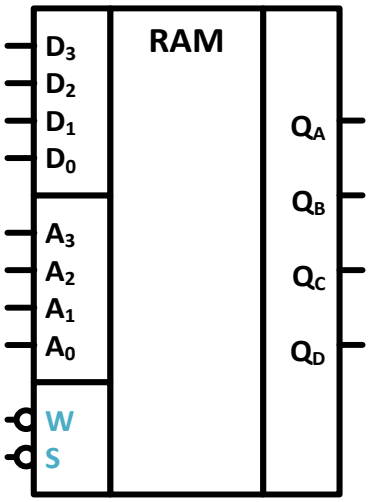
Obr. 85 Schéma uspořádání paměti

Podstatnou částí paměti je tzv. paměťová matice, která je tvořena paměťovými buňkami. Paměťové buňky jsou uspořádány do sloupců a řádků. V každé paměťové buňce je uložena informace o velikosti 1 bit. Vstupem paměti je adresa, výstupem jsou pak data na této adrese obsažená.

10.1 RAM 7489

V dalším textu si stručně popíšeme paměťový obvod RAM 7489, který je využíván na cvičeních z některých předmětů.

Paměť RAM 7489 polovodičová paměť s kapacitou 64 bitů. Tato paměť umožňuje adresovat čtyřbitová slova na 16 čtyřbitových adresách. Schématická značka paměti je na Obr. 86. Na vstupy A0, A1, A2, A3 se přivádí čtyřbitová adresa, na vstupy D0, D1, D2, D3 čtyřbitová data.



S	W	Akce
0	0	zápis dat
0	1	čtení dat
1	0	přímý přenos dat
1	1	zablokování paměti

Obr. 86 Paměť RAM 7489 a význam signálů S a W

Na výstupech Q0, Q1, Q2, Q3 je obsah adresovaného místa v negovaném tvaru. Význam řídicích vstupů S a W je v pravdivostní tabulce na Obr. mm. Vstup S (select) umožňuje blokování paměti. Tento vstup je aktivní v log.0. Aby bylo možné s pamětí pracovat, je nutné na vstup S přivést log. 0.

Z pravdivostní tabulky na vyplívá, že paměť může pracovat ve 4 režimech:

- Zápis – data ustálená na datových vstupech D se zapíši do paměti na adresu nastavenou na adresových vstupech A. Je nastaveno S = 0, W = 0.
- Čtení – obsah paměti na adrese nastavené na adresových vstupech A se v negovaném tvaru přenese na výstup Q.Obsah datových vstupů je ignorován. Je nastaveno S = 0, W = 1.
- Přenos – obsah datových vstupů D se po inversi přenese na výstup 4. Obsah adresových vstupů je ignorován. Je nastaveno S = 1, W = 0.
- Blokování – na výstupu Q jsou log.1, obsah datových i adresových vstupů je ignorován. Je nastaveno S= 1, W= 1.

Při zápisu dat do paměti se postupuje takto:

- Na adresových vstupech se nastaví adresa, AO je nejnižší řád adresy.
- Na datových vstupech se nastaví vkládaná binární informace, DO je nejnižší řád dat.
- Nastavíme S = 0, W = 0.
- Na výstupu se objeví zavedená informace v negovaném tvaru.

10.2 Použití pamětí RAM

Paměti RAM nebo ROM lze mj. využit ke generování logických funkci zadaných pravdivostní tabulkou. Nezávislé proměnné tvoří adresu, přiřazené funkční hodnoty v předepsaném pořadí data zapisovaná na tuto adresu. Pokud počet nezávislých logických proměnných není větší než 4 a požaduji se maximálně 4 logické funkce, lze použít paměti RAM 7489. Paměť PROM 74188

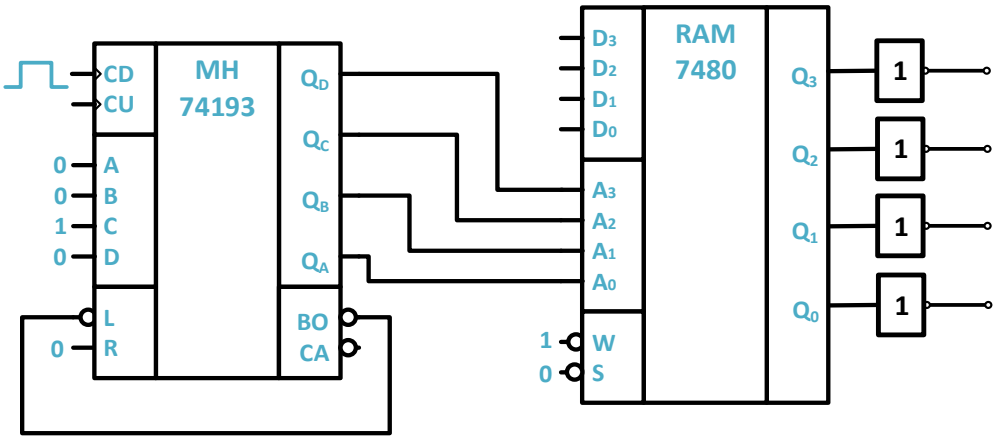
umožňuje zpracovat až 8 funkci s 5 logickými proměnnými. Realizace logické funkce paměti RAM sestává ze dvou kroků:

- zápis pravdivostní tabulky do paměti – programování paměti
- zobrazeni funkčních hodnot.

Pravdivostní tabulku zapisujeme do paměti RAM řádek po řádku. Na adresový vstup obvodu 7489 přivedeme nezávisle proměnné. Při tom je nutno respektovat, že nejnižší řád je na vstupu A0. Na datovém vstupu nastavíme příslušné funkční hodnoty a režim paměti krátce změníme na zápis. Při zobrazeni funkčních hodnot se na výstup paměti RAM připojují invertory a indikační diody. Na adresovém vstupu se nastaví příslušná kombinace proměnných, paměť udržujeme v režimu čtení a načteme příslušné funkční hodnoty.

Pokud se logická funkce často mění, používá se paměti RAM. To má ten nepříjemný důsledek, že po každém vypnuti zdroje je nutno funkční hodnoty znovu zavést do paměti. Pokud je logický systém odladěn, takže by se logické funkce již neměly měnit, je vhodné je naprogramovat jednou provždy do paměti PROM. Pak je ale již jakákoliv změna logických funkci vyloučena. Přesněji řečeno, lze změnit log.0 na log. 1, obrácený krok není možný. Jinou možností je použít paměť EPROM nebo EEPROM, kterou můžeme opakovaně přeprogramovat.

Spojením čítače a paměti lze realizovat programovatelné periodické řízení. Čítač se přizpůsobuje na adresové vstupy a čítá v úplném nebo neúplném cyklu. Do paměti se na odpovídající adresy naprogramuji předepsané stavy výstupů. Tyto stavy se pak v předepsaném pořadí a periodě cyklicky přenášejí na výstup.



Obr. 87 Spojení paměti a čítače

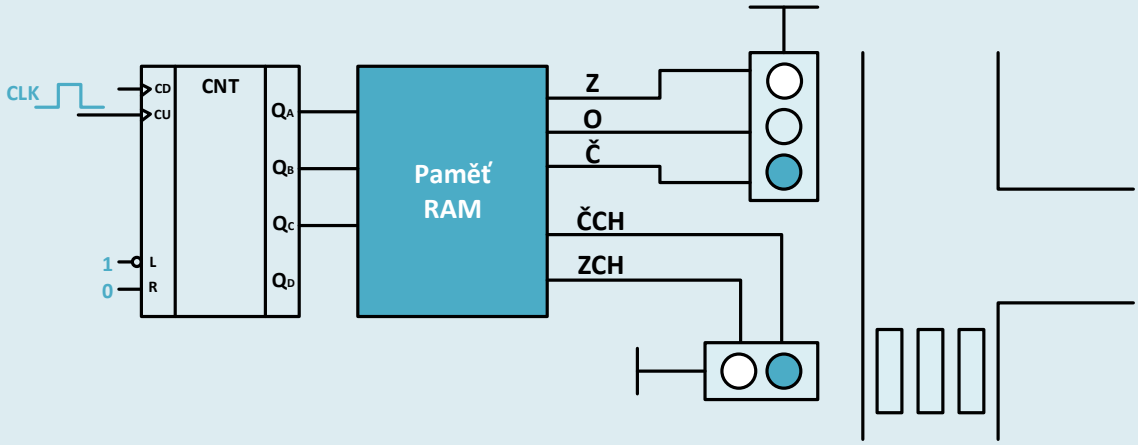
Př. 56 Řízení světelné křižovatky

Sestavte obvod ovládající semafor pro auta a semafor pro chodce na řízeném přechodu pro chodce. (0: Řízení světelné křižovatky) K řešení využijte spojení čítače a paměti.

Tab. 47 Řízení světelné křižovatky

abc	Č	O	Z	ČCH	ZCH
000	1	0	0	0	1
001	1	0	0	0	1
010	1	0	0	0	1
011	1	1	0	1	0
100	0	0	1	1	0
101	0	0	1	1	0
110	0	0	1	1	0
111	0	1	0	1	0

Postup: Využijeme pravdivostní tabulky z. kk. Proměnná ZCH je inversí proměnné ČCH, a proto stačí nastavovat pouze sekvenci logických hodnot pro proměnné Č, Z, O a ČCH, hodnoty proměnné ZCH získáme inversí signálu ČCH. Z tohoto důvodů můžeme využít paměti 7489, která umožňuje pracovat se čtyřbitovými slovy. Čítač s pamětí propojíme analogicky se zapojením uvedeným na Obr. 87 s tím, že není nutné zkracovat cyklus čítače a zároveň stačí využívat pouze tři nižší řády výstupů čítače. V režimu Zápis dat do paměti nahrajeme jednotlivé řádky invertované pravdivostní tabulky. Inversi provádíme z toho důvodu, abychom eliminovali vliv inverse výstupních signálů, která je zabudována ve vlastním paměťovém obvodu. V režimu Čtení dat bude paměť při přivádění adres z čítače postupně generovat sekvenci signálů semaforu. Výstupy obvodu z Obr. 88 se propojí k jednotlivým světlům semaforů.

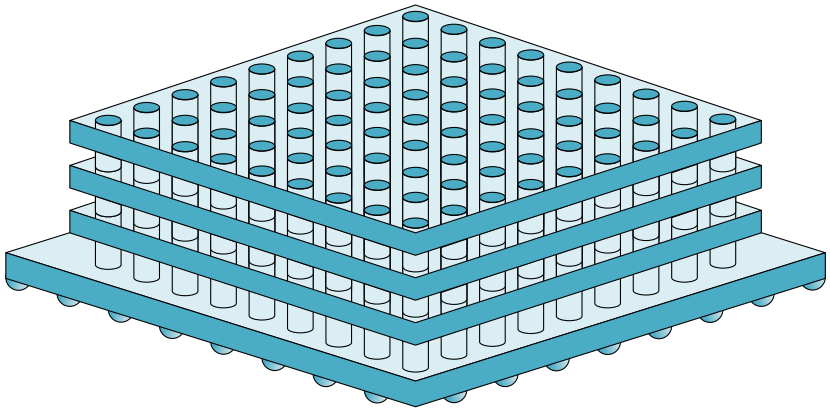


Obr. 88 Řízení semaforů na křižovatce – spojení paměti a čítače

10.2.1 3D Paměti

Zvýšení rychlostí a snížení nákladů při výrobě polovodičových pamětí lze například navrstvením několika paměťových čipů na sebe. Díky vrstvení paměťových čipů na sebe v rámci jednoho pouzdra prudce vzrůstá i paměťová kapacita. Přední světoví výrobci paměťových čipů

pracují na vývoji a výrobě nového typu přepisovatelných paměťových čipů, označovaných jako 3D paměť. Ta má potenciál stát se budoucím nástupcem v současnosti velmi rozšířených flash pamětí typu NAND. Princip 3D paměti můžeme vidět na následujícím obrázku, kde je vyobrazen čip 3D paměti společnosti IBM.



Obr. 89 Koncepční schéma vrstev 3D stacking

Koncept 3D pamětí není nijak nový. Technologie byla vyvinuta již před několika roky jako způsob, jak snížit cenu a prodloužit dobu uchovávání informací až na 100 let, což je více, než umožňují dnes populární paměti typu NAND.

Vzhledem k neustále se zvyšujícím nárokům na vyšší kapacitu a rychlost a nižší spotřebu a velikost u polovodičových pamětí je skládání paměťových vrstev na sebe (neboli tzv. 3D stacking) jednou z možných budoucích cest vývoje paměťových čipů.

11 ZAKÁZKOVÉ OBVODY

11.1 ZÁKLADNÍ ROZDĚLENÍ

Pro realizaci obvodů v předchozích kapitolách jsme využívali virtuálních obvodových prvků, které reprezentovali jednotlivá hradla, klopné obvody nebo vyšší konstrukční celky - čítače, multiplexory, paměti RAM, apod. Pro vytvoření skutečných obvodů můžeme využít tyto prostředky:

- Obvody 74XX jsou z dnešního pohledu jednoduché integrované obvody, které obsahují základní, nebo vyšší konstrukční celky číslicové elektroniky. Typickými představiteli jsou např. obvody 7400 (4 x NAND), 7404 (6 x NOT), 7474 (2 x D klopný obvod) nebo 74138 (dekodér 3-8). S pomocí diskrétních prvků lze vytvářet pouze relativně jednoduché obvody čítající desítky hradel. Jejich výhodou je, že lze funkci číslicového obvodu snadno analyzovat, nevýhodou je snadná zkopírovatelnost a nevhodnost pro velkosériovou výrobu.
- Zakázkové obvody jsou obvody, jejichž návrh plně nebo částečně ovlivňuje zákazník. Po vyrobení obvodu je funkce neměnná. Zakázkové obvody mohou integrovat stovky milionů hradel, mají bezkonkurenčně nejlepší provozní parametry a jsou jen obtížně kopírovatelné.
- Programovatelné obvody jsou podskupinou zakázkových obvodů. Programovatelné obvody je možné nakonfigurovat tak, aby realizovaly různé logické funkce. S pomocí programovatelných obvodů lze vytvářet obvody čítající až desítky milionů hradel. Díky programovatelnosti může uživatel použít jeden programovatelný obvod pro realizaci různých zapojení. Obvody jsou vhodné pro prototypy nebo malé série.

11.2 ZAKÁZKOVÉ OBVODY

Zakázkové obvody, označované též jako obvody ASIC (Application Specific Integrated Circuit) se dělí do několika kategorií, dle zapojení zákazníka do procesu výroby obvodu:

- Zakázkové integrované obvody (Custom Integrated Circuits) navrhuje zákazník kompletně sám, tj. dodává podklady pro výrobu všech masek technologických procesů.
- Při výrobě Polo-zakázkových integrovaných obvodů (Semi-custom Integrated Circuits), se podle zadání uživatele navrhují jen propojovací masky (jedna nebo více); ostatní masky technologického procesu (např. rozvržení hradel na křemíkovém substrátu) jsou univerzální.
- Vnořená pole (Embedded Arrays) jsou kombinací zakázkových a polozakázkových technik. Zákazník dodává masky dílčích částí obvodu (např. procesorů, řadičů pamětí), zbytek obvodu je vyroben analogicky s technikami polozakázkového návrhu.
- Programovatelné integrované obvody (Programmable Devices), jsou obvody které může uživatel sám programovat, např. elektricky přerušováním propojek nebo jiným způsobem (ne však ve smyslu počítačového programu). Uživatel neovlivňuje žádný krok výroby obvodu.

Další dělení zakázkových integrovaných obvodů může být dle stavebních bloků užitých při výrobě obvodu. Zákazník může užívat:

Standardní buňky (Standard Cell) jsou číslicové funkční bloky, kterou jsou na čipu rozmisťovány do řad a sloupců. Zákazník nemůže jejich složení ani funkci ovlivnit, ale může navrhnout jejich vzájemné propojení.

Standardní bloky jsou obdobou standardních buněk v analogové oblasti.

- Parametrizovatelné funkční bloky jsou obdobou standardních buněk, zákazník však může nastavovat parametry pasivních i aktivních prvků těchto bloků.
- Plně zakázkové obvody se vyznačují tím, že si všechny funkční bloky navrhuje zákazník sám. Takto vyrobené obvody dosahují nejlepších provozních parametrů, ovšem za cenu požadavků na značné zkušenosti návrháře, vysokou cenu a dlouhou dobu vývoje.

Výběr způsobu realizace obvodu závisí na mnoha faktorech, z nichž ekonomické aspekty vývoje a provozu hrají nemalou úlohu. Nezpochybnitelná kvalita plně zakázkových obvodů je vykoupena enormně dlouhou dobou vývoje (měsíce až roky), která prodražuje výsledný produkt. Plně zakázkové obvody se tak vyplatí vyrábět pouze ve velkých sériích, kvůli nutnosti rozpustit cenu v počtu vyrobených kusů. Programovatelné obvody jsou oproti tomu vhodné pro malosériovou výrobu, poněvadž je doba návrhu obvodu řádově kratší (týdny až měsíce) a v malých sériích si zákazník může dovolit vyšší cenu za kus. Ekonomické aspekty výběru realizace ilustruje Obr. 90.

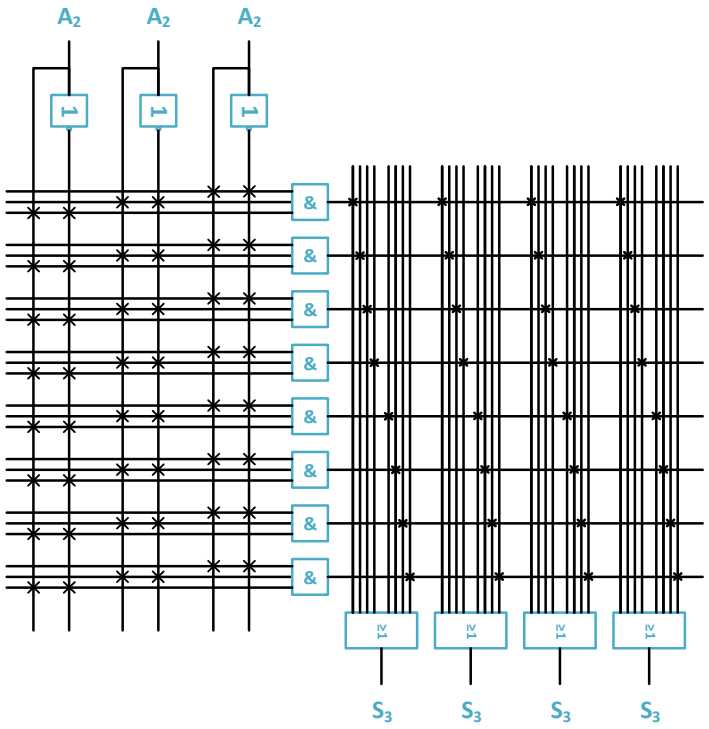
Obr. 90 Ekonomické parametry návrhu IO

Mnohdy se při výrobě výsledného produktu využívá kombinované řešení. Výrobce nejprve dodá malou sérii založenou na programovatelných obvodech, za kterou si nechá zaplatit vysokou částku ospravedlněnou rychlým příchodem výrobku na trh. Zisk z toho plynoucí využije k návrhu plně zákaznického obvodu a sníží si tak výrobní náklady druhé řady produktu.

11.3 PROGRAMOVATELNÉ OBVODY

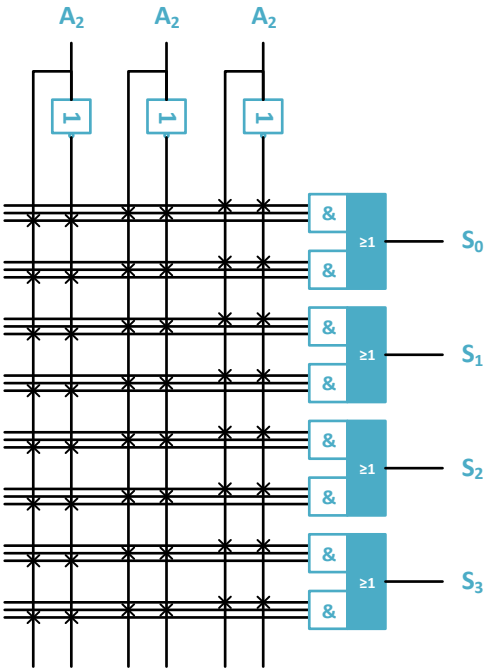
Programovatelnými obvody budeme chápat obvody, u kterých lze přeprogramováním změnit funkci, kterou vykonávají. Rozdělíme je dle historického příchodu na trh na obvody PROM (60 léta 20. století), PLA (70.léta), PAL (konec 70.), CPLD (80. léta - 2000) a FPGA (80. léta – současnost).

- Paměť PROM (Programmable Read Only Memory) s n bitovou paměťovou sběrnicí lze využít jako generátor libovolné logické funkce o n proměnných. S jednou PROM lze realizovat i více kombinačních funkcí, pokud tyto sdílejí proměnné a pokud je datový výstup z PROM větší než jeden bit. Obdobným způsobem lze využít i paměť RAM. Více o pamětech, viz kapitola 10.
- Obvody PLA (Programmable Logic Array) tvoří do matice uspořádaná struktura programovatelých polí AND a OR. Obvody PLA využívají toho, že každou logickou funkci lze zapsat jako součet p-termů (viz 3.1.4). V poli AND jsou vytvářeny uživatelské p-termy, které jsou na výstupní části obvodu sečteny hradly OR. Strukturu PLA ilustruje Obr. 91, křížky je vyznačeno programovatelné propojení.



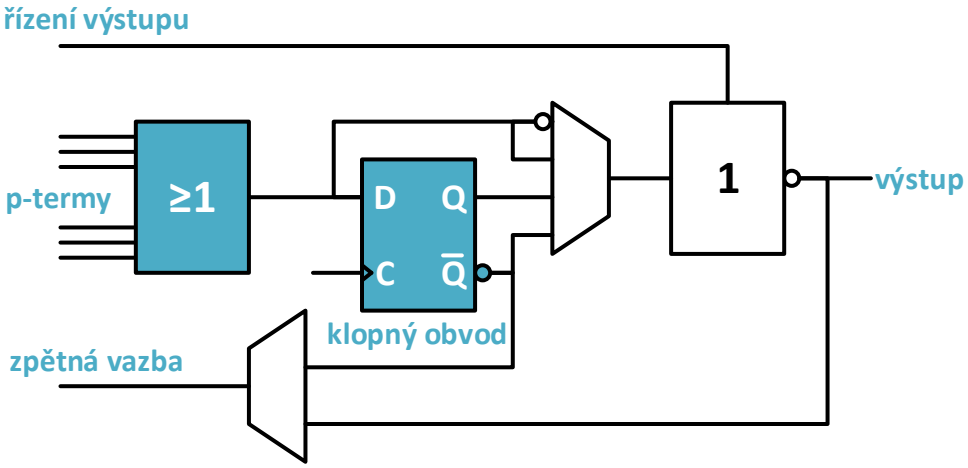
Obr. 91 Principielní struktura PLA

- Obvody PAL (Programmable Array Logic) je programovatelný obvod s pevnou strukturou výstupních hradel OR a programovatelnou strukturou vstupních p-termů. Zatímco u PLA lze pro každé hradlo OR volit které p-termy se sčítají, PAL je omezen jak počtem výstupních hradel OR, tak počtem vstupů do nich (2, 4, 8, 16). Obvody PAL nabídly možnost programování u zákazníka. Principielní struktura PAL obvodu s dvouvstupovými hradly OR je na **Obr. 92**.



Obr. 92 Struktura PAL

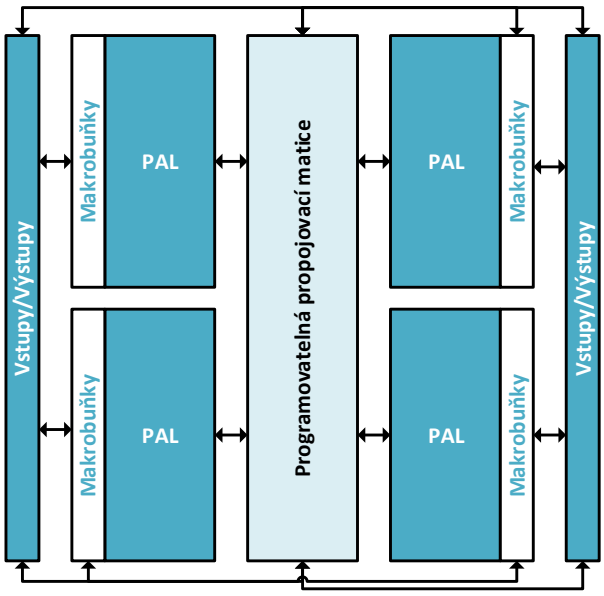
- GAL (Generic Array Logic) je technologicky upravená varianta obvodu PAL. Struktura GAL je obohacena o výstupní makrobuňku, která obsahuje konfigurovatelný klopný obvod se zpětnou vazbou do struktury pole. Struktura GAL tak dovoluje vytvářet stavové automaty bez nutnosti užít externí paměť. Zjednodušené schéma výstupní makrobuňky je na **Obr. 93**.



Obr. 93 Makrobuňka GAL

- CPLD (Complex Programmable Logic Device) ve své struktuře kombinují obvody GAL a částečně i FPGA. CPLD obsahuje více programovatelných polí AND, které jsou mezi sebou,

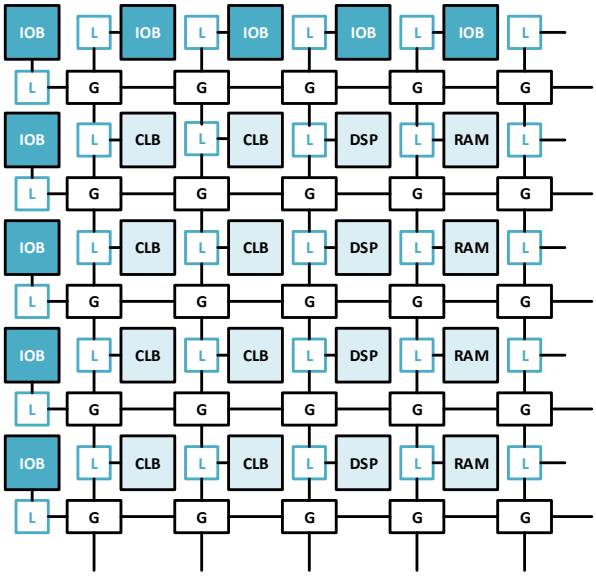
výstupními makrobuňkami a vstupy propojeny reprogramovatelnou propojovací maticí, která umožňuje vytvářet složitější booleovské funkce. Blokové schéma CPLD je na **Obr. 94**.



Obr. 94 Bloková struktura obvodů CPLD

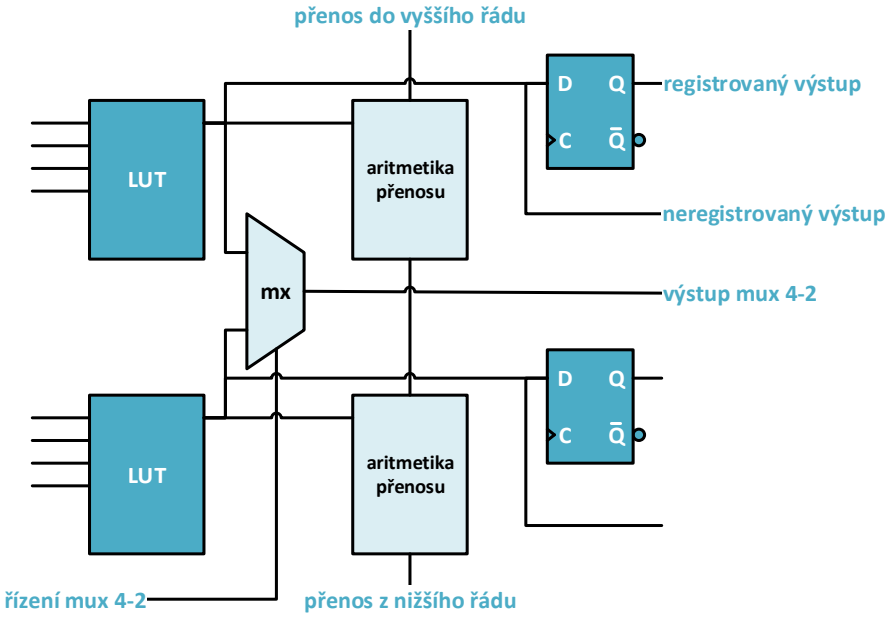
11.3.1 Obvody FPGA

Obvody FPGA, česky označovaná jako hradlová pole, jsou tvořeny do matice uspořádanou soustavou programovatelných bloků, obklopených vstupně výstupními bloky IOB (Input Output Block), které jsou vzájemně propojené propojovací maticí. Struktura obecného FPGA je na Obr. 95.



Obr. 95 Obecné schéma obvodu FPGA

Základní programovatelné bloky CLB (Configurable Logic Block, někdy PLB – Programmable LB) jsou tvořeny vyhledávací tabulkou LUT (Look Up Table), sekvenčním obvodem a dodatečnou logikou. Struktura CLB je na **Obr. 96**, jednotlivé prvky jsou vysvětleny níže.



Obr. 96 Zjednodušené schéma CLB

- LUT si lze představit jako paměť s n bitovou adresou a m výstupy, která umožňuje realizovat libovolnou booleovskou funkci n proměnných. Je-li počet výstupů m větší než 1 je možné jednou LUT realizovat m funkcí, pokud tyto sdílejí vstupní proměnné. V moderních FPGA se vyskytují především 6vstupé LUT se dvěma výstupy, ve výjimečných případech se užívají 4 vstupé LUT s jedním výstupem.
- Sekvenční obvod. Funkci sekvenční logiky plní klopné obvody zapojené na výstup tabulky. Klopné obvody bývají nejčastěji typu D a je možno volit, zda reagují na hladinu nebo náběžnou, či sestupnou hranu hodinového signálu.
- Aritmetická logika. Programovatelné bloky jsou v moderních FPGA spojeny do vyšších celků, které obsahují logiku umožňující efektivně zapojit často realizované funkce, jako sčítačky (obvody arimetiky přenosu) nebo multiplexory (HW multiplexor).

FPGA navíc obsahují obvody, které by při realizaci pomocí LUT příliš vyčerpávaly zdroje FPGA. Mezi takové obvody patří bloky násobiček (na Obr. 95 značeny DSP), pamětí (RAM), řidčeji řadiče externích pamětí, síťí, či sběrnic a někdy i bloky procesorů.

Vstupně výstupní bloky umožňují flexibilní propojení hradlového pole s okolím. IOB je možné konfigurovat jako vstupy, výstupy nebo třístavové budiče a splňují celou řadu napěťových standardů (CMOS, TTL, aj.). Každý IOB ve své struktuře obsahuje klopný obvod, serializér/deserializér (viz kapitola 7.2) a páry IOB je možné konfigurovat jako přijímače/vysílače diferenciálních signálů.

K propojení všech výše zmíněných funkčních bloků slouží propojovací matice. Matice a její řízení zabírá přes 3/4 plochy celého čipu, protože především na propojení závisí schopnost obvodu

realizovat složitější návrhy. Matice se skládá z globálních (na Obr. 95 označeny písmenem G) a lokálních (L) přepínačů. Globální přepínače spravují více vodičů, ale neumožňují propojení typu „vše se vším“. Lokální přepínače jsou flexibilnější, ale obsluhují pouze vodiče vedoucí z a do programovatelných bloků.

Nastavení všech prvků hradlového pole je řízeno z konfigurační paměti zařízení. Nejčastěji se jedná o paměť SRAM, řidčeji o non-volatilní paměť typu FLASH. Největší výhodou FPGA založených na volatilní paměti je možnost je libovolně (tj. i za běhu obvodu) přeprogramovat. Nevýhodou je, že po zapnutí obvodu je nutné konfiguraci obvodu načíst z externí nebo integrované non-volatilní paměti.

Parametry obvodů FPGA jsou udávány nejrůznějšími ukazateli. Kvantitativně je obvod popsán počtem klopných obvodů/LUT, počtem IO bloků, celkovou kapacitu blokových pamětí nebo počtem násobiček. Často se udávají frekvence, kterých jsou dílčí bloky schopny dosahovat a hradlová pole stejných kvantitativních parametrů jsou dělena do kvalitativních košů.

Nemá smysl uvádět parametry současných hradlových polí, protože jsou nové modely uváděny na trh s přibližně ročním cyklem a jakékoliv údaje tak brzy zastarají. Zapálený čtenář si bližší informace může najít na webových stránkách předních výrobců: Xilinx, Altera, Lattice a Microsemi (seřazeno dle podílu na trhu).

11.4 METODIKA NÁVRHU ZAKÁZKOVÝCH OBVODŮ

Základní postup návrhu číslicového obvodu sestává z několika navazujících kroků. Prvním krokem návrhu je vytvoření specifikace obvodu. Specifikace říká, jakou činnost daný obvod provádí, jakého charakteru jsou vstupní a výstupní data, kolik má obvod vstupních a výstupních pinů apod. Se složitostí obvodu roste složitost specifikace a u moderních návrhů typu systém na čipu, které se skládají z procesorů, sběrnic a složitých periferií je nutné vytvářet softwarový model zařízení.

Druhým krokem návrhu je popsání funkce obvodu. K tomu lze využít následující prostředky:

- Schématický vstup – návrhář popisuje strukturu obvodu kreslením zjednodušeného schematu, které se skládá ze základních obvodových prvků. Tento návrhový postup je v dnešní době vhodný jen pro malé, jednoduché obvody. Pro složitější funkční bloky je nepřehledný, a proto byl opuštěn.
- Návrh v HDL (Hardware Desription Langugage). Jazyky návrhu hardware jsou jazyky užívané pro popis hardwarových struktur. Na rozdíl od imperativních jazyků (C, Java, Python) se jedná o jazyky paralelní – popisuje se spíše interakce mezi jednotlivými funkčními bloky než přímo kroky algoritmu. Mezi zástupce HDL patří na syntaxi ADA založené VHDL, na syntaxi C založený Verilog a na objektovém C++ založený SystemVerilog. V současné době se jedná o nejpoužívanější způsob návrhu obvodů.
- Blokový schematický vstup. Na rozdíl od schematického vstupu se pracuje s komplexními obvody – od bloků zpracování signálů (FFT, konvoluční filtry, apod) až po procesory, sběrnice a jiné, vyšší funkční celky. Produktem SW blokového vstupu je často HDL popis obvodu, který sestává z předpřipravených funkčních bloků. Představiteli jsou LabView, System Generator, apod.

- Vysokoúrovňová syntéza HLS (High Level Synthesis). HLS je relativně mladá technologie, která je založená na metodách, které umožňují převést algoritmus popsaný v běžném programovacím jazyce (nejčastěji C/C++) do HDL. Z principu jsou znemožněny dynamické alokace paměti, některé možnosti užívání ukazatelů, nebo tvorba heterogenních datových struktur. Výhodami jsou snadná a rychlá simulace obvodu a rychlá změna jeho parametrů.

Popis systému se předává software, který provádí tzv. syntézu obvodu. Typická syntéza popis obvodu lexikálně analyzuje a na základě popsaných struktur vytvoří orientovaný graf, který se skládá ze základních primitiv (např. u FPGA LUT, D obvody, u ASIC hradla, aj.) a jejich propojení a obecně se nazývá netlist. Mezi základní syntézní nástroje patří proprietární SW výrobců hradlových polí – Xilinx dodává prostředí Vivado, Altera návrhový systém Quartux II. Univerzální syntézy dodávají firmy Synopsys, Cadence a Mentor Graphics jak pro FPGA, tak pro ASIC.

Netlist je dále implementován v procesech mapování, umisťování a propojování. Mapování převádí základní prvky na technologická primitiva daného obvodu. Primitiva jsou umístěna po ploše čipu, vzájemně propojena a z těchto informací jsou vytvořena výstupní data. U ASIC jsou to masky dílčích technologických procesů (rozvržení hradel, metalické propoje různých vrstev), u obvodů FPGA je vytvořen konfigurační soubor zařízení (bitstream), který je nutné speciálním programátorem (např. JTAG) do cílového čipu nahrát.

12 JAZYK VHDL

Jazyk VHDL (VHSIC HDL – Very High Speed Integrated Circuits Harware Description Language) byl počátkem 80. let vytvořen na objednávku ministerstva obrany spojených států amerických s cílem sjednotit dokumentaci k číslicovým obvodům dodávaným v rámci projektů ministerstva zadávaných komerčním subjektům. Původně popisný jazyk se brzy začal používat i pro simulaci a následně vznikly nástroje, které umožnili část konstruktů jazyka použít pro návrh obvodů. Následující text nemá ambici důkladně popsat všechny vlastnosti a konstrukty jazyka. Zaměříme se pouze na základy, ale i s nimi bychom měli být schopni popsat i velmi komplexní obvody. V textu se zaměříme na normovaný standard jazyka IEEE 1076.3 (VHDL 2002), jelikož poslední verze (VHDL 4.0 IEEE 1076-2008) není ve všech vývojových prostředích plně podporována.

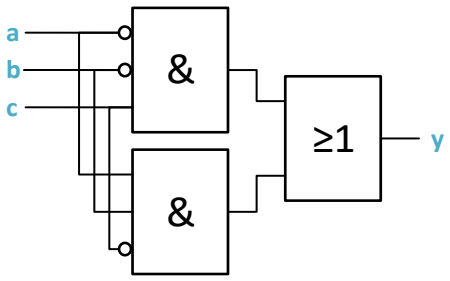
12.1 ZÁKLADNÍ BLOKY

Jazyk VHDL vychází z jazyka ADA, je strukturovaný, hierarchický, paralelní a silně typově orientovaný. Soubory jazyka VHDL mají příponu *.vhd nebo *.vhdl. Ukázkový soubor je na příkladu níže. V jazyce VHDL je implementován kombinační obvod se vstupy a, b, c a výstupem y, který realizuje funkci $y = \bar{a}\bar{b}c + ab\bar{c}$.

Pokud se v této kapitole objeví příklad obecné syntaxe, pak jsou hranatými závorkami označeny nepovinné části kódu. Tučně, nebo barevně jsou zvýrazněna klíčová slova. Objekty (identifikátory) nejsou zvýrazněny nijak.

Př. 57 Příklad obvodu v jazyce VHDL

```
-- Komentář k prvnímu obvodu
library ieee;
use ieee.std_logic_1164.all;

entity example_001 is
	port(
		a, b : in  std_logic;
		c    : in  std_logic;
		y    : out std_logic
	);
end entity example_001;

architecture RTL of example_001 is
	signal term_0 : std_logic;
	signal term_1 : std_logic;
begin

	term_0 <= not a and not b and c;
	term_1 <= a and b and not c;
	y <= term_0 or term_1;

end architecture RTL;
```



Obr. 97 Schéma obvodu z 0

Povšimněte si, že základní struktura obvodu popsaného ve VHDL se skládá z trojice elementů: Deklarace použitých knihoven, entity a architektury obvodu.

12.1.1 KNIHOVNY

Použití knihovny je deklarováno pomocí klíčového slova library, za kterým následuje název mateřské knihovny, typicky adresáře. Z této knihovny se pomocí klíčového slova use určuje hierarchicky nižší knihovna. Pro účely úvodu textu si zatím vystačíme s knihovnou IEEE std_logic_1164, která deklaruje (a tak nám i umožňuje používat) datové typy std_logic a std_logic_vector, které budeme používat především pro vstupy a výstupy obvodu. Použití knihoven je platné pouze pro první **entitu**, před níž se nacházejí…

Př. 58 Ukázka deklarace použití knihovny IEEE a podknihovny IEEE std_logic_1164.

```
library ieee;
use ieee.std_logic_1164.all;
-- Víceřádkové komentáře v C stylu /**/
-- podporuje VHDL až od verze VHDL2008?
```

12.1.2 ENTITA

Entita má ve VHDL význam popisu rozhraní navrhovaného obvodu s vnějším světem. Obsahuje deklarace jmen a typů vstupů a výstupů obvodu a jeho název. V některých případech může obsahovat i parametry obvodu.

Základní deklarace **entity** je zapsána v následujícím příkladu. Obsahuje klíčové slovo **entity** jméno_entity **is** a je zakončena slovy **end entity** [jméno_entity];

Př. 59 Příklad deklarace entity

```
entity example_001 is
	port(
		a, b : in  std_logic;
		c    : in  std_logic;
		y    : out std_logic
	);
end entity example_001;
```

Deklarace portů – vstupů a výstupů obvodu je popsána v části **port** (); Jednotlivé záznamy seznamu portů jsou odděleny středníkem, za posledním záznamem středník nepíšeme. Deklarace portů má syntax:

jméno_portu$_0$ [,jméno_portu$_1$,…, jméno_portu$_n$] : **mód** datový_typ[;]

Každý záznam ze seznamu portů může obsahovat čárkou oddělený seznam jmen portů, pokud tyto jsou stejného módu a datového typu. Možné módy portu uvádí tabulka níže.

Tab. 48 Módy portu

Mód	Typ	Poznámka
in	Vstupní port	-

Mód	Typ	Poznámka
out	Výstupní port	Výstupní porty nelze uvnitř architektury číst
inout	Vstupně – výstupní port	Slouží pouze pro připojení portu k třístavové sběrnici
buffer	Výstupní port	Na rozdíl od portů v módu out lze z buffer uvnitř architektury číst. V praxi se téměř nepoužívá.

Datový typ portu může být jakýkoliv datový typ, který podporuje VHDL (budou popsány dále v textu). V praxi se pro porty používají především datové typy std_logic a std_logic_vector.

Jména portů, stejně jako i ostatních identifikátorů ve VHDL podléhají určitým pravidlům. VHDL povoluje užívat alfanumerické znaky (na češtinu ale při psaní kódu zapomeneme) a podtržítka a nerozlišuje mezi velikostí znaků. Identifikátor nesmí začínat číslicí nebo podtržítkem a podtržítkem nesmí ani končit.

Zápis identifikator_6 a IDENtiFIkatOR_6 je ve VHDL významově úplně shodný. Tato vlastnost, ale neznamená, že velikost znaků nehraje roli v softwarových nástrojích, které se soubory pracují, ba naopak. Většina návrhářských nástrojů je na velikost znaků citlivá, a proto volíme styl kódování tak, abychom v rámci projektů velikost znaků neměnili.

Součástí entity bývá i blok **generic**. V něm v syntax podobné seznamu **port** uvádíme seznam konstant platných v dané entitě, který můžeme využít pro parametrizaci obvodu. Generickým parametrům je věnována samostatná kapitola 12.3.4

12.1.3 ARCHITEKTURA

Zatímco entita popisuje obvod zvnějšku (jako černou skříňku), architektura popisuje vnitřní uspořádání nebo funkci obvodu. Ke každé entitě může existovat více typů architektur – jedna architektura může obsahovat např. popis na funkční úrovni, zatímco jiná ho popisuje na úrovni hradel.

Př. 60 Deklarace a implementace architektury

```
architecture RTL of example_001 is

	signal term_0 : std_logic;
	signal term_1 : std_logic;

begin

	term_0 <= not a and not b and c;
	term_1 <= a and b and not c;
	y <= term_0 or term_1;

end architecture RTL;
```

Architektura je deklarována pomocí zápisu **architecture** jméno_architektury **of** jméno_entity **is** a je ukončena sekvencí **end architecture** [jméno_architektury];

Popis architektury je rozdělen na dvě část – deklarační a implementační část. Deklarační část se nachází před klíčovým slovem **begin** a umožňuje deklarovat signály, konstanty, podprogramy a jiné.

Signály mohou reprezentovat vodiče, registry, kombinační elementy, aj. Jelikož je v mnoha případech význam signálů a portů zaměnitelný budeme v následujícím textu porty označovat jako signály. V uvedeném příkladě jsou deklarovány dva signály term_0 a term_1, které reprezentují výstupy dvojice hradel výrazu ze začátku kapitoly. Deklarace signálu je podobná deklaraci portu (chybí mód) a má syntax:

```
signal jméno_signálu0 [,jméno_signálu1,..., jméno_signálun] : datový_typ [:= inicializace];
```

Obdobně, jako signály je možné deklarovat i konstanty. Na rozdíl od signálu je inicializace povinná. Do konstanty nelze v těle architektury zapisovat.:

```
constant jméno_konstanty : datový_typ := inicializace;
```

V deklarační části architektury můžeme dále deklarovat i podprogramy. S těmi se ale seznámíme až později. Tělo architektury obsahuje popis vnitřního uspořádání nebo chování popisovaného obvodu. V dalších kapitolách si vysvětlíme, jak takový popis vytvořit.

12.2 DATOVÉ TYPY

Jazyk VHDL obsahuje poměrně bohatý, avšak pro potřeby praxe nedostačující výčet implicitních skalárních datových typů. Práce s nimi je navíc poměrně svázaná rigidním systémem typové kontroly. V základu je rozdělujeme na numerické, fyzikální a výčtové datové typy. Numerické dále dělíme na celočíselné a na typy s plovoucí desetinnou čárkou.

Datový typ	Význam		Zápis
integer	celočíselný datové typy	numerický	-10
natural	celočíselný kladný datový typ <0, integer'max>	numerický	10
positive	celočíselný kladný datový typ <1, integer'max>	numerický	10
real	datový typ s plovoucí desetinnou řádovou čárkou	s desetinnou čárkou	10.1
boolean	booleovský datový typ nabývající hodnot true a false	výčtový	true, false
character	znak	výčtový	'a'
bit	booleovský datový typ nabývající hodnot logická 0 a logická 1	výčtový	'0', '1'
time	čas v rozlišení (fs, ps, ns, us, ms, s, m, hr)	fyzikální	10 ns
file	soubor	soubor	soubor

12.2.1 NUMERICKÉ DATOVÉ TYPY

Numerické datové typy označují veličiny, které lze vyjádřit jako nějaké číslo – hodnota čítače, hodnota načtená z AD převodníku, aj.

Základním numerických datovým typem je integer. Dle definice je integer celočíselný datový typ kódovaný dvojkovým doplňkem v minimálním rozsahu –2147483648 až 2147483647. Pozor. Norma specifikuje pouze minimální rozsah – rozhodne-li se tvůrce našeho syntézního nástroje jej implementovat větší, než potřebných 32 bitů, pak normu splnil, ale náš obvod může značně nabobtnat. Navíc není zaručena přenositelnost kódu pod jiné platformy nebo jiné vývojové nástroje. Kladná celá čísla reprezentuje datový typ **natural**, kódovaný stejně jako integer, ale pouze s polovinou rozsahu (kladného). Nenulová kladná čísla v rozsahu <1, 2147483647> reprezentuje datový typ positive.

Chceme-li využít kratší numerické rozsahy než integer, pak musíme deklarovat vlastní datové typy, např.:

```
type short_uint is range 0 to 65535;
type short_int is range -1 to 32767;
```

U prvního (short_uint) můžeme očekávat délku $\log_2$(rozsah) a kódování pomocí binárního váhového kódu. Délka signálů typu short_uint bude 16 bitů. Délka druhého typu (short_int) bude stejná, ale na kódování bude implicitně použit dvojkový doplněk, jelikož jeho rozsah zasahuje do oboru záporných čísel.

Chceme-li použít číslo jiného rozsahu, ale stejného kódování a bitové délky, pak můžeme deklarovat subtyp. V níže uvedeném příkladu si můžeme být jistí, že datový typ my_int bude kódovaný nejméně 32 bity v dvojkovém doplňku, ale typová kontrola nedovolí přiřadit hodnoty mimo rozsah <0, 10>

```
subtype my_int is integer range 0 to 10;
```

Potíž numerických datových typů je v tom, že na první pohled není z popisu entity poznat kolik bitů, registrů nebo pinů ten který signál nakonec potřebuje. A i když zvolíme nějaký způsob zápisu (např. unit16, int5, atp.), stále nebude možné provádět určité operace nebo obvod úplně simulovat. Z výše uvedených důvodů se celočíselné numerické datové typy využívají spíše pro vnitřní proměnné – k indexaci polí, řízení smyček, řízení toku programu, parametrizaci obvodu atp.

Datové typy v plovoucí řádové čárce (**real**) jsou často syntézními nástroji vyhodnoceny jako nesyntetizovatelné a pokud tomu tak není, pak jsou výsledné hardwarové struktury velmi objemné. Při popisu syntetizovatelných obvodů se jim raději vyhneme.

12.2.2 VÝČTOVÉ DATOVÉ TYPY A TYP STD_LOGIC

VHDL implicitně nabízí trojici základních výčtových datových typů. První – character – reprezentuje znak (ve smyslu ASCII znaku) a zbývající dvě – boolean a bit – reprezentují binární hodnoty. Boolean je používán jak pro vyhodnocení řízení toku programu – např. jako výsledek relačních operátorů, ale může reprezentovat i fyzické objekty, např. výstup hradla. Ptáte-li se, k jakému účelu pak slouží datový typ bit, pak odpověď asi spočívá ve snaze usnadnit práci návrhářům. Jistě je výhodnější zapsat např. 255_{10} jako 11111111_2, než jako true, true, true, true, true, true, true, true. Zbývá pouze dodat dvě poslední fakta – boolean a bit nelze vzájemně přiřazovat a ….

…. a při návrhu datových typů byly opomenuty úplně základní vlastnosti reálných obvodů. Pomocí booleovské algebry totiž nelze popsat nedefinované stavy, obvody s vysokou impedancí,

sběrnice typů pull up/down, atp. K návrhu jazyka musel být zákonitě doplněn další datový typ, který výše zmíněné reflektuje. Jedná se o výčtový typ, ale namísto dvou nabývá devíti hodnot:

Tab. 49 Hodnoty std_logic

Hodnota	Název	Význam
'U'	uninitialized	Neinicializovaná hodnota, např. obsah registru po resetu, nezapojený vstup, atd.
'X'	strong drive, unknow logic value	Neznámá hodnota buzená normálním budičem
'0'	strong drive, logic zero	Logická 0 buzená normálním budičem
'1'	strong drive, logic one	Logická 1 buzená normálním budičem
'Z'	high impedance	Obvod s vysokou impedancí v odpojeném stavu
'W'	weak drive, unknown	Neznámá hodnota výstupu budiče s nízkou impedancí (např. pull up/down)
'L'	weak drive, low	Logická 0 na výstupu budiče s nízkou impedancí
'H'	weak drive, high	Logická 1 na výstupu budiče s nízkou impedancí
'-'	don't care	Hodnota na níž nezáleží, viz 3.2.2

Výčtový typ std_logic byl definován normou IEEE 1164 a ve VHDL je jeho použití podmíněno užitím knihovny ieee.std_logic_1164.all, se kterou jsme se seznámili v úvodu. Pro simulaci je naprosto klíčový a slušný vývojář jej používá minimálně pro interakci svého obvodu s vnějším světem. Všimněte si, jednoduchých uvozovek v sloupci hodnota v tabulce Tab. 49, std_logic je deklarovaný jako znakový výčtový datový typ.

Výčtové datové typy symbolizují takové objekty obvodu, jež mohou nabývat pouze definovaných stavů. Toho můžeme využít např. pro popis stavů při popisu stavových automatů. Vlastní výčtový datový typ deklarujeme pomocí zápisu:

```
type enum_t is (Value1, Value2, …, ValueN);
```

V kódu pak pracujeme se symboly, jež jsou explicitně ve výčtu vyjmenovány. Do objektů výčtového datového typu můžeme přiřazovat pouze hodnoty <Value1, ValueN>, stejná pravidla platí pro porovnání a jiné operace. Ve výchozím nastavení jsou výčtové typy kódovány jako binární čísla délky n, kde n je taková mocnina 2, pomocí které je možné zakódovat všechny prvky výčtu.

12.2.3 POLE, ROZSAH, STD_LOGIC_VECTOR

Ve většině číslicový obvodů se můžeme setkat se slučováním sémanticky shodných bitů do celků jako sběrnice, registry atp. Kompozitní datové typy jsou reprezentovány pomocí datových typů pole a záznam. Základními, v jazyce přímo deklarovanými složenými typy jsou bit_vector (v IEEE1164 je jeho obdobou std_logic_vector) a string. Dle interní deklarace se jedná o pole skalárních výčtových tipů bit u typu bit_vector (a std_logic u std_logic_vector) a character u string.

VHDL umožňuje deklarovat i jiná pole. Obecná deklarace pole je následující:

```
type jméno_typu is array (left downto/to right) of data_type;
```

Z příkladu můžeme odhadnout, že se vytvoří nový typ jméno_typu – pole (o nějaké velikosti?) složené z prvků datového typu data_type. Jak velké pole je, definuje parametr rozsah – range, který ve VHDL omezuje rozsah a indexaci pole.

Ve VHDL rozlišujeme dva druhy endianity. Pokud označujeme více významné bity nižším indexem, tj. číslujeme inkrementálně zleva doprava, pak hovoříme o big endian a pro zápis rozsahu používáme klíčové slovo to. Např. n prvkové pole deklarujeme následovně:

```
type vector8bit_t is array (m to m+n-1) of data_type;
```

Chceme-li indexaci obrátit a vyšším indexem označovat prvky více vlevo, pak naše pole deklarujeme:

```
type vector8bit_t is array (m+n-1 downto m) of data_type;
```

Takové pole má rozsah druhu little endian a kvůli konvencím (čísla jsme v našem kulturním okruhu zvyklí psát s nejvyšším řádem vlevo) ho při popisu prvků obvodů užíváme častěji. I v příkladech v tomto textu užívat budeme využívat „malou endianitu“ pro popis „proměnných“ téměř výlučně. VHDL umožňuje v jednom souboru využívat více datových typů a signálů s různou endianitou. Little endian rozsahy bývají využívány pro čísla, big endian rozsahy např. pro popisy pamětí, smyček v kódu atp. Různé příklady indexace naleznete v Tab. 50.

Tab. 50 Příklady rozsahů

Rozsah	Nejvýznamnější	Nejméně významný	Rozsah
7 downto 0	7	0	7 – 0 + 1 = 8
0 to 7	0	7	0 + 7 + 1 = 8
10 downto 3	10	4	10 – 4 + 1 = 7
4 downto -3*	4	-3	4 – (-3) + 1 = 8

*takto jsou často indexovány vektory, jež reprezentují čísla s pevnou řádovou čárkou

Pomocí deklarace typu lze vytvářet i složená dvourozměrná pole (např. paměti):

```
type dim2_t is array (m downto n) of std_logic_vector;
```

Jednotlivé položky polí lze adresovat pomocí v kulatých závorkách () zapsaného indexu, který musí být numerického datového typu a je umístěn za indexovanou proměnnou. Indexace vybere proměnnou základního typu, ze kterého se pole skládá. Chceme-li indexovat bit z položky použijeme dvojici kulatých závorek (slovo, bit).

Vícerozměrná pole dimenze D lze vytvářet zápisem:

```
type multidim_t is array (m0 downto n0, m1 downto n1, ..., mD-1 downto nD-1) of data_type;
```

Indexace vícerozměrných polí je prováděna pomocí čárkou oddělených indexů v pořadí deklarace dimenzí ($Index_0$, $Index_1$, …, $Index_{D-1}$). Index nejvíce vlevo indexuje hierarchicky nejvyšší pole.

Speciálním případem pole je pole, jehož rozsah je neomezený a je deklarován až při vytváření signálu:

```
type unconstrained_t is array (integer range <>) of data_type;
```

```
signal uarr : unconstrained_t (m downto n);
```

Př. 61 : Ukázka práce s poli

```
    type typ_pole1D is array (M downto N) of typ;
    type typ_pole2Da is array (M downto N, M downto N) of typ;
    type typ_pole2Db is array (M downto N) of typ_pole1D;

    signal pole1Da : typ_pole1D := (0, 1, 2);
    signal pole1Db : typ_pole1D := (3, 4, 5);

    signal pole2Da : typ_pole2Da := ((0, 1, 2), (2, 3, 4), (5, 6, 7));
    signal pole2Db : typ_pole2Db;
begin

    -- lze přiřazovat celá pole stejného typu
    pole1Da <= pole1Db;
    -- lze přiřazovat do položky.
    -- Indexujeme pomocí kulatých závorek a integer indexu
    pole1Da(0) <= 1;

    -- do 2D pole lze přiřazovat pomocí čárkou oddělených indexů
    pole2Da(0,0) <= 1;
    -- následná přiřazení nelze provést kvůli špatné indexaci v prvním případu
    -- a typové kontrole v příkladu druhém
    pole2DA(0) <= pole1Da;
    pole2DA(0, M downto N)  <= pole1Da;

    -- druhý způsob deklarace vícerozměrného pole se liší v indexaci
    pole2Db(0)(0) <= 0;
    -- nyní lze přiřazovat i celé pole, typová kontrola sedí
    pole2Db(0) <= pole1Da;
```

V druhé ukázce práce s poli využijeme předdeklarovaného datového typu std_logic_vector. Použijeme-li k indexaci pole rozsah, pak je z pole vyjmuta jeho část (stejného datového typu, ale jiného rozsahu). Rozsah vyjmutého pole může být maximálně rozměrů rozsahu indexovaného signálu. Nelze indexovat pomocí rozsahu jiné endianity, s výjimkou výběru rozsahu (N to N) nebo (N downto N), který vrátí jednoprvkové pole (0 (down)to 0), a ne položku, jak by se mohlo zdát.

Př. 62 : Práce s poli std_logic_vector

```
architecture RTL of indexing is

      -- ukázka inicializace “std_logic_vector”
      signal a   : std_logic_vector(7 downto 0) := "00000000";
      signal b   : std_logic;
      signal c   : std_logic_vector(0 downto 0);
       -- ukázka inicializace, když se chováme k std_logic_vector jako k “poli”
      signal d   : std_logic_vector(3 downto 0) := ('0','0','0','0');
      signal idx : integer;

begin
```

```
        idx <= 3;
        -- indexovat lze přímo (3) nebo pomocí numerického indexu
        b   <= a(idx);
        -- rozsah (3 downto 3) vytvoří pole o rozměru (0 downto 0)
        c   <= a(idx downto idx);
        -- toto přiřazení realizovat nelze, nesedí datové typy
        b   <= a(idx downto idx);
         -- rozsah (7 downto 4) vytvoří pole (3 downto 0)
        d   <= a(7 downto 4);

end architecture RTL;
```

12.2.4 ZÁZNAM

Záznam (**record**) slouží k organizaci položek různých datových typů do jednoho datového typu, který popisuje vlastnosti nějakého objektu.

Typ záznamu deklarujeme v deklaračních částech kódu:

```
    type jméno_typu_záznamu is
        record
            <položka0 [, položka1, …, položkaN] : typ;]>
    end record [jméno_typu_záznamu];
```

Záznam se skládá ze seznamu středníkem ukončených seznamů čárkou oddělených identifikátorů položek stejného datového typu. Žádná položka záznamu nesmí odkazovat sama na sebe. Práci se záznamy demonstruje kód z příkladu Př. 63.

Př. 63 : Práce se záznamem

```
architecture RTL of example_001 is

    type jméno_typu_záznamu is
        record
            polozka0, polozka1, polozkaN : typ;
    end record jméno_typu_záznamu;

    signal jméno_signálu     : jméno_typu_záznamu := (0, 0, 0);
    constant jméno_konstanty : jméno_typu_záznamu := inicializace;

begin

    jméno_signálu.položka0 <= výraz;
    jméno_signálu          <= jméno_konstanty;
```

Inicializaci proměnné nebo signálu typu záznam můžeme provést pomocí kulatých závorek, které obsahují čárkou oddělené hodnoty inicializace pro jednotlivé položky v pořadí, ve kterém byly umístěny v deklaraci záznamu. Záznamy můžeme vzájemně přiřazovat, pouze pokud jsou stejného typu. K dílčím položkám přistupujeme pomocí notace s tečkou, kterou následuje jméno položky záznamu.

12.2.5 ATRIBUTY

Datové typy, signály (porty a proměnné nevyjímaje) a pole mají předdeklarované atributy, díky kterým lze v kódu přistupovat k jejich vlastnostem. K atributům lze přistupovat pomocí operátoru apostrofu '. Na levé straně operátoru je uveden signál, jehož atribut vybíráme, na pravé pak název atributu. Nejprve si představíme atributy polí.

Tab. 51 : Atributy polí

Atribut	Význam
A'*left(N)*	Levá mez pole.
A'*right(N)*	Pravá mez pole
A'*high(N)*	Nejvyšší mez pole
A'*low(N)*	Nejnižší mez pole
A'*range(N)*	Rozsah pole
A'*reverse_range(N)*	Obrácený rozsah pole
A'*length(N)*	Počet prvků pole
A'*ascending(N)*	vrátí true pokud se jedná o rozsah to

U atributů polí uvádíme názvy např. názvy signálů. Doplníme, že u jednorozměrných polí není třeba parametr N uvádět, jelikož je implicitně 0. Např. pro signál x:

```
signal x : std_logic_vector(1 downto 0);
```

Atribut	Hodnota
x'*left*	1
x'*right*	0
x'*high*	1
x'*low*	0
x'*range*	1 **downto** 0
x'*reverse_range*	0 **to** 1
x'*length*	2
x'*ascending*	false

U atributů typů je situace trochu komplikovanější. Nechť T reprezentuje název datového typu (např. integer) a N je jeho zástupce, např. signál:

```
signal x : integer := 10;
report "x value is " & integer'image(x);
```

Platí, že atribut 'image není atributem zástupce x, ale atributem typu integer. Všechny atributy typů jsou v následující tabulce. Bohužel, ne všechny atributy jsou implementovány pro všechny datové typy. Doplníme, že příkaz report z příkladu výše vypíše v okně simulace zprávu *„x value is 10“*.

Tab. 52 : Atributy typů

Atribut	Význam
skaláry	
T'base	bázový typ T
T'left	levá mez T
T'rigth	pravá mez T
T'high	nejvyšší mez T
T'low	nejnižší mez T
výčtové	
T'pos(N)	pozice N v T
T'val(N)	hodnota v T na N
T'succ(N)	hodnota v T na N+1
T'pred(N)	hodnota v T na N-1
T'leftof(N)	prvek vlevo s ohledem na endianitu
T'rightof(N)	prvek vpravo s ohledem na endianitu
T'image(N)	string reprezentace N
T'value(N)	hodnota řetězce N v T

Zmíníme ještě atributy signálů a řetězců:

Tab. 53 Atributy

Atribut	Význam
signály	
S'event	true, byla-li zaznamenána událost
S'stable	true, nebyla-li zaznamenána událost
S'last_value	předchozí hodnota signálu
entity	
E'simple_name	textový řetězec jména objektu
E'path_name	hierarichické umístění objektu
E'instance_name	jméno instance objektu

12.3 PARALELNÍ PŘÍKAZY

V Př. 60 jsme si ukázali popis jednoduchého číslicového obvodu. Změníme-li libovolně pořadí řádků v příkladu, např.:

```
y <= term_0 or term_1;
term_1 <= a and b and not c;
term_0 <= not a and not b and c;
```

Tak popíšeme a po syntéze získáme obvod shodný s tím, který je na Obr. 97! Všechny příkazy v těle architektury jsou totiž vykonávány paralelně, tj. stejně tak, jako tomu bylo např. v zápisu funkce obvodu v booleově algebře.

Paralelní příkazy ve VHDL slouží majoritně k popisu kombinačních obvodů. Mezi paralelní příkazy patří nepodmíněné přiřazení, podmíněné přiřazení, instance komponenty, blok generování a příklad procesu. V dalším textu se s nimi seznámíme.

12.3.1 NEPODMÍNĚNÉ PŘIŘAZENÍ, VÝRAZ

Základním paralelní příkazem, se kterým jsme se v příkladě seznámili je příkaz nepodmíněného přiřazení ve tvaru:

```
[návěští:] cíl <= výraz;
```

Cíl na levé straně přiřazení může být libovolný signál s možností zápisu, tedy i např. port, ale ne v módu **in**. Datový typ na pravé a levé straně přiřazení musí být shodný. Na pravé straně příkazu se vyjma výrazu může nacházet ještě jiný signál (port, proměnná), konstanta nebo funkce. Výraz nesmí obsahovat výstupní port (to lze až ve VHDL2008). Návěští je nepovinné, ale jeho použití zlepšuje orientaci v kódu.

Do konstant nebo vstupních portů přiřazovat nelze. Stejně tak bychom neměli přiřazovat v různých příkazech do stejného cíle. Tato pravidlo vyplývá z paralelního charakteru jazyka a pokud ho porušíme, pak vytvoříme vícenásobný budič signálu, který typicky vede k chybě syntézy. Zjednodušeně řečeno – syntéza nebude vědět, co má udělat s více vodiči, které jsou „zapojené“ do jednoho vstupu.

Výrazy sestavujeme obdobně, jako v případě jiných programovacích jazyků, tj. pomocí infixového zápisu operátorů, funkcí, procedur a signálů. Priorita operátorů je definována ve specifikaci jazyka a lze ji upravit použitím závorek, tak jak jsme zvyklí. VHDL rozlišuje následující skupiny operátorů:

- Logické **and**, **or**, **nand**, **nor**, **xor**, **not**. Všechny operátory mají stejnou prioritu. Jsou definovány pro datové typy, **bit**(**_vector**), **boolean** a **std_logic**(**_vector**).
- Relační **=**, **/=**, **>**, **<**, **>=**, **<=** (rovno, není rovno, větší, menší, větší rovno, menší rovno). Výstupem všech relačních operátorů je hodnota typu boolean. Operátory rovnosti a nerovnosti jsou definovány pro všechny datové typy. Operátory určující pořadí jsou definovány pro numerické skalární typy.
- Aritmetické operátory jsou uvedeny v Tab. 54. Pořadí operátorů v tabulce odpovídá prioritě.

Tab. 54 Aritmetické operátory s prioritou

Třída	Operátory	Význam	Datové typy
Ostatní	**	mocnina	Základní a odvozené skalární numerické typy (integer, real, atp.)
	abs	absolutní hodnota	
Násobení	*	násobení	
	/	dělení	
	mod	modulo	
	rem	zbytek po dělení	
Znaménka	+	kladné	
	-	záporné	
Sčítání	+	součet	
	-	rozdíl	
	&	sloučení vektorů	složené jednodimenzionální s bázovým typy bit, boolean, std_logic.
Posuv	sll	logický posuv vlevo	
	srl	logický posuv vpravo	
	sla	aritmetický posuv vlevo	

Třída	Operátory	Význam	Datové typy
	sra	aritmetický posuv vpravo	
	rol	rotace vlevo	
	ror	rotace vpravo	

Operátory nejsou implicitně definovány pro všechny datové typy. Např. pro typ std_logic(_vector) jsou definovány pouze operátory sloučení, logických posuvů a rotací. Obdobně není definován logický posuv vlevo např. pro fyzikální datový typ time. Problematikou aritmetických operací se budeme dále zabývat v kapitole 12.6 .

Konstantní numerické výrazy

Přiřazení skalární hodnoty ve VHDL se provádí pro každý datový typ odlišně. Zápis konstantních hodnot numerických typů je poměrně přímočarý, pokud používáme desítkovou soustavu – číslo prostě zapíšeme na pravou stranu přiřazení. Do čísla můžeme vložit znak podtržítka, chceme-li např. zdůraznit řády.

Př. 64 Přiřazení numerické konstanty

```
architecture example of assignment is

	signal numeric_object : integer;
	type small_t is range 0 to 15;
	signal small          : small_t;

begin

	-- zápis dekadického čísla 1000000 (0xFFFF2153)
	numeric_object <= 1_000_000;
	-- zápis hexadecimálního čísla
	numeric_object <= -16#FF#;
	-- zápis v oktalové soustavě
	numeric_object <= 8#70#;
	-- zápis v trojkové soustavě
	numeric_object <= 3#012#;
	-- zápis dekadického čísla v mezích datového typu
	small          <= 10;
	-- nelze přiřadit. Hodnota je mimo rozsah datového typu!
	small          <= -1;

end architecture example;
```

Pro zápis čísel v jiné soustavě než desítkové musíme hodnotu obalit znaky maltézského kříže # a před ně zapsat v jaké soustavě je zápis proveden. VHDL podporuje soustavy o základech v intervalu celých čísel <2, 16>. Při přiřazení musíme zohlednit rozsah datového typu, viz poslední přiřazení z Př. 64. Obdobně je třeba pohlídat znaménka, která uvádíme před maltézským křížem. Pamatujme, že záporná čísla jsou ve VHDL vždy kódována dvojkovým doplňkem.

Pro (výčtové) datové typy bit a boolean, které nabývají jen dvojice hodnot – log. 0 a log. 1 používáme zápis s buď s jednoduchými uvozovkami - ’0’ nebo ’1’, či s pomocí hodnot true a false respektive. Způsob zápisu hodnot std_logic a bit se shoduje v případě log. 0 a 1, ale u std_logic je ještě možné zapisovat hodnoty U, X, Z, W, L, H a -.

Zápis std_logic(_vector)

Pro pole bitů bit_vector a std_logic_vector používáme zápis ohraničený dvojitými uvozovkami (stejně jako pro řetězce znaků), např "01UXZWLH-". Před uvozovky můžeme uvést znak, který označuje bázi, ve které je vektor zapsán. Knihovna IEEE1164 podporuje zápis Binárních (implicitně), Oktalových a Hexadecimálních vektorů. Při přiřazování vektorů v jiné bázi než ve dvojkové, nelze zapisovat jiné hodnoty než '0' nebo '1', zatímco v dvojkové lze ve vektoru kombinovat všechny hodnoty std_logic. Pro nedvojkové báze dále platí, že je-li délka výrazu na pravé straně po převodu do binárního kódu delší než rozsah levé strany, pak nelze přiřazení provést. Připomeňme, že jeden znak v šestnáctkové soustavě vyžaduje 4 bity, v oktalové 3.

Chceme-li zapsat do všech prvků pole stejnou hodnotu, pak můžeme využít konstruktu (others => konstanta). Tento konstrukt je výhodné použít, když neznáme délku rozsahu signálu na pravé straně. Chceme-li např. vynulovat vícerozměrné pole, pak lze využít vícenásobného vnoření. Konstantu nahradíme others => a tak činíme až do vyčerpání dimenze pole. Ukázky zápisů konstant jsou uvedeny v následujících příkladech. Výše uvedené shrnuje Př. 65.

Př. 65 Přiřazení hodnot do std_logic(_vector)

```
architecture example of assignment_vec is

	signal x   : std_logic;
	signal x_v : std_logic_vector(11 downto 0);

	type mem_t is array (7 downto 0) of std_logic_vector(7 downto 0);
	signal mem : mem_t;

begin

	-- přirazeni '0'/'1'
	x   <= '0';
	-- binární zápis do vektoru.
	x_v <= "00001111ZZZZ";
	-- zápis v šestnáctkové a oktalové soustavě
	x_v <= X"ABC";
	x_v <= O"7654";
	-- vynulovaní vektoru a vícerozměrného pole pomocí others
	x_v <= (others => '0');
	mem <= (others => (others => '0'));

end architecture example;
```

VHDL umožňuje vektory na pravé straně přiřazení nejen „rozdělovat“ (viz indexace polí), ale i slučovat pomocí operátoru slučování &. Slučování si lze představit jako napojování jednotlivých polí za sebe. Názorně je ukázáno na příkladu.

Slučovat vzájemně je možné různě dlouhá pole nebo bázové prvky pole. Stejně jako v jiných případech je nutné, aby rozsahy polí na obou stranách přiřazení odpovídaly. Chceme-li doplnit vektor zleva nebo zprava konstantními hodnotami, nemůžeme využít přiřazení (others =>), ale můžeme využít indexace na levé straně přiřazení. Ve VHDL2008 tento problém odpadá a others lze využít i pro doplnění vektorů.

Př. 66 Slučování

```
architecture RTL of concat is

	signal a_vec   : std_logic_vector(7 downto 0);
	signal a, b, c : std_logic;
	signal b_vec   : std_logic_vector(3 downto 0);

begin
	-- slučovat lze téměř cokoliv
	a_vec             <= a & b & c & '1' & b_vec;
	-- doplňovat vektory lze až od VHDL2008
	a_vec             <= b_vec & (others => '0');
	-- musíme se spokojit s dvojicí přiřazení
	a_vec(7 downto 4) <= b_vec;
	a_vec(3 downto 0) <= (others => '0');

end architecture RTL;
```

Aggregate

Operátor aggeregate () umožňuje zapsat hodnotu pole (tedy i vektoru) a záznamu do objektu na levé straně přiřazení.

$$(volba_0, volba_1, \ldots, volba_N)$$

Volby se dělí na poziční a jmenné. Poziční volby musí předcházet jmenným. Jmenná volba specifikuje pozici nebo rozsah pozic, do kterých se bude přiřazovat bázový typ. Poslední jmennou volbou může být výraz others. Pozice přiřazení poziční volby je dána umístěním jejího zápisu v aggregate a deklarací datového typu do kterého se přiřazuje.

Rozumíte? Já ne.

Ukažme si to jednoduchých příkladech:

```
type mem_t is array (7 downto 0) of integer;
-- toto už známe intuitivně z inicializace polí
signal mem0 : mem_t := (1, 2, 3, 4, 5, 6, 7, 8);
signal mem1 : mem_t := (10, 5 => 7, (4 downto 3) => 4, others => 0);
-- vytvořené pole bude mít hodnotu (10, 0, 7, 0, 4, 4, 0, 0)
```

U prvního aggregate jsou všechny volby poziční. Zapsané hodnoty budou umístěny tak, jak byl deklarován datový typ mem_t (7 **downto** 0), tj. zleva doprava vzestupně od 1 do 8.

Druhé aggregate kombinuje poziční přiřazení s jmennými. Na pozici 7 bude umístěna 10 – první poziční přiřazení vždy přiřazuje na ‘high pozici. Jmenné přiřazení 5 => 7 umístí na index 5 hodnotu 7. Rozsah (4 **downto** 3) přiřadí na pozice 4 a 3 hodnotu 4. Poslední jmenné přiřazení others => 0 přiřadí na ostatní pozice nulové hodnoty.

Pojďme si ukázat další příklady. První ukazuje validní syntax, ale asi bychom si takové zápisy v běžném kódu odpustili:

```
-- Zkusme přiřadit do vektoru 3 downto 0 hodnotu "00X1"
signal vec0 : std_logic_vector(3 downto 0) := (1=>'X', 0=>'1', others => '0');
-- můžeme, ale moc si oproti := "00X1" nepomůžeme
signal vec1 : std_logic_vector(3 downto 0) := ("00", 'X', others => '1');
-- NEMŮŽEME - "00" není bázového typu std_logic
signal vec2 : std_logic_vector(3 downto 0) := ((3 downto 2) => '0', 'X', others => '1');
-- NEMŮŽEME – poziční přiřazení musí být před jmenným.
```

Další dva aggregate ukazují, co nemůžeme – přiřazovat jiné bázové typy nebo měnit pořadí jmenných a pozičních voleb. Všimněte si jmenné volby pomocí rozsahu u druhého aggregate.

Aggregate může být ale i velmi elegantní. Například maximální a minimální hodnotu čísla ve dvojkovém doplňku můžeme napsat jako:

```
signal min : std_logic_vector(3 downto 0) := ('1', others => '0');
signal max : std_logic_vector(3 downto 0) := ('0', others => '1');
```

Aggregate můžeme dát i na levou stranu přiřazení, ačkoliv to ne všechny syntézní nástroje dovolují. Musíme mít na paměti, že datové typy při přiřazení musí být jasné:

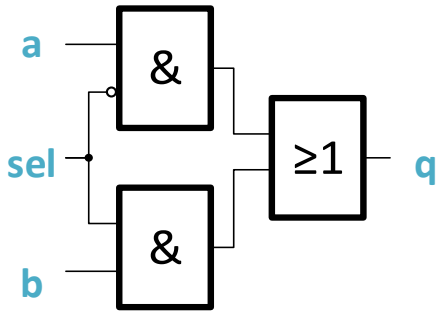
```
    -- aggregate může být i na pravé straně přiřazení
    signal a,b,c : std_logic;
begin

    -- ale musí být definován datový typ
    std_logic_vector'(c, b, a) <= "000";
```

12.3.2 Podmíněné přiřazení

V Př. 67 je zobrazen kód, který realizuje dvoubitový multiplexer pouze s pomocí příkazů nepodmíněného přiřazení. K jeho popisu jsme mohli dojít tak, že vytvoříme pravdivostní tabulku, zapíšeme ji do Karnaughovy mapy, provedeme minimalizaci a výslednou MNDF zapíšeme pomocí výrazů v jazyce VHDL. A je to!

Př. 67 Strukturně popsaný multiplexor ve VHDL

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

entity multiplexor is
      port (
            a, b, sel : in  std_logic;
            q         : out std_logic
      );
end entity multiplexor;

architecture STRUCT of multiplexor is
begin
      q <= (not sel and a) or (sel and b);
end architecture STRUCT;
```



Obr. 98 Obvod z Př. 67

Proces tvorby popisu je evidentně pro větší obvody pracný a se vzrůstající komplexitou hrozí riziko, že v jeho průběhu uděláme chybu. Přepsání výrazů do VHDL je na závěr spíš trestem než ulehčením práce. Přitom v každém smysluplném jazyce existují konstrukce, které by funkci, jež multiplexer provádí popsaly jednodušeji – třeba pomocí podmínek. Nejinak je tomu i u VHDL.

VHDL disponuje dvojicí paralelních příkazů, které realizují podmíněné přiřazení – příkazem podmíněného přiřazení **when** a příkazem výběrového přiřazení **with-select**. Syntax **when** si ukážeme nejprve obecně a posléze ji vysvětlíme na příkladu:

```
[návěští:] cíl <= výraz_0 when podmínka_0
            [else výraz_1  when podmínka_1
             else výraz_2  when podmínka_2
             ...]
            [else výraz_n];
```

Syntax příkazu si lze vyložit tak, že je do cíle přiřazení přiřazen výraz_0, právě když je splněna podmínka_0. Příkaz má další dvě „nepovinné“ části.

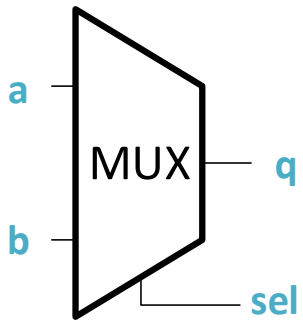
První nepovinná část obsahuje prioritně zadané podmínky přiřazení. Příkaz postupně vyhodnocuje podmínky a narazí-li na nějakou splněnou, vyhodnotí výraz, přiřadí ho na levou stranu a je ukončen. Priorita tedy spočívá v tom, že k přiřazení dojde u první splněné podmínky. Nenarazí-li na žádnou podmínku, jež je vyhodnocena jako true, je příkaz ukončen, a to nejlépe pomocí části druhé nepovinné části [**else** výraz_n].

Nepovinná část z posledního řádku [**else** výraz_n] určuje hodnotu, která se přiřadí v případě vyhodnocení všech předchozích podmínek jako false. Pokud tuto nepovinnou část vynecháme, tak typicky popíšeme hladinový klopný obvod typu D, řízený hodnotou, a ne hodinovým signálem, což není moc dobrá návrhářská praxe. Proto „nepovinnou“ část [**else** výraz_n] raději uvádíme vždy.

Návrh obvodu z Př. 67 je se znalostí příkazu podmíněného přiřazení možné značně zjednodušit a získat obvod, který se na venek chová úplně stejně. Rozdíl mezi strukturním a behaviorálním popisem se objeví po syntéze. Jelikož příkaz **when** vede na generování multiplexorů bude struktura trojice hradel z Obr. 98 nahrazena nějakým multiplexorem. Jestli ten bude ve výsledku implementován pomocí hradel závisí pouze na softwarových nástrojích a cílové technologii.

Př. 68 Behaviorálně popsaný multiplexor ve VHDL

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

entity multiplexor is
      port(
            a, b, sel : in  std_logic;
            q         : out std_logic
      );
end entity multiplexor;

architecture behavioral of multiplexor is
begin
      q <= a when sel = '0' else b;
```



Obr. 99 Obvod z Př. 67

```
end architecture STRUCT;
```

12.3.3 VÝBĚROVÉ PŘIŘAZENÍ

Zatímco příkaz **when** se používá pro tvorbu kombinační logiky, s prioritním charakterem přiřazení, příkaz **with-select** slouží pro popis kombinačních struktur v nichž priorita úlohu nehraje. Obecná syntaxe příkazu je následující.

```
[návěští]: with výraz_výběru select cíl <=
               výraz_přiřazení0 when volba00 [| volba01 | ...],
              [výraz_přiřazení1 when volba10 [| volba11 | ...],
               ...
               výraz_přiřazenín when volban0 [| volban1 | ...];
```

Za nepovinným návěštím následuje blok `with výraz_výběru select` cíl. Výraz_výběru může být libovolný výraz, který řídí přiřazení. Typicky se volí porty nebo signály, někdy bývá využit výraz sloučení nebo výběru. Cílem přiřazení je signál, do kterého se bude hodnota přiřazovat. Samotný výběr probíhá v následujících řádcích.

Je-li vyhodnoceno, že výraz_výběru odpovídá kritériím seznamu voleb (oddělených znakem | se sémantikou logického nebo), pak je do cíle přiřazen výraz_přiřazení z daného řádku. Volby mohou být libovolné konstanty, rozsahy nebo klíčové slovo **others**. **Others** musí být použito samostatně a musí být umístěno jako poslední řádek příkazu – říká, co se má přiřadit, když neplatí žádná z dříve zapsaných voleb.

Poměrně komplikovaná syntax se v praxi uplatňuje poměrně snadno. Jako výraz_výběru a cíl budeme používat nějaký signál nebo port, výrazy_přiřazení jsou typicky konstanty a volby jakbysmet. Oddělovač seznamu voleb se v praxi nepoužívá. Nezapomeneme však na poslední řádek napsat vždy volbu others, abychom omylem nevytvářeli hladinové klopné obvody.

Př. 69 Dekodér kódu 1 ze 4 pomocí výběrového přiřazení

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

entity decoder1of4 is
      port (
            BCD : in  std_logic_vector(1 downto 0);
            q   : out std_logic_vector(3 downto 0)
      );
end entity decoder1of4;

architecture RTL of decoder1of4 is
begin

      dec : with BCD select q <=
            "0001" when "00",
            "0010" when "01",
            "0100" when "10",
            "1000" when "11",
            "1111" when others;
```

```
end architecture RTL;
```

Obecně se tvrdí, že podmíněné přiřazení zpracovává podmínky prioritně, zatímco výběrové přiřazení prioritní není. Syntézním výstupem obou příkazů bude ale kombinační logika zakončená multiplexorem. Jaký je tedy rozdíl mezi příkazy? Můžeme si ho osvětlit na příkladu.

Př. 70 Rozdíl podmíněného a výběrového přiřazení

Pásový dopravník převáží výrobek po stanovištích A, B, C opatřených stejnojmennými detektory. Rozměry objektu a princip linky vylučují, aby se objekt nacházel na více stanovištích zároveň, nebo aby bylo na pásu přítomno více objektů. Navrhněte obvod, který dekóduje pozici dopravníku do binárního váhového kódu.

Popišme obvod ve VHDL.

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

entity WithVsWhen is
	port(
		signal a, b, c  : in  std_logic;
		signal q_c, q_s : out std_logic_vector(1 downto 0)
	);
end entity WithVsWhen;

architecture RTL of WithVsWhen is

begin
	-- podmíněné přiřazení
	q_c <= "01" when a = '1'
		else "10" when b = '1'
		else "11" when c = '1'
		else "00";

	-- výběrové přiřazení
	with std_logic_vector'(a & b & c) select q_s <=
		"01" when "100",
		"10" when "010",
		"11" when "001",
		"00" when others;

end architecture RTL;
```

Ze zadání je zřejmé, že nemůže dojít k situaci A = B = C = 1. Popíšeme-li obvod pomocí podmíněného přiřazení, pak tuto podmínku ale připouštíme – první řádek totiž vyhovuje všem situacím, kdy a = 1 (termům abc, ab$\bar{c}$, a$\bar{b}$c, a$\bar{b}\bar{c}$) a vede k pravdivostní tabulce, která bude obsahovat více jedniček. Výběrové přiřazení je v tomto příkladu specifičtější a vede k funkci, jejíž pravdivostní tabulka tolik jedniček neobsahuje.

Pozorný čtenář si povšimne specifického výrazu výběru u výběrového přiřazení – `std_logic_vector'(a & b & c)`. Nazývá se kvalifikovaný výraz (qualified expression) a umožňuje ad-hoc vytvářet signály datových typů specifikovaných na levé straně přiřazení, aniž bychom museli deklarovat signály daného datového typu.

12.3.4 INSTANCE KOMPONENTY

VHDL podporuje hierarchické využívání a opětovné používání již vytvořených obvodů. Obvody (entity), které jsou využity uvnitř jiných architektur budeme nazývat instancí komponenty nebo instancí entity. Na příkladu si ukážeme, jak můžeme znovu použít základní multiplexor popsaný v Př. 68 pro konstrukci multiplexoru se čtveřicí datových a dvojicí výběrových vstupů

Př. 71 : Instance komponenty

```
library ieee;
use ieee.std_logic_1164.all;
entity hierarchy is
    port(
        a   : in  std_logic_vector(3 downto 0);
        sel : in  std_logic_vector(1 downto 0);
        q   : out std_logic
    );
end entity hierarchy;

architecture Structural of hierarchy is
    -- v hlavičce deklarujeme použití jiné entity – nazýváme ji komponenta
    component multiplexor
        port(
            a, b, sel : in  std_logic;
            q         : out std_logic
        );
    end component multiplexor;
    signal mx : std_logic_vector(1 downto 0);
begin
    -- vytvoříme instanci entity a její vstupy přiřadíme k signálům architektury
    mux00_inst : component multiplexor
        port map(
            a   => a(0),
            b   => a(1),
            sel => sel(0),
            q   => mx(0)
        );

    -- máme-li VHDL zdroják, pak lze využít instanci entity
    mux01_inst : entity work.multiplexor
        port map(
            a   => a(2),
            b   => a(3),
            sel => sel(0),
            q   => mx(1)
        );

    -- port map lze psát i zkráceně, pokud si věříme
    mux1_inst : component multiplexor
```

```
        port map(mx(0), mx(1), sel(1), q);

end architecture Structural;
```

Deklarace komponenty je stejná jako deklarace entity, jíž přísluší. Vzájemně se liší v nahrazení klíčového slova **entity** slovem **component**. Deklaraci komponenty vkládáme do deklarační části architektury. Máme-li komponentu deklarovanou, pak můžeme instanci komponenty provést pomocí příkazu instance komponenty:

```
    jméno_instance : component jméno_komponenty[(architektura)]
        [generic map (
            mapování_generických_parametrů
        );]
        port map (
            mapování_portů
        );
```

Na rozdíl od ostatních paralelních příkazů je jméno_instance komponenty povinné; zaručuje vytvoření hierarchicky platného jména uvnitř naší architektury. V kulatých závorkách za jménem komponenty můžeme nepovinně vybrat architekturu, jejíž instanci chceme vytvořit. Následuje dvojice bloků – **port map** a nepovinný **generic map**.

Port map mapuje porty instancované komponenty signálům platným uvnitř architektury, do níž instanci komponenty vkládáme. VHDL podporuje dva způsoby mapování portů a signálů.

V úplném zápisu přiřazujeme pomocí operátoru =>

```
port map (
       port0  => výraz0,
       port1  => výraz1,
       …
       portn-1 => výrazn-1
);
```

Na levé straně mapování se nachází název portu instancované komponenty, na pravé výraz (typicky signál nebo jeho část), na který chceme port mapovat. Nechceme-li výstupní port mapovat, nahradíme výstupní signál klíčovým slovem **open**. Vstupní porty musíme mapovat vždy. U úplného zápisu nezáleží na pořadí zápisů mapování a stejně jako u všech přiřazení ve VHDL platí, že se datové typy levé a pravé strany mapování musí shodovat. Jednotlivá mapování oddělujeme čárkami; za posledním mapováním čárku neuvádíme. Ukázka úplného přiřazení je v 0 u instancí mux00_inst a mux01_inst.

Příklad druhé varianty přiřazení – zkráceného zápisu – představuje instance mux1_inst v témže příkladě. U zkráceného zápisu nahrazujeme dvojice mapování seznamem signálů seřazeným dle pořadí zápisu mapovaných portů v instancované komponentě. Jelikož je nutné znát pořadí portů komponenty je možné, že při zkráceném zápisu dojde k chybám, tudíž jej využíváme zřídka.

Disponujeme-li zdrojovým souborem VHDL entity, již instancujeme, pak instanci komponenty můžeme nahradit instancí entity. V takovém případě deklaraci komponenty do deklarační části architektury umisťovat nebudeme. Knihovna work, kterou instance entity využívá je implicitní knihovnou každého projektu ve VHDL. Pravidla pro mapování portů u instance komponenty i entity jsou shodná. Příklad instance entity nalezneme v 0 v instanci mux01_inst. Úplná syntax je obdobná s instancí komponenty:

```
jméno_instance : entity work.jméno_entity[(architektura)]
       [generic map(
            mapování_generických_parametrů
       );]
       port map(
            mapování_portů
       );
```

Instanci komponenty je možné využít i když je popis obvodu dodán v jiném formátu než ve VHDL (syntetizovaný netlist, jiný HDL jazyk), instanci entity nikoliv. **Generic map** si vysvětlíme na konci následující kapitoly.

12.3.5 GENERIC, GENERATE, GENERIC MAP

Struktura číslicových obvodů se dá často popsat algoritmicky – části obvodů se opakují (paralelní registry, aritmetické obvody), různé varianty obvodu se liší pouze v jednom parametru (sčítačka-odečítačka v dvojkovém doplňku), atp. K tvorbě parametrizovatelných obvodů slouží konstanty deklarované v generické části deklarace entity (12.1.2).

Syntaxe bloku **generic** se typicky nachází před blokem **port** a v mnohém se jeho syntaxi podobá. Nachází se za klíčovým slovem **is** a obsahuje středníkem oddělený seznam generických konstant. Za poslední konstantou středník nepíšeme.

```
generic (
       jméno_generické_konstanty : datový_typ := inicializace;
);
```

Na rozdíl od deklarace portů je nutné u generických konstant uvádět inicializační hodnotu. Standard jazyka do VHDL2008 nespecifikoval pořadí vyhodnocení inicializací, a proto nemůžeme odkazovat v inicializacích na jiné konstanty – mohla by vzniknou deklarace kruhem. Konstanty je možné využít už při specifikaci datových typů portů, avšak výrazy, které typy portů specifikují musí být staticky vyhodnotitelné v době syntézy a musí zůstat konstantní.

Hodnoty generických konstant lze přepsat při instanci komponenty. Do objektů konstant v těle architektury přiřazovat nemůžeme. Pravidla si můžeme ukázat na příkladu deklarace entity paměti s datovou sběrnicí o šířce DWIDTH, která umožňuje adresovat libovolný počet adresních buněk specifikovaný konstantou DEPTH.

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

```
entity memoryA is
    generic(
        DWIDTH : integer := 10;
        DEPTH  : integer := 16
    );
    port(
        din  : in  std_logic_vector(DWIDTH - 1 downto 0);
        dout : out std_logic_vector(DWIDTH - 1 downto 0);
        addr : in  std_logic_vector(integer(ceil(log2(real(DEPTH)))) - 1 downto 0)
    );
end entity memoryA;
```

Povšimněme si, že jsme generickou konstantu DWIDTH použili již při deklaraci portů din a dout. U port addr bylo nutné využít funkce ceil a log2 z knihovny ieee.math_real, které provádějí zaokrouhlení dolů a dvojkový logaritmus respektive. Jako argument funkce přijímají pouze datový typ **real** a proto je nutné provést příslušné konverze datových typů.

Generické konstanty jsou obzvláště užitečné v kombinaci s paralelním příkazem **generate**. Ten můžeme využít pro tvorbu struktur, které se opakují, nebo podmíněné vytváření popisů obvodu. Příkaz **generate** má dvě varianty:

```
návěští: if podmínka generate
    [deklarace
begin]
     blok_paralelních_příkazů;
end generate [návěští];

návěští: for řídící_proměnná in rozsah generate
    [deklarace
begin]
     blok_paralelních_příkazů;
end generate [návěští];
```

V první uvedené variantě příkazu musí být k vykonání bloku paralelních příkazů splněna booleovská podmínka uvedená za klíčových slovem **if**. Podmínka musí být staticky vyhodnotitelná při syntéze. Není-li podmínka splněna, pak není blok příkazů vykonán. Větev **else** v případě **generate** neexistuje a nahrazujeme jej dvojicí generujících příkazů; jeden je vykonán, když je podmínka splněna, druhý, když není.

Druhá varianta příkazu využívá iterační schéma **for**. Řídící proměnná schéma je dynamicky pro **generate** vytvořena, nelze do ní přiřazovat a je iterována v statickém rozsahu uvedeném v příkazu. Její datový typ je odvozen od datového typu rozsahu. Proměnná je platná pouze uvnitř příkazu. Smyčky můžeme vnořovat, názvy řídící proměnných vnořených smyček musí být rozdílné.

Ukázku použití příkazu **generate** v obou variantách demonstruje Př. 72. Je v něm popsána vícebitová sčítačka/odečítačka ve dvojkovém doplňku. Bitovou šířku sčítačky určuje generická konstanta C_WIDTH typu integer. Druhá generická konstanta, booleovský SUB, určuje, zda bude obvod adder realizovat operaci a + b, nebo a - b.

Generující příkaz nbitadder v architektuře obvodu je řízen iteračním schématem **for** v rozsahu <0, C_WIDTH-1>. V deklarační části příkazu jsou vytvořeny signály carry_chain a b_adjusted, které jsou platné jen uvnitř příkazu a jsou použity k realizaci přenosu vícebitové sčítačky a inverzi vstupní proměnné b. Uvnitř generující smyčky jsou vnořeny tři paralelní příkazy. První dva – is_sub a not_sub – ilustrují použití příkazu **generate** s podmínkou **if** bez možnosti využití **else**. Třetím příkazem je instance entity full_adder (full_adder_inst), jejíž kód realizuje úplnou jednobitovou sčítačku (viz Obr. 61 a Obr. 62).

Př. 72 : Užití příkazu **generate** ke konstrukci vícebitové sčítačky

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

entity adder is
    generic(
        C_WIDTH : integer := 10;
        SUB     : boolean := false
    );
    port(
        a, b : in  std_logic_vector(C_WIDTH - 1 downto 0);
        s    : out std_logic_vector(C_WIDTH - 1 downto 0)
    );
end entity adder;

architecture struct of adder is
begin

    nbitadder : for i in 0 to C_WIDTH - 1 generate
        signal carry_chain : std_logic_vector(C_WIDTH downto 0);
        signal b_adjusted  : std_logic_vector(b'range);
    begin

        is_sub : if SUB generate
            carry_chain(0) <= '1';
            b_adjusted     <= not b;
        end generate;

        not_sub : if not SUB generate
            carry_chain(0) <= '0';
            b_adjusted     <= b;
        end generate;

        full_adder_inst : entity work.full_adder
            port map(
                a    => a(i),
                b    => b_adjusted(i),
                cin  => carry_chain(i),
                cout => carry_chain(i + 1),
                sum  => s(i)
            );
    end generate;
end architecture struct;
```

S hotovým popisem vícebitové generické sčítačky se můžeme vrátit o kapitolu zpět a dokončit popis příkazu instance komponenty/entity. Ten umožňuje zahrnout nepovinný blok **generic map**. Blok uvádím, pokud entita jejíž instanci provádíme obsahuje blok **generic** a chceme generické konstanty v její instanci měnit.

Syntaxe mapování generických příkazů je podobná mapování portů. Můžeme využít jak úplného, tak zkráceného zápisu. U mapování generických konstant navíc můžeme dílčí konstanty vynechat s tím, že budou v instanci nahrazeny inicializační hodnotou uvedenou v popisu instancované entity.

```
generic map (
      generická_konstanta0  => výraz0,
      generická_konstanta1  => výraz1,
      …
      generická_konstantan-1 => výrazn-1
);
```

Instanci entity adder z Př. 72 ilustruje fragment kódu na následujícím příkladu

Př. 73 : Instance entity s generickými parametry

```
architecture RTL of dummy is
    constant C_WIDTH : integer := 16;
    signal a : std_logic_vector(C_WIDTH - 1 downto 0);
    signal b : std_logic_vector(C_WIDTH - 1 downto 0);
    signal s : std_logic_vector(C_WIDTH - 1 downto 0);

begin

    adder_generic_inst : entity work.adder
        generic map(
            C_WIDTH => C_WIDTH,
            SUB     => false
        )
        port map(
            a => a,
            b => b,
            s => s
        );

end architecture RTL;
```

12.4 PROCES A SEKVENČNÍ PŘÍKAZY

V předchozím textu jsme si představili paralelní příkazy – příkazy, jejichž vykonávání není prováděno v pořadí, ve kterém byly zapsány do zdrojového kódu. Příkazy ve VHDL ale mohou být vykonávány i sekvenčně – tj. v pořadí ve kterém byly napsány. Sekvenční příkazy musí být umístěny v těle speciálního paralelního příkazu – příkazu **process**. Syntaxe příkazu je následující:

```
    [návěští] : process[(citlivostní seznam)]
        [deklarační část]
```

```
    begin
        [sekvenční část]
    end process [návěští];
```

Po nepovinném návěští následuje klíčové slovo **process**. Za ním je v závorkách uveden nepovinný citlivostní seznam (někdy uváděn též jako seznam citlivostních proměnných) – čárkou oddělený seznam signálů zdrojového kódu, jejichž změna způsobí „spuštění“ procesu. Do seznamu nelze uvádět signály, které nelze číst.

Následuje deklarační část procesu, ve které lze uvádět deklarace proměnných a konstant, funkcí a procedur nebo deklarace typů. Vše, co je uvedené v deklarační části je platné pouze uvnitř procesu. V procesu nelze deklarovat signály, ale pouze proměnné. Rozdíl bude vysvětlen v následujícím textu.

V těle procesu, které se nachází mezi klíčovými slovy **begin** a **end process** jsou uvedeny sekvenční příkazy. Jsou to:

- nepodmíněný příkaz přiřazení do signálu tak, jak je definován v 12.3.1
- přiřazení do proměnné
- volání procedur a funkcí
- podmínka **if** a příkaz **case**, cykly **for** a **while** a příkazy **next** a **exit**, které s nimi souvisí
- příkaz **wait**

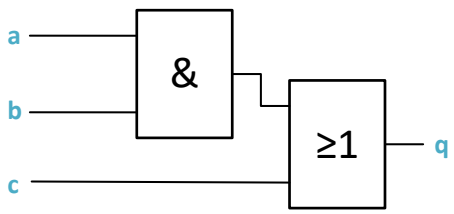
12.4.1 Spouštění a vyhodnocení procesu, citlivostní seznam

Proces je spouštěn, pokud se změní jakýkoliv signál uvedený v citlivostním seznamu. Pokud proces seznam citlivostních proměnných neobsahuje, pak je při simulaci spouštěn neustále – po ukončení procesu je proces spuštěn znovu.

V korektním citlivostním seznamu uvádíme všechny signály, které se nacházejí na pravých stranách přiřazení uvnitř procesu nebo jsou součástí výrazů podmínek. VHDL2008 umožňuje nahradit seznam citlivostních proměnných klíčovým slovem **all**, které zajistí, že bude citlivostní seznam vytvořen automaticky. Chybně sestavený citlivostní seznam demonstruje příklad Př. 74, který pomocí příkazu proces realizuje logickou funkci $q = ab + c$ popsanou strukturně viz Obr. 100. Funkce je realizována procesem dle **p0** dle syntax VHDL 2003. Proces **p1** využívá výhod VHDL 2008.

Př. 74 Chybný citlivostní seznam procesu

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

entity SimpleProcess is
    port(
        a, b, c : in  std_logic;
        q0, q1  : out std_logic
    );
end entity SimpleProcess;

architecture RTL of SimpleProcess is
```



Obr. 100 : Schéma obvodu z Př. 74

```
    signal a_and_b0 : std_logic;
    signal a_and_b1 : std_logic;
begin

    p0 : process(a, b, c)
    begin
        a_and_b0 <= a and b;
        q0       <= a_and_b0 or c;
    end process;

    p1 : process(all)
    begin
        a_and_b1 <= a and b;
        q1       <= a_and_b1 or c;
    end process;

end architecture RTL;
```

V Př. 74 je citlivostní seznam prvního procesu zdánlivě správně – obvod je evidentně citlivý na změny signálů (portů) **a**, **b** i **c**. Pokud bychom provedli syntézu obvodu, zřejmě bychom získali obdobu toho, co je zobrazeno na Obr. 100 a varování o neúplném citlivostním seznamu procesu **p0**. Pokud bychom ale provedli simulaci obvodu, zjistili bychom, že obvod, který je v procesu **p0** popsán, korektně nepracuje.

Vyhodnocení procesu v simulátoru totiž funguje jinak, než bychom dle zkušeností s paralelními příkazy očekávali. Při spuštění procesu je zbytek obvodu jakoby zamražen. Jednotlivé příkazy v procesu jsou sekvenčně vykonávány, ale výsledky **přiřazení do signálů** nejsou okamžitě propagovány dál – mohlo by totiž dojít k tomu, že výpočet nebude kvůli strukturním závislostem nikdy dokončen (např. kvůli vzájemným interakcím různých procesů by vznikaly algebraické smyčky). Namísto toho jsou výsledky vyhodnocení příkazů přiřazení do signálů přiřazeny až po ukončení procesu nebo po zavolání příkazu **wait**.

Z výše uvedeného je zřejmé, že se hodnota **a_and_b0** v procesu **p0** změní až po ukončení procesu a výstupní hodnota **q0** bude vypočtena z hodnoty **a_and_b0**, jež byla platná v okamžik volání procesu, tedy při poslední změně **a**, **b** nebo **c**. Do korektního citlivostního seznamu **p0** tak patří i proměnná **a_and_b0**. Háček je v tom, že její zavedení způsobí dvojí volání procesu. První volání bude způsobené změnou na signálech **a**, **b** nebo **c** a druhé bude následovat hned po ukončení procesu kvůli očekávané změně signálu **a_and_b0**.

Proces **p1** je díky použití **all** namísto citlivostního seznamu korektní jak pro syntézu, tak pro simulaci, ale při simulaci bude zpracován dvakrát. Chceme-li se dvojímu výpočtu vyhnout, musíme zavést způsob, kterým do „signálu“ hodnotu přiřadíme okamžitě, aniž bychom ovlivnili zbytek obvodu. VHDL toto řeší zavedením proměnných.

12.4.2 Proměnné

Proměnné jsou deklarovány v hlavičce procesu …

```
variable proměnná0 [,proměnná 1,..., proměnná n] : datový_typ [:= inicializace];
```

… a přiřazujeme do nich pomocí příkazu přiřazení proměnné:

```
cíl := výraz;
```

Pro proměnné a příkaz přiřazení do proměnné platí obdobná pravidla, jako pro signály a nepodmíněné přiřazení do signálů, s tím rozdílem, že do proměnných lze zapisovat a číst z nich pouze v rámci procesu, ve kterém byly deklarovány. K aktualizaci hodnoty proměnných dochází bezprostředně po vykonání příkazu přiřazení. Proměnné mají navíc persistentní charakter – proces si pamatuje hodnotu přiřazení do proměnné ze svého posledního volání. Je-li proces volán poprvé, je hodnota proměnné rovna inicializační.

S využitím proměnné, můžeme popis obvodu z Př. 74 upravit, viz 0.

Př. 75 : Využití proměnné v těle procesu.

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

entity SimpleProcess is
    port(
        a, b, c : in  std_logic;
        q0  : out std_logic
    );
end entity SimpleProcess;

architecture RTL of SimpleProcess is
begin

    p2 : process(a, b, c)
        variable a_and_b : std_logic := '0';
    begin
        a_and_b  := a and b;
        q0       <= a_and_b or c;
    end process;

end architecture RTL;
```

Po úpravách bude obvod syntetizován bez varování, jelikož je citlivostní seznam úplný, a navíc zabráníme dvojímu volání procesu v simulátoru. V simulaci bude proces spuštěn při změně signálů **a**, **b** nebo **c**, dále bude vyhodnocen výraz **a and b**, výsledek výrazu budu okamžitě uložen do proměnné **a_and_b** a na dalším řádku bude hodnota použita pro výpočet výrazu **a_and_b or c**, který bude díky následnému ukončení procesu okamžitě uložen do signálu **q0**.

Nevýhodou zavedení proměnné je to, že díky svému lokálnímu charakteru ji nebude možné číst v jiném paralelním příkazu a také ji nebude možné zobrazit na časovém simulačním výstupu většiny simulátorů. Pro náhled na aktuální hodnotu proměnné bude nutné simulaci krokovat, nebo proces opatřit kontrolními výpisy.

12.4.3 PODMÍNKY

Řízení toku sekvenčních příkazů uvnitř procesu umožnují konstrukty známé i z ostatních programovacích jazyků – podmínky a cykly. Sekvenční příkazy umožňují užít dvou podmiňujících příkazů – příkazu **if** a **case**. Jejich význam je obdobný jako u podmíněného a výběrového přiřazení.

Příkaz **if**

Příkaz **if** zapisujeme:

```
[návěští :] if podmínka_0 then
       [sekvenční_příkazy_0]
[elsif podmínka_1 then
       [sekvenční_příkazy_1]]
…
[elsif podmínka_N then
       [sekvenční_příkazy_N]]
[else
       [sekvenční_příkazy_E]]
end if [návěští];
```

Podmínka **if** je vyhodnocována prioritně. Je-li vyhodnocena booleanovská podmínka_0 jako pravdivá, pak jsou vykonány sekvenční příkazy 0 a příkaz je ukončen. Pokud ne, je testována podmínka 1 v bloku **elsif**, vykonávány příkazy 1, atd. Pokud není splněna žádná z podmínek jsou vykonány sekvenční příkazy uvedené za **else**. Stejně jako v případě paralelního příkazu podmíněného přiřazení není **else** nutné uvádět, ale jeho vynechání může vést k vytváření klopných obvodů. Z popisu vyplívá, že větve příkazu **if** mají prioritní charakter – záleží na pořadí jejich uvedení.

Příkaz **case**

Příkaz **case** je obdobou výběrového přiřazení:

```
[návěští :] case výraz_výběru is
[when volba00 | volba01 =>
       sekvenční_příkazy0;]
[when volba10 | volba11 =>
       sekvenční_příkazy1;]
[when volbaN0 | volbaNN =>
       sekvenční_příkazyN;]
when others =>
       sekvenční_příkazyO;
end case [návěští];
```

Po nepovinném návěští uvádíme klíčové slovo **case** doplněné o výraz výběru. Datový typ musí být jednoznačně určitelný (toto dobře splňuje např. signál). Výraz musí být buď skalární a základního datového typu nebo se může využít jednorozměrné pole. Je-li využito pole, pak musí být založeno na znacích (to platí např. pro **std_logic_vector**, ale ne pro pole **integer**).

Na základě hodnoty výrazu jsou vykonávány jednotlivé větve příkazu. Větve jsou specifikovány volbami výběru uvedené klíčovým slovem **when**. Je-li splněna podmínka rovnosti

výrazu výběru a volby výběru, pak jsou vykonány příslušné sekvenční příkazy uvedené za znaky =>. Jednotlivé volby mohou být pouze konstanty a ty mohou být odděleny svislicí | ve smyslu logického nebo. U voleb nejsou připouštěny redundance – podmínka musí být jednoznačně přiřaditelná k dané větvi.

VHDL 2008 umožňuje využití nespecifikovaných bitů ve výrazech volby. Příkaz case se potom může uvádět v syntaxi **case**? výraz_výběru a ve volbách můžeme využít znaku pomlčky jako tzv. don't care hodnotu. Např. volbu "000"|"100" lze nahradit volbou "-00".

Nevyhovuje-li výraz volbě žádné větve, je příkaz ukončen, nebo jsou vykonány sekvenční příkazy bloku **others**. Větev **others** nemusí být specifikována, pokud jsou vyčerpány všechny volby. Pokud chceme uvést, že se v dané větvi nemá vykonávat žádný příkaz použijeme klíčové slovo **null**. Větve příkazu **case** nemají prioritu – nezáleží na pořadí jejich uvedení.

Syntézní nástroje z výrazů s podmínkou **if** generují na multiplexorech založené struktury, z příkazů **case** jsou generovány dekodéry. Příkazy se chovají analogicky ke svým paralelním protějškům. Při užití určitého druhu podmínky v příkazu **if** lze popisovat i sekvenční logiku, ale to si necháme na později. Multiplexor a dekodér z Př. 68 a Př. 69 lze pomocí procesu popsat následovně:

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

entity multiplexor is
    port(
        a, b, sel : in  std_logic;
        q         : out std_logic
    );
end entity multiplexor;

architecture RTL of multiplexor is
begin
    mux: process(a,b,sel)
    begin
        if sel = '0' then
            q <= a;
        else
            q <= b;
        end if;
    end process mux;

end architecture RTL;
```

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

entity decoder1of4 is
    port(
        BCD : in  std_logic_vector(1 downto 0);
        q   : out std_logic_vector(3 downto 0)
    );
end entity decoder1of4;

architecture RTL of decoder1of4 is
begin
    dec: process(BCD)
    begin
        case BCD is
            when "00"   => q <= "0001";
            when "01"   => q <= "0010";
            when "10"   => q <= "0100";
            when others => q <= "1000";
        end case;
    end process dec;
end architecture RTL;
```

12.4.4 Smyčky

V těle procesu lze pomocí příkazu **loop** vytvářet části opakujícího se kódu – smyčky. Smyčku je možné řídit dvěma způsoby – pomocí iteračního schéma **for** nebo **while**. Obecná syntax sekvenčního příkazu **loop** je následující

```
[návěští :] [iterační_schéma] loop
       sekvenční_příkazy;
```

```
end loop [návěští];
```

Není-li iterační schéma smyčky uvedeno, pak je smyčka prováděna donekonečna, pokud není uveden příkaz **exit**, který smyčku ukončí nebo **next**, který spustí další iteraci. Příkaz **next** může ukončit smyčku pouze v případě vnořené smyčky:

```
[návěští :] exit [návěští_smyčky] [when podmínka];
[návěští :] next [návěští_smyčky] [when podmínka];
```

Parametr návěští smyčky u příkazů **exit** a **next** určuje, ke které smyčce v případě vnořených se příkaz vztahuje. Nepovinná podmínka příkazy spustí, pokud je vyhodnocena jako *true*.

Iterační schéma **for**

Iterační schéma **for** může deklarovat implicitní proměnnou řízení cyklu. Do vytvořené proměnné není možné zapisovat, existuje pouze uvnitř cyklu a jsou ji postupně přiřazovány hodnoty ze specifikovaného diskrétního **rozsahu**. Vytvořená proměnná má datový typ bázového typu rozsahu. Pro syntézu typicky platí, že **rozsah** musí být konstantní.

Syntax iteračního schéma **for** je:

```
for řídící_proměnná in rozsah
```

Schéma **for** si demonstrujme na konstrukci obvodu, který bude realizovat funkci N-bitového hradla XOR.

Př. 76 : N bitové hradlo XOR pomocí for loop

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

entity forloop is
    generic(
        N : integer := 5
    );
    port(
        a : in  std_logic_vector(N - 1 downto 0);
        q   : out std_logic
    );
end entity forloop;

architecture RTL of forloop is

begin

    xor_gate_N: process(a)
        variable res : std_logic := '0';
    begin
        res := '0';
        xor_loop: for i in a'range loop
            res := res xor a(i);
        end loop;
```

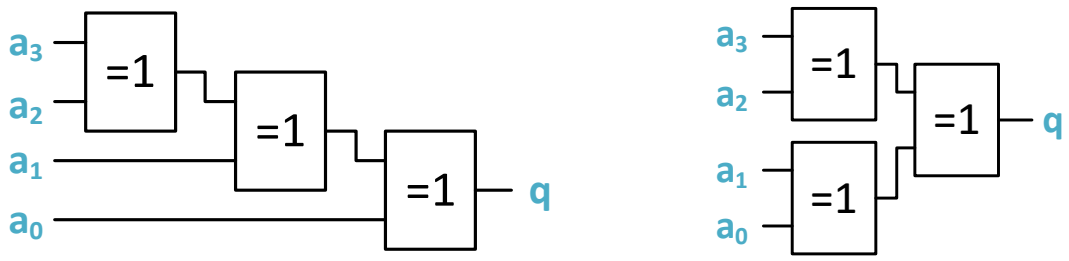
```
        q <= res;
    end process;

end architecture RTL;
```

V cyklu xor_loop jsme deklarovali proměnnou **i** s jejíž pomocí indexujeme jednotlivé prvky vstupního vektoru **a.** Rozsah, ve kterém bude **i** iterováno jsme specifikovali atributem 'range signálu **a** – tím splňujeme podmínku konstantního rozsahu pro účely syntézy. Prvky v cyklu přičítáme (xor) do proměnné **res**, kterou jsme před spuštěním cyklu inicializovali na nulovou hodnotu. Inicializace je nezbytná. Persistentní charakter proměnné by způsobil, že by byla v proměnné zachována poslední hodnota výpočtu (tj. přiřazená při posledním spuštění procesu). Výsledek je přiřazen do výstupního signálu **q**.

K příkladu je nutné uvést několik poznámek. Pro jazyk VHDL ve specifikacích starších než verze 2008 se jedná o nejelegantnější způsob, jak generické N-vstupé hradlo realizovat pomocí behaviorálního popisu! Ve VHDL2008 by stačil zápis:

```
q <= xor(a);
```

Po syntéze nejspíš nezískáme N-vstupé hradlo, ale kaskádně zapojená dvouvstupá hradla xor, viz Obr. 101. Ani kaskádní struktura ale neodpovídá optimálnímu zapojení, které je na stejném obrázku vpravo. Pro vyšší hodnoty N bude rozdíl významnější v poměru lineární funkce vs. logaritmus N. Syntézu ale vinit nelze – vyrobila přesně to, co bylo popsáno. Na závěr uveďme, že N-vstupé hradlo xor bude pravděpodobně vyrobeno při následné optimalizaci návrhu v procesu mapování.



Obr. 101 : Výsledek syntézy (vlevo) vs optimální struktura Př. 76

Iterační schéma **while**

Iterační schéma **while** je řízeno pouze podmínkou, která musí nabývat hodnoty *true*, pokud se má smyčka opakovat. Podmínka musí být v čase syntézy splnitelná, jinak je využití cyklu vyhodnoceno jako chyba.

```
while podmínka
```

Př. 77 demonstruje možné využití cyklu **while** pro konstrukci obvodu prioritního enkodéru – obvodu, který detekuje pozici nejvíce významné jedničky v N-bitovém slově. Přiřazení do výstupní proměnné q si prozatím nebudeme všímat, vysvětlena bude v následujících kapitolách.

Pozici nejvíce významné jedničky ve vstupním vektoru **a** je ukládána do proměnné **pos**. Ta je inicializována na hodnotu atributu '**high** (MSb) vektoru **a**. V cyklu **while** testujeme přítomnost '1' od nejvyššího bitu, v případě nalezení je cyklus ukončen. S každou iterací je snížena hodnota **pos**. Dosáhne-li hodnoty -1, pak je cyklus ukončen, protože nebyla splněna podmínka řízení cyklu. Hodnota **pos** je převedena na výstup **q**.

Př. 77 Prioritní enkodér pomocí while

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

entity whileloop is
    generic(
        N     : integer := 16;
        log2N : integer := 5
    );
    port(
        a : in  std_logic_vector(N - 1 downto 0);
        q : out std_logic_vector(log2N - 1 downto 0)
    );
end entity whileloop;

architecture RTL of whileloop is
begin

    prio_enc : process(a)
        variable pos : integer;
    begin
        pos := a'high;
        pe_loop : while i > 0 loop
            exit pe_loop when a(pos) = '1';
            pos := pos - 1;
        end loop;
        q <= std_logic_vector(to_unsigned(pos, q'length));
    end process;

end architecture RTL;
```

12.4.5 POPIS SEKVENČNÍCH OBVODŮ, PŘÍKAZ WAIT

Pomocí paralelního příkazu **process** lze popisovat i sekvenční obvody. K tomu může využít sekvenční podmínky **if** u které vynecháme větev **else** a do výrazu podmínky uvedeme hodinový signál. Chceme-li popsat hladinový klopný obvod typu D, pak postupujeme následovně:

Př. 78 : D latch pomocí procesu

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

entity DLatch is
    port(
        clk : in  std_logic;
        D   : in  std_logic;
        Q   : out std_logic
    );
end entity DLatch;

architecture Behavioral of DLatch is

    signal latch : std_logic;

begin

    process(clk, D)
    begin
        if clk = '1' then
            latch <= D;
        end if;
    end process;

    Q <= latch;

end architecture Behavioral;
```

Do citlivostního seznamu procesu musíme uvést hodinový signál clk i datový signál D. Při aktivní hladině obvodu chceme, aby výstup kopíroval změny na vstup. Přiřazení do Q můžeme provést již uvnitř procesu (Q <= D), ale zamezíme tím možnosti hodnotu výstupu klopného obvodu číst uvnitř architektury.

Popis hranového klopného obvodu je o něco komplikovanější. Do citlivostního seznamu uvádíme pouze hodinový signál, protože chceme, aby k přiřazení D na Q došlo pouze při žádané změně řídícího signálu. Podmínku uvnitř těla procesu musíme doplnit o logický součin s atributem clk'event, který nabývá hodnoty *true* při změně signálu:

Př. 79 : Hranový klopný obvod typu D

```
library ieee;
use ieee.std_logic_1164.all;
entity DFF is
    port(
        clk : in  std_logic;
        D   : in  std_logic;
        Q   : out std_logic
    );
end entity DFF;
architecture Behavioral of DFF is
    signal FF : std_logic;
begin
    process(clk)
    begin
        if clk = '1' and clk'event then
            FF <= D;
        end if;
    end process;
    Q <= FF;
end architecture Behavioral;
```

Podmínku clk=’1’ and clk’event náběžné hrany a clk=’0’ and clk’event sestupné hrany můžeme nahradit funkcemi rising_edge(signál) a falling_edge(signál) z knihovny IEEE1164, které plní stejný účel. Hranový obvod doplněný o synchronní a asynchronní reset popisují následující příklady. Citlivostní seznam procesu hranového klopného obvodu s asynchronním resetem musí být doplněn o signál resetu. U obvodu se synchronním resetem tomu tak není.

Př. 80 : Hranové klopné obvody D se synchronním a asynchronním resetem

```
library ieee;
use ieee.std_logic_1164.all;
entity DFFSR is
    port(
        clk : in  std_logic;
        rst : in  std_logic;
        D   : in  std_logic;
        Q   : out std_logic
    );
end entity DFFSR;
architecture Behavioral of DFFSR is
    signal FF : std_logic;
begin

process(clk)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                FF <= '0';
            else
                FF <= D;
            end if;
        end if;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity DFFAR is
    port(
        clk : in  std_logic;
        rst : in  std_logic;
        D   : in  std_logic;
        Q   : out std_logic
    );
end entity DFFAR;
architecture Behavioral of DFFAR is
    signal FF : std_logic;
begin

process(clk, rst)
    begin
        if rst = '1' then
            FF <= '0';
        elsif falling_edge(clk) then
            FF <= D;
        end if;
    end process;
```

```
    end process;
    Q <= FF;
end architecture Behavioral;
```

```
    Q <= FF;

end architecture Behavioral;
```

K popisu sekvenčního obvodu jsme využili vlastnost procesu spouštět se při změnách signálů uvedených v citlivostním seznamu. K popisu hranového klopného obvodu by šlo využít i sekvenčního příkazu **wait**. V příkladu můžeme užít exkluzivně jen první, nebo druhou variantu:

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

entity DFF is
    port(
        clk : in  std_logic;
        D   : in  std_logic;
        Q   : out std_logic
    );
end entity DFF;

architecture Behavioral of DFF is

    signal FF : std_logic;

begin

    process
    begin
        -- 1. způsob
        wait on clk;
        if clk = '1' then
            FF <= D;
        end if;

        -- 2. způsob
        wait until rising_edge(clk);
        FF <= D;
    end process;

    Q <= FF;

end architecture Behavioral;
```

Příkaz **wait** je především simulačního charakteru a pro popisy syntetizovatelných obvodů se využívá zřídka. Je-li příkaz v procesu uveden, pak proces nesmí obsahovat seznam citlivostních proměnných. Syntaxe příkazu má čtyři formy:

```
návěští: wait [for časový_výraz] [on signál_0[, signál_n]] [until podmínka];
```

- Není-li uvedena žádná z klauzulí **for**, **on** nebo **until**, pak po spuštění příkazu dojde k úplnému zastavení vykonávání procesu.

- Klauzule **for** pozastaví proces na uvedený čas
- Klauzule **on** obnoví vykonávání sekvenčních procesů, dojde-li ke změně jednoho z uvedených signálů – má obdobných charakter, jako citlivostní seznamu procesu.
- Klauzule **until** pozastaví vykonávání dalších příkazů, dokud nebude platná uvedená podmínka.

Příkaz **wait** využíváme nejčastěji v simulaci – čekáme-li na změnu nějakého signálu, ustálení výstupu, odpověď simulovaného obvodu atp. Pro syntézní účely není možné uvádět variantu příkazu s klauzulí **for**, jelikož číslicové obvody, které z VHDL vznikají neobsahují žádný prostředek na implicitní měření času, narozdíl od simulátorů. Uveďme jednoduchý příklad použití příkazu pro generování „hodinového" signálu s konstantní periodou:

Př. 81 : Použití příkazu wait

```
    constant CLOCK_PERIOD : time      := 10 ns;
    signal clock          : std_logic := '0';

begin

    clk_gen : process
    begin
        wait for CLOCK_PERIOD / 2;
        clock <= not clock;
    end process;
```

V příkladu výše je uveden **process** clk_gen. Ten je spuštěn ihned po zahájení simulace. Příkaz **wait** uvnitř procesu nastaví simulaci pozastavení vykonávání procesu po dobu CLOCK_PERIOD/2, tj. na 5 ns. Po uplynutí žádaného času je vykonávání příkazů procesu obnoveno. Je provedena inverze hodin, proces je ukončen, dojde k přiřazení nové hodnoty do signálu clock a jelikož proces neobsahuje citlivostní seznam, tak je spuštěn znovu. Celá situace se bude cyklicky opakovat do ukončení simulace. Signál clock musí být deklarován s inicializační hodnotou – pokud bychom ji neuvedli, pak by výsledkem výrazu **not** clock byla vždy hodnota nedefinováno 'U'.

12.4.6 SIMULAČNÍ PŘÍKAZY

Na závěr jsme si nechali příkazy, které se používají především v simulačních vhdl souborech. Jsou to příkazy **assert** a **report**.

Assert slouží k vygenerování chybová hlášky, není-li splněna podmínka:

```
[návěští:] assert podmínka [report výraz] [severity výraz];
```

Podmínkou může být libovolná booleovská podmínka. Nepovinný report musí být typu string ("řetězec znaků"), **severity** je enumerátor implicitního typu SEVERITY_LEVEL a nabývá hodnot NOTE, WARNING, ERROR a FAILURE; výchozí hodnota **severity** je ERROR. Aby **assert** vyvolal chybu a případně vypsal hlášení musí nabývat podmínka hodnoty false. **Assert** je paralelní příkaz, ale může být použit i v sekvenčním bloku.

Druhým příkazem je příkaz report, který je vykonán bezpodmínečně a do konzole simulátoru vypíše textový řetězec, např:

```
    process(a, shift, left, arith)
        variable i : integer;
    begin
        …
         jméno_hlášení: report "číslo i =" & integer'image(i);
        …
    end process;
```

Součástí příkazu může být i blok **severity**, se stejným významem jako u příkazu **assert**. Vynecháme-li ho, je hodnota „vážnosti“ NOTE. Příkaz report je sekvenčním příkazem.

12.5 PODPROGRAMY A KNIHOVNY

Nedílnou součástí každého programovacího jazyka je volání podprogramů. VHDL rozlišuje dva typy podprogramů – **funkce** a **procedury**. Argumenty **funkcí** jsou ve VHDL předávány hodnotou, v případě **procedur** lze argumenty předávat i odkazem. **Funkce** vždy vrací návratovou hodnotu, **procedura** nevrací nic. Syntaxe zápisu obou podprogramů je podobná. Skládá se z hlavičky podprogramu a těla podprogramu. Hlavička funkce má následující zápis:

```
[impure|pure] function jméno_funkce[
([třída] parametr00 [,parametr01,…, parametr0N] : typ
[;[třída] parametrM1[, parametrM2, parametrMN] : typ])]
return návratový_typ;
```

hlavička procedury pak:

```
procedure jméno_procedury[
([třída] parametr00 [,parametr01,…, parametr0N] : [inout|in|out] typ;
<[[třída] parametrM1[, parametrM2, parametrMN] : [inout|in|out] typ]);>];
```

Tělo funkce má syntax:

```
[impure|pure] function jméno_funkce[
([třída] parametr00 [,parametr01,…, parametr0N] : typ
[;[třída] parametrM1[, parametrM2, parametrMN] : typ])]
return návratový_typ is
    [deklarační_část;]
begin
    [sekvenční_příkazy]
    return návratová_hodnota;
end function jméno_funkce;
```

Tělo procedury:

```
procedure jméno_procedury[
([třída] parametr00 [,parametr01,…, parametr0N] : [inout|in|out] typ
[;[třída] parametrM1[, parametrM2, parametrMN] : [inout|in|out] typ])]
is
    [deklarační_část;]
begin
    [sekvenční_příkazy]
end procedure jméno_procedury;
```

Hlavička podprogramu je nepovinná, tělo podprogramu povinné je. Hlavičku uvádíme, pokud chceme podprogram volat ještě před tím, než jsme napsali jeho tělo – např. máme-li dvě funkce, které se vzájemně odkazují. Hlavička a hlavička těla funkce musí být samozřejmě stejná.

Seznam předávaných argumentů je středníkem oddělený seznam **konstant**, **proměnných**, **signálů** a **souborů** (specifikátor **třídy**) se známými datovými **typy**. Za posledním záznamem středník nepíšeme. U procedur je možné volit navíc **mód** argumentu – **in**, **out**, **inout** – podobně jako u deklarace portů. Argumenty s módem **in** se chovají jako konstanty, argumenty v módu **out** jako návratové parametry, které nelze v těle procedury číst. Argumenty **inout** lze v těle **procedury** jak číst, tak upravovat. U **funkcí** je mód **in** implicitní a není ho možné měnit. Není-li **třída** argumentu specifikovaná, pak je automaticky volena **proměnná**.

V deklarační části procedury deklarujeme **proměnné**, **soubory**, **podprogramy**, **typy** a jiné – podobně, jako tomu je u deklarační části **procesu**. Elementy deklarované v rámci podprogramu jsou platné pouze v něm. Na rozdíl od proměnných v procesu nejsou proměnné ve funkci persistentní – jejich hodnota je inicializována při každém volání funkce.

Podprogram nesmí číst nebo zapisovat do objektů, které nejsou deklarovány v těle podprogramu, nebo nejsou součástí argumentů podprogramu. Pokud zapisujeme do **signálu**, který není součástí předaných argumentů nebo deklarovaných objektů, pak musíme deklarovat tzv. nečistou funkci. Toto provedeme zápisem klíčového slova **impure** před deklaraci funkce.

Implicitně jsou všechny funkce čisté (**pure**) a modifikátor před deklaraci funkce psát nemusíme. Doplňme, že v mnoha syntézních nástrojích nejsou nečisté funkce syntetizovatelné.

Mezi klíčovými slovy **begin** a **end** se nachází tělo podprogramu, do nějž umísťujeme sekvenční příkazy, které jsou vykonávány stejně jako příkazy v těle procesu. Tělo funkce musí obsahovat příkaz **return** s parametrem návratové hodnoty. Po zavolání tohoto příkazu je vykonávání **funkce** ukončeno. Volání **procedury** je ukončeno vykonáním příkazu **return** bez parametru nebo vykonáním všech příkazů **procedury**.

V těle podprogramu nesmí být volán příkaz **wait** a nesmí být přistupováno k atributům 'STABLE, 'QUIET, 'TRANSACTION a 'DELAYED. U **procedur** si musíme dávat pozor na přiřazení do signálu. Proceduru, která mění signál je možné deklarovat a volat pouze tak, aby nevytvářela vícenásobné budiče.

Všechny podprogramy lze volat rekurzivně a je možné je přetěžovat. Název **funkce** může být i již existující operátor, ten umisťujeme do uvozovek. Tímto způsobem lze například upravit funkci operátorů pro uživatelem deklarované datové typy. Podprogramy si představíme na dvojici příkladů. První realizuje dvojkový logaritmus z celého kladného nenulového čísla (**positive**).

Př. 82 : Dvojkový logaritmus celého čísla pomocí funkce

```
function log2(x : positive) return natural is
variable i : natural;
    begin
    i := 0;
    while (2**i < x) and i < 31 loop
        i := i + 1;
    end loop;
    return i;
end function;
```

Druhá ukázka představuje užití procedury, která existující signál typu **std_logic_vector** naplňuje jiným signálem zprava nebo zleva (from_left) a doplňuje ho o uživatelem definovanou konstantu (fill). Součástí příkladu je i možná exkluzivní aplikace procedury – přiřazení kratšího vektoru a doplnění zbytku míst. Procedura používá funkci resize z knihovny IEEE numeric_std (viz 12.6.1).

Př. 83 : Procedura doplnění vektoru zleva nebo zprava.

```
architecture
    procedure mkvec(
        signal a  : inout std_logic_vector;
        signal b  : std_logic_vector;
        fill      : std_logic;
        from_left : boolean
    ) is
        variable f : unsigned(0 downto 0) := (others => fill);
    begin
        if from_left then
            a <= b & std_logic_vector(resize(f, a'length - b'length));
        else
            a <= std_logic_vector(resize(f, a'length - b'length)) & b;
        end if;
```

```
    end procedure;

    signal a : std_logic_vector(4 downto 0);
    signal b : std_logic_vector(1 downto 0);
    signal c : std_logic_vector(1 downto 0);

begin
    -- Naplní a výrazem'0' & … & '0' & b & c
    mkvec(a, b & c, '0', true);
    -- nebo naplní a výrazem c & b & '1' & '1'…
    mkvec(a, c & b, '1', false);
```

Procedura v příkladu skončí chybou simulace nebo syntézy, pokud bude délka argumentu b vetší nebo rovna délce argumentu a.

12.5.1 PLATNOSTI, BALÍČKY

Všechny deklarované objekty (typy, podprogramy, signály, konstanty, aj.) jsou platné pouze v regionech, v jejichž deklaračních částech byly deklarovány. Deklarujeme-li typ v podprogramu, nelze se na něj odkazovat v jiném podprogramu; podprogram deklarovaný v architektuře nějaké entity nelze volat v instancované komponentě, atp. Chceme-li deklarovat tyto objekty globálně, pak musíme využít knihovny, jež se ve VHDL nazývají **package** - balíček.

Balíčky se typicky umisťují do samostatných souborů (s příponou **vhd**). Jejich struktura se skládá z povinné deklarace balíčku a nepovinného těla balíčku.

```
package jméno_package is
    deklarační_část;
end package [jméno_package];

[package body jméno_package is
   deklarační_část;
end package body [jméno_package];]
```

V deklarační části deklarace balíčku umisťujeme deklarace typů, signálů, hlaviček podprogramů, konstant a jiných. V této části nesmíme psát těla podprogramů. V nepovinné deklarační části těla balíčku můžeme deklarovat typy, konstanty, těla podprogramů a další. Obsahuje-li náš balíček deklaraci podprogramu, pak je tělo balíčku povinné.

Použití balíčku uvádíme zápisem před entitu ve formátu:

```
library jméno_knihovny;
use jméno_knihovny.balicek.(element | all);
```

Jméno knihovny je typicky adresář, ve kterém se nachází balíčky od jednoho dodavatele, nebo významově podobné balíčky. Je-li soubor s balíčkem umístěn ve stejném adresáři jako soubor, který na něj odkazuje, pak je jméno_knihovny implicitně nastaveno na work; **library** work není zapotřebí uvádět. Výjimkou je případ, kdy by byly deklarované objekty ve dvou různých balíčcích stejně pojmenovány.

Klauzule **use** vyvolává použití balíčku z knihovny pro danou entitu (pouze pro první **entitu** za **use**). Za název balíčku můžeme za tečku uvést buď konkrétní deklarovaný objekt nebo klíčové slovo **all**, které zpřístupní všechny deklarace v balíčku.

Jednoduchý příklad na vytvoření balíčku a jeho následné použití při deklaraci dvojice entit nalezneme níže.

Př. 84 : Deklarace a využití balíčku

```
-- deklarace balíčku (pouze hlavička – tělo nemusíme deklarovat)
package balíček is
    constant LENGTH : integer := 10;
    constant WIDTH  : integer := 100;
end package;

-- deklarace entity 1
library ieee;
use ieee.std_logic_1164.all;
use work.balíček.all;

entity jméno_entity is
    port(
        a : in std_logic_vector(LENGTH - 1 downto 0)
    );
end entity;

-- deklarace entity 2
use work.balíček.WIDTH;

entity jméno_další_entity is
    port(
        a : in bit_vector(WIDTH - 1 downto 0)
    );
end entity;
```

Všimněme si, že v případě druhé entity jsme z balíčku dovolili využít pouze konstantu WIDTH. Konstanta LENGTH je nepřístupná, stejně jako datový typ std_logic_vector.

12.6 ARITMETIKA, NUMERIC_STD

V předchozím textu jsme si představili většinu základních konstrukcí jazyka VHDL. Pomocí nich bychom mohli popsat valnou většinu číslicových obvodů. Výjimku by tvořili obvody realizující aritmetické operace, pokud bychom jejich funkce nenahradili jejich strukturním popisem. Tvorba takových obvodů by byl zdlouhavý proces, který by znamenal vytvoření generických entit realizujících různé aritmetické operace pro každý myslitelný datový typ. Vytváření jakýchkoliv složitějších obvodů by opět vedlo na strukturní popis a bylo by zdlouhavé a nepraktické.

V kapitole 12.3.1 jsme si představili aritmetické operace a zdůraznili jsme, že jsou podporované pouze u skalárních bázových datových typů; tedy prakticky jen pro typy **integer** nebo **real** (a odvozené).

Typů s plovoucí řádovou čárkou se snažíme při popisu číslicových obvodů vyvarovat, jelikož mají tendenci vést k složitému, velkému a pomalému hardwaru, pokud nám vůbec syntéza nějaký vyrobí. V simulaci se jim nebráníme.

Použití typu **integer** je pro velikost a rychlost výsledného obvodu výhodnější, ale i pro integer platí celá řada nevýhod:

- Doporučená velikost **integer** 32 bitů limituje velikosti proměnných a čítačů. V některých případech jsou struktury zbytečně velké v jiných nedostatečně velké.
- **Integer** je kódován pouze ve dvojkovém doplňku – polovina rozsahu pro bezznaménkové typy by přišla vniveč.
- **Integer** neumožňuje simulovat jiné stavy než log. 0 nebo 1
- Neexistují oficiální převodní funkce **std_logic_vector** ↔ **integer** – museli bychom si je napsat

Tyto a další potíže vedly k vytvoření standardu IEEE 1076.3, který zosobňuje knihovna ieee.numeric_std (a tlustý manuál). Knihovna deklaruje dva nové datové typy:

```
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

Datový typ **signed** je určen pro popis dat kódovaných v dvojkovém doplňku, **unsigned** pro bezznaménková data. Od **std_logic_vector**, který je deklarován jako …

```
type std_logic_vector is array (natural range <>) of std_logic;
```

… se liší v tom, že jsou pro ně definovány i téměř všechny aritmetické operace, relační operátory, logické operátory a nejrůznější konverzní funkce.

Historické okénko.
O palčivém problému s aritmetikou se vědělo dávno před specifikací standardu IEEE 1076.3 (1997) a protože je průmysl rychlejší než akční skupiny standardizačních organizací, tak v roce 1990 představila firma Synopsis knihovny ieee.std_logic_arith a ieee.std_logic_(unsigned/signed). Tyto knihovny umožňovaly v simulaci i syntéze nástrojů od Synopsis používat aritmetické operace pro datový typ std_logic_vector. Návrhář si mohl vybrat, zda jsou signály std_logic_vector v jeho entitě kódovány jako znaménkové (ieee.std_logic_signed) nebo bezznaménkové (ieee.std_logic_unsigned) typy. Knihovny následně přejaly i ostatní firmy a staly se de-facto standardem pro realizaci aritmetických operací s datovým typem std_logic_vector. Využívání knihoven překvapivě nezastavilo ani uvedení standardu – využívaly se minimálně další dvě dekády po uvedení ieee.numeric_std. Teprve v posledním desetiletí je standardní knihovna numeric_std, která je vymyšlena o něco lépe, nahradila.

12.6.1 Přetypování a konverze

První úkol, kterému budeme čelit je změna datového typu mezi **std_logic_vector**, pomocí kterého budeme často komunikovat s okolním světem a typy (**un**)**signed**, které využijeme pro popis vnitřní funkce obvodu. Pro popis rozhraní mezi naším obvodem a světem vůkol je použití

std_logic_vector v mnohém výhodnější. Popisujeme-li obecnou datovou sběrnici, nevíme, jaká data po ní proudí; obdobně tomu tak je v napojení registrů procesoru na aritmetickologickou jednotku – jednou bity uložené v registrech představují celá čísla, jindy znaky nebo desetinná čísla, atp. Jejich význam je závislý na kontextu operací, které s nimi provádíme. Pro popisy obvodů, kde je datový typ jasný budeme užívat (un)signed.

Vzájemné změny datových typů provádíme přetypováním. Pro přetypování do **unsigned** a **signed** využijeme výrazů:

```
    signál_unsigned <= unsigned(výraz);
    signál_signed   <= signed(výraz);
```

Platí, že signál na levé straně přiřazení musí být stejného typu jako signál na straně pravé. Výraz může být typu **signed** nebo **std_logic_vector** v případě konverze do **unsigned**, v případě konverze do **signed** pak typů **unsigned** nebo **std_logic_vector**. Přetypování do stejných typů je samozřejmě také možné, ale nadbytečné. Při přetypování mezi **signed** a **unsigned** si musíme dávat pozor – převádí se binární obsah, ne jeho význam.

Přetypování do **std_logic_vector** se provádí obdobně:

```
    signál_std_logic_vector <= std_logic_vector(výraz_(un)signed);
```

Výraz může být libovolný výraz typu **signed** nebo **unsigned** o stejné délce jako signál na levé straně.

Převody do numerických datových typů řeší funkce konverzní. Jelikož (**un**)**signed** popisuje celočíselné hodnoty, tak lze konvertovat pouze do a z celočíselných datových typů. Při konverzích musíme mít na zřeteli omezený rozsah a kódování datového typu **integer**.

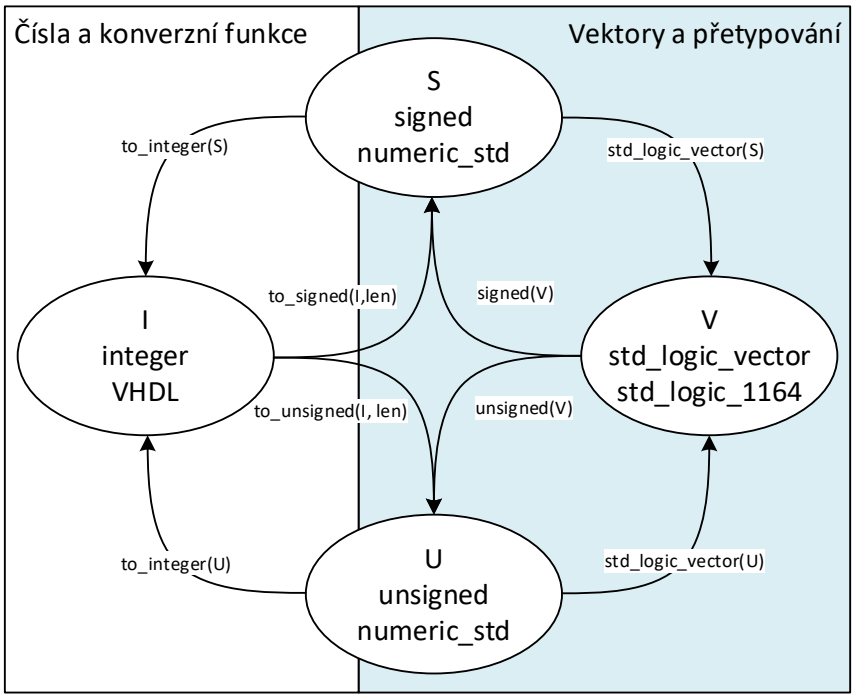
Pro konverzi do **integer** využíváme funkce to_integer() s parametrem typu unsigned nebo signed:

```
    signál_integer <= to_integer((un)signed_výraz);
```

Pro konverzi integer do (**un**)**signed** využíváme funkce to_(un)signed(hodnota, délka):

```
    signál_signed   <= to_signed(výraz_integer, délka);
    signál_unsigned <= to_unsigned(výraz_integer, délka);
```

Při zadávání délky můžeme využít atribut *'length* signálu na levé straně konverze. Schéma přetypování a konverzí vystihuje obrázek



Obr. 102 Schéma konverzí a přetypování

12.6.2 Operace

Datové typy (**un**)**signed** podporují všechny logické operace, relační operace a některé aritmetické operace. Podporovány jsou:

Tab. 55 : Aritmetické operátory (un)signed

Třída	Operátory	Význam
Násobení	*	násobení
	/	dělení
	mod	modulo
	rem	zbytek po dělení
Znaménko	-	záporná hodnota
Sčítání	+	součet
	-	rozdíl
	&	sloučení vektorů
Posuv	shift_left(s, c)	posuv vlevo
	shift_right(s, c)	posuv vpravo
	rotate_left(s, c)	rotace vlevo
	rotate_right(s, c)	rotace vpravo
Ostatní	abs	absolutní hodnota
	resize(s, l)	změna bitové délky
	std_match(s, s)	porovnání s don't care
	to_01(s, v)	převod do '0'/'1' logiky

, u funkcí **s** znamená argument typu signál, **c** počet, **l** délku a **v** hodnotu.

Operace si popíšeme v pořadí uvedeném v tabulce Tab. 55:

- Výsledkem operace násobení dvou **unsigned** signálů od délce M a N je **unsigned** signál o délce M+N. Násobit lze i (**un**)**signed** s celočíselnými datovými typy. Ty jsou před násobením konvertovány do stejného datového typu a délky, jakou má druhý operand. **Unsigned** lze násobit pouze kladnými čísly(**natural**), **signed** i zápornými (**integer**). Násobit **unsigned** a **signed** vzájemně nelze.
- Výsledkem operace dělení je datový typ o délce dělence. **Unsigned** lze dělit pouze dělitelem typu **unsigned** nebo **natural**. **Signed** lze dělit děliteli **signed** a **integer**.
- Unární operátor znaménka '-' je podporován pouze pro datový typ **signed**.
- Výsledkem součtu M a N bitových (**un**)**signed** je signál délky max(M, N). Sčítat lze pouze stejné datové typy, **unsigned** lze sčítat navíc i s **natural**, **signed** s **integer**. Stejná pravidla platí pro odečítání.
- VHDL operace sll, slr, sal, sar, rol a ror nahrazují funkce shift_left/right(s, c) a rotate_left/right(s, c). U rotací se význam nemění a je podporován u obou typů stejně. Význam logického a aritmetického posuvu je dán typem operandu – jedná-li se o **unsigned**, pak jsou posuvy vždy logické. Aritmetického posuvu můžeme docílit pouze u typu **signed** a to při posuvu vpravo; v takovém případě jsou hodnoty vlevo doplněny hodnotou bitu nejvíce vlevo původního signálu. V jiných případech jsou z příslušných stran doplňovány '0'. Funkce přijímají kromě argumentu typu (**un**)**signed** parametr o kolik pozic bude posuv/rotace provedena. Tento parametr musí být kladný (**natural**). Původní operace lze používat dál, druhým argumentem může být i **integer**, ale je nutné dát pozor na typová pravidla.
- Absolutní hodnota je podporována pouze u **signed** a vrací **signed**.
- Operace resize(s, l) provede změnu velikosti signálu. Při zmenšování jsou odříznuty bity zleva; u **signed** je nejvýznamnější bit zachován. Při rozšiřování jsou u **unsigned** bity vlevo doplňovány '0', u **signed** hodnotou bitu na pozici nejvíce vlevo původního signálu. Argumenty funkce jsou (**un**)**signed** signál a jeho nová kladná velikost (**natural**).
- std_match(s, s) porovnává dva signály stejného datového typu ((**un**)**signed** a **std_logic_vector**). Vrací true, pokud jsou signály shodné nebo je jeden z prvků signálu '0' a druhý na stejné pozici 'L' nebo je jeden '1' a druhý 'H' nebo je-li hodnota jednoho z prvků signálu don't care '-'.
- to_01(s, v) převádí signál do stejného datového typu nahrazující hodnoty 'L' hodnotou '0' a 'H' hodnotou '1'. Jsou-li v signálu jiné hodnoty, jsou tyto nahrazeny hodnotou druhého argumentu funkce.
- Logické operace jsou podporovány pouze u stejných datových typů. U relačních operátorů lze porovnávat **signed** s **integer** nebo **unsigned** s **natural**.

12.6.3 PŘÍKLADY

Použití datových typů (**un**)**signed** si ukážeme na trojici příkladů. V prvním případě popisujeme cyklický binární čítač s předvolbou, synchronním resetem a možností inkrementu nebo dekrementu.

Př. 85 : Cyklický binární čítač

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

entity counter is
    generic(
        WIDTH : integer := 16
    );
    port(
        clk    : in  std_logic;
        rst    : in  std_logic;
        preset : in  unsigned(WIDTH - 1 downto 0);
        load   : in  std_logic;
        enable : in  std_logic;
        up     : in  std_logic;
        q      : out unsigned(WIDTH - 1 downto 0)
    );
end entity counter;

architecture RTL of counter is

    signal cnt_reg : unsigned(WIDTH - 1 downto 0);

begin

    process(clk)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                cnt_reg <= (others => '0');
            elsif load = '1' then
                cnt_reg <= preset;
            elsif enable = '1' then
                if up = '1' then
                    cnt_reg <= cnt_reg + 1;
                else
                    cnt_reg <= cnt_reg - 1;
                end if;
            end if;
        end if;
    end process;

    q <= cnt_reg;

end architecture RTL;
```

Druhý příklad ukazuje využití operátoru násobení pro konstrukci Multiply-Accumulate jednotky. Ta provádí operaci $MAC = MAC + a * b$, všechny operandy jsou **signed** v uživatelem zvolené délce WIDTH. Jednotka je vybavena detekcí přetečení (viz 8.1.3) a saturací. Povšimněte si použití příkazu **assert**, který zastaví simulaci, jsou-li délky operandů a akumulátoru špatně

deklarovány a agregátů ('0', **others** => '1') a ('1', **others** => '0') při popisu minimální a maximální hodnoty **signed** čísla neznámé délky.

Př. 86 : MAC jednotka

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

entity MAC is
    generic(
        A_WIDTH   : integer := 5;
        B_WIDTH   : integer := 5;
        ACC_WIDTH : integer := 10
    );
    port(
        clk    : in  std_logic;
        rst    : in  std_logic;
        enable : in  std_logic;
        a      : in  signed(A_WIDTH - 1 downto 0);
        b      : in  signed(B_WIDTH - 1 downto 0);
        q      : out signed(ACC_WIDTH - 1 downto 0)
    );
end entity MAC;

architecture RTL of MAC is
    signal acc : signed(ACC_WIDTH - 1 downto 0);
begin

    assert ACC_WIDTH >= A_WIDTH + B_WIDTH report "Insufficient ACC width" severity failure;

    process(clk)
        variable product : signed(A_WIDTH + B_WIDTH - 1 downto 0);
        variable sum     : signed(acc'range);
    begin
        if rising_edge(clk) then
            if rst = '1' then
                acc <= (others => '0');
            elsif enable = '1' then
                product := signed(a) * signed(b);
                sum     := product + acc;
                if (acc(acc'high) = '0' and product(product'high) = '0' and sum(sum'high) = '1') then
                    acc <= ('0', others => '1');
                elsif (acc(acc'high) = '1' and product(product'high) = '1' and sum(sum'high) = '0') then
                    acc <= ('1', others => '0');
                else
                    acc <= sum(acc'range);
                end if;
            end if;
        end if;
    end process;

    q <= acc;

end architecture RTL;
```

Ve třetím příkladu ukážeme využití funkcí shift_left/right, které využijeme ke konstrukci obvodu barrel shifter. Obvod realizuje obecnou operaci posuvu. Uživatel vstupy left volí směr posuvu, arith typ posuvu a vstup shift volí počet míst posuvu.

Př. 87 : Barrel shifter

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

entity barrel_shifter is
    generic(
        C_WIDTH : integer := 32
    );
    port(
        a     : in  std_logic_vector(C_WIDTH - 1 downto 0);
        shift : in  unsigned(integer(ceil(log2(real(C_WIDTH)))) - 1 downto 0);
        left  : in  std_logic;
        arith : in  std_logic;
        q     : out std_logic_vector(C_WIDTH - 1 downto 0)
    );
end entity barrel_shifter;

architecture RTL of barrel_shifter is

begin

    process(a, shift, left, arith)
    begin
        if left = '0' then
            q <= std_logic_vector(shift_left(unsigned(a), to_integer(shift)));
        else
            if arith = '1' then
                q <= std_logic_vector(shift_right(signed(a), to_integer(shift)));
            else
                q <= std_logic_vector(shift_right(unsigned(a), to_integer(shift)));
            end if;
        end if;
    end process;

end architecture RTL;
```

12.7 SOUBORY, TEXTIO

Typ soubor – **file** – je jedním ze základních datových typů VHDL. Umožňuje vytvářet homogenní binární soubory různých datových typů. Jelikož číslicové obvody většinou neobsahují disky, operační systémy a souborové systémy je použití souborů limitováno na simulaci a soubory lze číst a zapisovat pouze v hostitelském počítači.

Základní syntax pro deklaraci souboru je následující:

```
    type file_type is file of data_type;
    file F : file_type;
```

V prvním řádku deklarujeme typ souboru, v druhém „proměnnou“, se kterou budeme pracovat. Jazyk nabízí následující procedury…

Tab. 56 : Procedury přístupu k souborům

procedura	Význam
FILE_OPEN ([Status: **out** FILE_OPEN_STATUS,] **file** F: file_type, External_Name: **in** STRING, Open_Kind: **in** OPEN_FILE_KIND);	Otevírá soubor. Nepovinný výstup Status vrací stav otevření souboru, viz Tab. 57. F je proměnná typu soubor. External_Name je cesta k souboru v hostitelském počítači. Výchozím adresářem je typicky adresář, ve kterém probíhá simulace. Cesta by měla dodržovat konvence psaní cest v hostitelském počítači, nespecifikuje-li to simulátor jinak. Open_Kind volí způsob otevření souboru, viz Tab. 58.
FILE_CLOSE (**file** F: file_type);	Uzavře soubor F.
READ (**file** F: file_type, Value: **out** data_type);	Čte ze souboru F, vrací hodnotu předanou odkazem Value. data_type musí být shodný s data_type z deklarace datového typu souboru.
WRITE (**file** F: file_type, Value: **in** data_type);	Zapisuje do souboru F. Datový typ Value musí být shodný s data_type z deklarace datového typu souboru.

… a funkci ENDFILE (**file** F: file_type), která vrací booleovskou hodnotu *true*, jsme-li na konci souboru F. FILE_OPEN_STATUS a OPEN_FILE_KIND jsou deklarovány jako enumerační datové typy a slouží k popsání stavu otevření souboru a typu přístupu k souboru:

Tab. 57 : FILE_OPEN_STATUS

FILE_OPEN_STATUS	Význam
OPEN_OK	Soubor se podařilo otevřít.
STATUS_ERROR	Soubor je již otevřený.
NAME_ERROR	Soubor nebyl nalezen nebo není přístupný.
MODE_ERROR	Nelze otevřít soubor s daným typem otevření.

Tab. 58 : OPEN_FILE_KIND

OPEN_FILE_KIND	Význam
READ_MODE	Soubor je pouze pro čtení.
WRITE_MODE	Soubor je pouze pro zápis. Obsah existujícího je přepsán.
APPEND_MODE	Soubor je pouze pro zápis, ale je otevřen na svém konci a není přepsán.

Práci s binárními soubory si nejprve ukážeme na příkladu zápisu:

Př. 88 : Zápis binárního std_logic_vector souboru

```
architecture RTL of file_example is
    type slv_file_t is file of std_logic_vector;
    file F : slv_file_t;
begin
    process
        variable status : file_open_status;
    begin
        file_open(status, F, "C:\some_path\file.slv", WRITE_MODE);
        if status = OPEN_OK then
            for i in -4 to 3 loop
                write(F, std_logic_vector(to_signed(i, 3)));
            end loop;
            file_close(F);
        end if;
        wait;
    end process;
end architecture RTL;
```

Deklaraci typu souboru i vytvoření souboru provádíme v deklarační části architektury. Povšimněte si, že v deklaraci slv_file_t využíváme neomezený std_logic_vector; v procesu soubor otevřeme (vytvoříme), zkontrolujeme, zda se operace zdařila a do souboru zapíšeme 8 čísel v intervalu <-4,3> kódovaných jako čísla ve dvojkovém doplňku s délkou 3 bity. Soubor zavřeme a zavoláme příkaz **wait**, abychom zabránili opětovnému spuštění procesu bez citlivostního seznamu. Zobrazíme-li si obsah souboru v libovolném hexa editoru, pak uvidíme následující:

Tab. 59 : Binární obsah souboru z příkladu Př. 88

offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	03	02	02	03	02	03	03	03	02	03	03	03	02	02	02	02
10	02	03	02	03	02	02	03	03								

Jednotlivé bajty odpovídají deklaraci std_logic. Jelikož se jedná o znak, tak „bit“ zabírá jeden bajt. Jeho hodnoty jsou kódovány vzestupně dle toho, jak byly seřazeny v deklaraci std_logic ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'), viz Tab. 49.

Všimněte si, že nikde v souboru není uloženo, jak jsou vektory dlouhé. V druhém příkladu provedeme načtení souboru a hodnoty v něm uložené použijeme pro inicializaci signálu. Při čtení musí být délka specifikována. Inicializace musí být řešena pomocí funkce:

Př. 89 : Inicializace paměti pomocí souboru

```
architecture RTL of example_025_files is

    constant WIDTH : integer := 3;
    constant DEPTH : integer := 8;

    type mem_t is array (DEPTH - 1 downto 0) of std_logic_vector(WIDTH - 1 downto 0);

    function init_mem(init_file : in string) return mem_t is
        type file_slv_t is file of std_logic_vector(WIDTH - 1 downto 0);
        file f          : file_slv_t;
        variable status : file_open_status;
```

```
        variable x      : std_logic_vector(WIDTH - 1 downto 0);
        variable mem    : mem_t := (others => (others => '0'));
    begin
        file_open(status, f, init_file, READ_MODE);
        if status = OPEN_OK then
            for i in mem'range loop
                read(f, x);
                mem(i) := x;
            end loop;
        end if;
        return mem;
    end;

    signal mem : mem_t := init_mem("C:\path\file.bin");

begin
```

Největší nevýhoda binárních souborů je jejich obtížná přenositelnost a škálovatelnost. Kdybychom se rozhodli parametr WIDHT nebo HEIGHT v předchozím příkladě změnit, pak bychom zcela jistě načetli jiné hodnoty, než bylo zamýšleno. Obdobně u příkladu zápisu musíme příjemci sdělit jakým způsobem jsou data kódována a smířit se s tím, že zabírají příliš mnoho místa a ani tak nejsou lidsky čitelná.

V praxi se pro práci se soubory používá standartní balíček textio, který umožňuje vytvářet, číst a formátovaně zapisovat textové soubory a mírně tak vylepšit přenositelnost souborů. Balíček je součástí standardní knihovny STD a deklaruje následující datové typy:

```
    type line is access string;
    type text is file of string;
    type side is (RIGHT, LEFT);
    subtype width is natural;
```

Typ line je ukazatel na řetězec, text je textový soubor, side určuje zarovnání a typ width slouží k určení počtu vypsaných znaků datového typu. Se souborem text se pracuje pomocí řádků ukončených koncem řádku dle konvence operačního systému. K tomu slouží procedury:

```
        procedure readline(file f : text; l : out line);
        procedure writeline(file f : text; l : inout line);
```

S řádky se pracuje pomocí procedur, které jsou deklarovány pro datové typy bit, bit_vector, boolean, character, integer, real, string a time. Chceme-li pracovat se std_logic_vector, pak musíme deklarovat použití balíčku ieee.std_logic_textio, který je s std.textio kompatibilní. Procedury jsou deklarovány pro čtení a zápis:

```
        procedure read(l : inout line; value : out type; good : out boolean);
        procedure read(l : inout line; value : out type);
        procedure write(l : inout line; value : in type; justified : in side :=
RIGHT; field : in width := 0);
```

Balíček dále deklaruje implicitní soubory input a output, které slouží k práci s konzolí. Práci s textovými soubory demonstruje následující příklad. V ukázce sledujeme výstup čítače z Př. 85. Změní-li se výstup čítače, zapisujeme do textového souboru všechny signály, které nás zajímají:

```
    constant WIDTH : integer := 10;
    constant CLK_P : time := 10 ns;
    signal clk, rst, load, enable, up : std_logic := '0';
    signal q, preset : unsigned(WIDTH - 1 downto 0);

begin

    ckgen: process
    begin
        wait for CLK_P / 2;
        clk  <= not clk;
    end process;

    process
        file f : text;
        variable l : line;
        variable opened : boolean := false;
        variable status : file_open_status;
    begin
        wait until q'event;
        if not opened then
            file_open(status, f, "C:\work\file.txt", WRITE_MODE);
            if status /= OPEN_OK then
                wait;
            end if;
            opened := true;
        else
            write(l, now, right, 8);
            write(l, rst, right, 4);
            write(l, load, right, 5);
            write(l, enable, right, 7);
            write(l, up, right, 3);
            write(l, std_logic_vector(preset), right, preset'length + 1);
            write(l, std_logic_vector(q), right, q'length + 1);
            writeline(f,l);
        end if;
    end process;

    counter_inst : entity work.counter
        generic map(
            WIDTH => WIDTH
        )
        port map(
            clk, rst, preset, load, enable, up, q
        );

    tb: process
    begin
        rst <= '1';
        wait for CLK_P;
        rst <= '0';
        enable <= '1';
        up <= '1';
        wait;
    end process;
```

Kód má drobný nedostatek – soubor není nikdy uzavřen. Chceme-li jej uzavřít, pak musíme ukončit simulátor. Fragment vytvořeného souboru vypadá následovně. Pro formátování času je využito tolik míst, kolik je aktuálně potřeba, a navíc je připojen rozměr. Pro formátování std_logic je využit jen jeden znak. Hodnota preset je zarovnána doprava (a proto je u ní vlevo mezera), q je zarovnáno doleva a proto mezi preset a q mezera není:

```
...
75 ns0011 UUUUUUUUUUU0000000111
85 ns0011 UUUUUUUUUUU0000001000
95 ns0011 UUUUUUUUUUU0000001001
105 ns0011 UUUUUUUUUUU0000001010
115 ns0011 UUUUUUUUUUU0000001011
125 ns0011 UUUUUUUUUUU0000001100
135 ns0011 UUUUUUUUUUU0000001101
...
```

Zdá se to jednoduché, ale pokud se pokusíme předchozí soubor přečíst:

```
    process
        file f          : text;
        variable l      : line;
        variable status : file_open_status;
        variable t      : time;
        variable s      : std_logic;
        variable v      : std_logic_vector(WIDTH - 1 downto 0);
    begin
        file_open(status, f, "C:\work\filerd.txt", READ_MODE);
        if status = OPEN_OK then
            while not endfile(f) loop
                readline(f, l);
                read(l, t);
                read(l, s);
                read(l, s);
                read(l, s);
                read(l, s);
                read(l, v);
                read(l, v);
            end loop;
            file_close(f);
        end if;
        wait;
    end process;
```

… tak ho sice přečteme, ale hodnoty v něm správně do proměnných nerozdělíme. Při zápisu i čtení mějme na paměti jednotlivé položky oddělovat mezerami. Výjimku tvoří zápisy a čtení znaků.

Tímto jsme vyčerpali vše, co by se nám při kurzech číslicové techniky mohlo z jazyka VHDL hodit. Přejeme vám hodně štěstí a trpělivosti při popisování vlastních číslicových obvodů.