

Číslicové systémy a jazyk VHDL

Jiří Pinker
Martin Poupa



 **kontron**

426410

TECHNICKÁ
LITERATURA
BEN

Kniha je zaměřena na návrh číslicových systémů s využitím jazyka VHDL. Po zvládnutí této moderní metody může čtenář pracovat s programovatelnými logickými obvody. Informace o jednotlivých typických číslicových obvodech a systémech se prolínají s jejich popisem v jazyce VHDL, takže čtenář může porovnávat běžný popis schématem s popisem speciálním jazykem.

Kniha je určena především studentům magisterských oborů zabývajících se elektronikou. Vybrané partie mohou sloužit i pro studenty bakalářských oborů.

Jiří Pinker, Martin Poupa

Číslicové systémy a jazyk VHDL

Odborná recenze:

Ing. Robert Kvaček, Ing. Tomáš Tóth, Ing. Luboš Hradecký, Ing. Miloš Bečvář (ASICentrum s.r.o.)
Ing. František Kostka, CSc., AV ČR – ÚRE; doc. Ing. Jaromír Kolouch, CSc., VUT v Brně - FEKT

Bez předchozího písemného svolení nakladatelství nesmí být kterákoli část kopírována nebo rozmnožována jakoukoli formou (tisk, fotokopie, mikrofilm nebo jiný postup), zadána do informačního systému nebo přenášena v jiné formě či jinými prostředky.

Autoři a nakladatelství nepřejímají záruku za správnost tištěných materiálů. Předkládané informace jsou zveřejněny bez ohledu na případné patenty třetích osob. Nároky na odškodnění na základě změn, chyb nebo vynechání jsou zásadně vyloučeny.

Všechny registrované nebo jiné obchodní známky použité v této knize jsou majetkem jejich vlastníků. Uvedením nejsou zpochybňena z toho vyplývající vlastnická práva.

Veškerá práva vyhrazena

© Prof. Ing. Jiří Pinker, CSc., Ing. Martin Poupa, Ph.D., Západočeská univerzita v Plzni, 2006

© Nakladatelství BEN – technická literatura, Věšínova 5, Praha 10

Jiří Pinker, Martin Poupa: Číslicové systémy a jazyk VHDL

BEN – technická literatura, Praha 2006

1. vydání

ISBN 978-80-7300-198-5 (tištěná kniha)

ISBN 978-80-7300-385-2 (elektronická kniha v PDF)

OBSAH

Úvod

11

Signály v číslicových systémech

13

2.1 Dvojstavové signály	14
2.2 Třístavové signály	16
2.3 Dynamické parametry číslicových signálů	16

Jazyk VHDL

19

3.1 Historie, současnost, budoucnost a vlastnosti jazyka VHDL	20
3.2 Použité formátování pro výpisy kódu a syntaxi jazyka VHDL	23
3.3 Komentáře a identifikátory	24
3.4 Zápis čísel, znaků a řetězců	25
3.4.1 Zápis čísel v dekadické soustavě	25
3.4.2 Zápis čísel v dalších soustavách	25
3.4.3 Zápis znaku	26
3.4.4 Zápis textových řetězců	26
3.4.5 Zápis bitových řetězců	26
3.5 Entita, architektura a další návrhové jednotky	27
3.6 Základní datové typy	30
3.6.1 Výčtový typ	31
3.6.2 Celocíselný typ	32
3.6.3 Fyzický typ	33
3.6.4 Typ s plovoucí řádovou čárkou	34
3.6.5 Typ pole	35
3.6.6 Typ záznam	35
3.6.7 Typ soubor	36
3.7 Operátory	37
3.7.1 Logické operátory	37
3.7.2 Relační operátory	38
3.7.3 Operátory posuvu	38
3.7.4 Sčítací operátory a operátor spojení	39

3.7.5	Znaménkové operátory	40
3.7.6	Násobící operátory	40
3.7.7	Různé operátory	40
3.8	Základní objekty	41
3.8.1	Konstanty	41
3.8.2	Signály	41
3.8.3	Proměnné	42
3.8.4	Aliasy	42
3.8.5	Soubory	43
3.9	Paralelní příkazy	43
3.9.1	Nepodmíněné přiřazení	44
3.9.2	Podmíněné přiřazení	45
3.9.3	Výběrové přiřazení	46
3.9.4	Proces	47
3.9.5	Generující příkaz	48
3.9.6	Použití komponenty	49
3.9.7	Volání procedury	51
3.9.8	Příkaz bloku	51
3.9.9	Příkaz assert	51
3.10	Sekvenční příkazy	51
3.10.1	Sekvenční přiřazení do proměnné	52
3.10.2	Sekvenční přiřazení do signálu	52
3.10.3	Příkaz wait	53
3.10.4	Příkaz if	54
3.10.5	Příkaz case	54
3.10.6	Příkaz loop	55
3.10.7	Příkaz next	57
3.10.8	Příkaz exit	57
3.10.9	Příkaz return	57
3.10.10	Příkaz null	57
3.10.11	Příkaz assert	57
3.10.12	Příkaz report	58
3.10.13	Procedury	59
3.10.14	Funkce	59
3.11	Atributy	60
3.11.1	Atributy typů	60
3.11.2	Atributy polí	63
3.11.3	Atributy signálů	64
3.11.4	Atributy pojmenovaných objektů	66
3.11.5	Uživatelem definované atributy	67

3.12	Knihovny a knihovní balíky	67
3.12.1	Knihovní balík std_logic_1164	69
3.12.2	Knihovní balík numeric_std	71
3.12.3	Knihovna LPM	72
3.12.4	Uživatelské knihovny a knihovní balíky	75
3.13	Testovací prostředí pro ověřování funkčnosti navrženého systému	77
3.14	Seznam klíčových slov jazyka VHDL	80

Číslicové součástky a technologie 81

4.1	Vlastnosti číslicových součástek	82
4.2	Značení logických členů	85
4.3	Bipolární technologie	85
4.4	Unipolární technologie – CMOS	89
4.5	Technologie BiCMOS	92
4.6	Nevyužité vstupy číslicových součástek	93

Třístavové výstupy a otevřené kolektory 95

5.1	Obvody s třístavovými výstupy	96
5.2	Terminátory sběrnice	98
5.3	Výstupy s otevřenými kolektory	100

Kombinační obvody 103

6.1	Základní pravidla Booleovy algebry	104
6.2	Hlavní pravidla pro tvorbu a úpravy logických výrazů	105
6.3	Pravdivostní tabulka	106
6.4	Neurčené stavy	108
6.5	Minimalizace logické funkce	109
6.6	Skupinová minimalizace	111
6.7	Mapy a jejich použití	112
6.8	Návrh kombinačních obvodů	116
6.9	Využití multiplexorů	120

6.10	Využití členů EX-OR	123
6.11	Realizace kombinačních obvodů pamětí	125

Přechodné děje v kombinačních obvodech 127

7.1	Zpoždění signálů ze vstupů na výstupy	128
7.2	Hazardní stavy v kombinačních obvodech	130
7.3	Hledání hazardu pomocí map	132
7.4	Hledání hazardu z výrazu	134
7.5	Hazard při změně více než jedné proměnné	135
7.6	Potlačení falešného impulzu filtrem	135
7.7	Potlačení falešného impulzu registrem	136
7.8	Vliv úprav výrazu a obvodu na hazardy	136
7.9	Dynamický hazard	137

Základní funkční bloky 139

8.1	Dekodér	140
8.2	Multiplexor	143
8.3	Demultiplexor	146
8.4	Prioritní kodér	148
8.5	Číslicový komparátor	150
8.6	Sčítáčka	152
8.7	Odčítáčka	154
8.8	Převodník kódu	156
8.9	Asynchronní klopné obvody	158
8.10	Synchronní klopné obvody	163
8.11	Transformace klopných obvodů	173
8.12	Blokování klopných obvodů	174

Registry a čítače 177

9.1	Datové registry	178
9.2	Posuvné registry	180
9.3	Posuvné registry se zpětnou vazbou	184

9.4	Charakteristika čítačů	188
9.5	Asynchronní čítače	190
9.6	Synchronní čítače	191
9.7	Nulování a přednastavení čítače	195
9.8	Přenosy z čítače	197
9.9	Čítače „modulo M“	199

Sekvenční obvody

203

10.1	Přechodová a výstupní funkce	204
10.2	Popis pomocí grafu	205
10.3	Popis pomocí soustavy rovnic	207
10.4	Popis pomocí tabulek	208
10.5	Popis v některém programovacím jazyku	208
10.6	Obvodová realizace konečného automatu	213
10.7	Kódování stavů	215
10.8	Volba klopných obvodů	217
10.9	Návrh budicích funkcí	217
10.10	Časování signálů v synchronním sekvenčním obvodu	219
10.11	Nastavení počátečního stavu	223

Tvarovací a časovací obvody

225

11.1	Asynchronní časovací obvody	226
11.2	Synchronní časovací obvody	230
11.3	Impulzní šířkový modulátor	234

Paměti

237

12.1	Rozdělení pamětí	238
12.2	Důležité parametry pamětí	238
12.3	Paměti paralelní a sériové	239
12.4	Vnitřní uspořádání paměti s adresovým výběrem	240
12.5	Statické paměti RAM	243
12.6	Dynamické paměti RAM	245

12.7	Synchronní dynamické paměti RAM	250
12.8	Synchronní dynamické paměti DDR	254
12.9	Permanentní paměti	255
12.10	Dvojbránová paměť	261
12.11	Paměť fronty	266

Programovatelné logické obvody

271

13.1	Historie programovatelných logických obvodů	272
13.2	Jednoduché programovatelné logické obvody (SPLD)	277
13.3	Komplexní programovatelné logické obvody (CPLD)	279
13.4	Programovatelná logická pole (FPGA)	283
13.5	Základní dynamické parametry PLD	288

Mikroprogramový automat

289

14.1	Základní obvody mikroprogramového automatu	290
14.2	Obvody pro modifikaci adres	292
14.3	Obvody pro vyhodnocení podmínek přechodu	296
14.4	Nastavení počáteční adresy	298
14.5	Časování signálů mikroprogramového automatu	299
14.6	Dynamické výstupy	301
14.7	Pomocné obvody	301
14.8	Použití mikroprogramového automatu	303

Návrh rozsáhlých systémů

305

15.1	Synchronizace vnitřních obvodů systému	307
15.2	Časování vstupních signálů	309
15.3	Korespondenční provoz	310
15.4	Zřetelené zpracování	312
15.5	Systémy RTL	314
15.6	Návrh číslicového obvodu při použití RTL popisu	320

Literatura**323****Seznam zkratek a anglických termínů****327****Příloha: zjednodušená syntaxe jazyka VHDL****331****Rejstřík****341**

Adresy prodejen technické literatury	351
Pár slov o nakladatelství	352

ÚVODNÍ SLOVO

Tato kniha je zaměřena na číslicové součástky a systémy. Čtenáři jsou předkládány informace postupně „od malého k velkému“. Jsou vysvětleny vlastnosti jednotlivých číslicových součástek, standardních funkčních bloků, kombinačních a sekvenčních obvodů a rozsáhlých systémů. Důraz je kladen na ty informace, které jsou podstatné pro navrhování systémů.

Naprosto samozřejmostí při návrhu systémů je v současné době počítačová podpora. Pro popis systémů byly vyvinuty speciální jazyky HDL (Hardware Description Language – jazyk pro popis hardware), které zastávají současně tři funkce: popis systému, simulace systému a generace výrobních podkladů. Pod pojmem výrobních podkladů je zde méněn výstupní soubor, který je počítačem vytvářen, a který popisuje vzájemné propojení prvků, jimiž disponuje cílová technologie. To se týká zvláště programovatelných logických obvodů. Obvody tohoto typu se v současné době velmi rychle vyvíjejí a dosahují rozsahu a výkonu, kterým konkurují signálovým procesorům, nebo je i přední. Popisu funkčních bloků a systémů jazykem VHDL je proto v této knize věnována patřičná pozornost. Jazyk VHDL patří mezi jazyky HDL a je v Evropě nejvíce rozšířen.

V prvních dvou kapitolách jsou probrány vlastnosti signálů specifické pro číslicové systémy. Třetí kapitola podává poměrně podrobný popis jazyka VHDL, ze kterého se pak vychází v dalších kapitolách. Čtvrtá a pátá kapitola se týkají součástek v číslicových systémech. V šesté a sedmé kapitole jsou probrány kombinační obvody a jejich dynamické vlastnosti. Základní stavební bloky používané v návrhu systémů jsou popsány v osmé a deváté kapitole. Desátá kapitola pojednává o sekvenčních systémech a o některých problémech časování signálů. Časovací obvody jsou probrány v následující jedenácté kapitole. Velmi rozsáhlá problematika číslicových paměťových obvodů je shrnuta ve dvanácté kapitole. Ve třinácté kapitole jsou popsány programovatelné logické obvody, což jsou velmi perspektivní a stále častěji používané součástky v číslicových systémech. Problematice rozsáhlých systémů je věnována čtrnáctá a zvláště pak patnáctá kapitola. V příloze je shrnuta syntaxe jazyka VHDL. Kniha je dále doplněna seznamem zkratky a anglických termínů, seznamem literatury s příbuznou tématikou, a rejstříkem.

Práce na této publikaci byla podporována z projektu č. CZ.04.1.03/3.2.15.1/0056 Operačního programu Rozvoj lidských zdrojů.

Autoři děkují všem, kteří svými připomínkami a radami přispěli k finální podobě této knihy. Zejména děkují doc. Ing. Jaromíru Kolouchovi, CSc., z Vysokého učení technického v Brně a Ing. Františku Kostkovi, CSc., z Ústavu radiotechniky a elektroniky AV ČR v Praze za pečlivou recenzi rukopisu. Přes veškerou snahu, která byla psaní této knihy věnována, je bohužel pravděpodobné, že se v ní vyskytnou chyby. Pokud vážený čtenář nějaké chyby najde, prosíme, aby nás na ně upozornil. Uvítáme také všechny Vaše názory a případné dotazy.

Plzeň, srpen 2006
Jiří Pinker a Martin Poupa



ÚVOD

Číslicové systémy jsou v současné době podstatnou složkou převážné většiny elektrotechnických zařízení. Může se jednat o personální počítače, počítače pro řízení, systémy pro zpracování signálů, řídicí systémy v automobilech, televizní přijímače, laboratorní přístroje, atd., až po pračky, myčky a jiná zařízení v domácnosti. K takovému rozšíření číslicových systémů přispělo několik vlivů.

Číslicové systémy pracují s číslicovými signály. Jejich podstatnou vlastností je to, že nabývají vždy jen konečného počtu hodnot. Tím se liší od signálů analogových, které jsou spojité a nabývají nekonečného počtu hodnot. Tento rozdíl je podstatný pro snížení nároků na přesnost obvodů a tím i na jejich cenu a možnost dosažení vysokého stupně integrace.

Podstatné vlastnosti číslicových systémů lze shrnout do několika bodů:

- Lze je využít pro konstrukci univerzálních systémů, jako je počítač, jejichž činnost lze definovat programem.
- Jsou vhodné pro zpracování informace v různé formě – čísel, textů, obrazů, atd.
- Konečný počet hodnot číslicových signálů může být reprezentován vektory, jejichž složky mají jen dvě hodnoty.
- Zvýšení přesnosti při zpracování čísel lze jednoduše docílit zvětšením počtu bitů.
- Jsou necitlivé ke změnám parametrů svých součástek, hlavně k vlivům kolísání teploty a výrobních tolerancí.
- Dobře se hodí pro realizaci jako integrované obvody.
- Při jejich návrhu lze respektovat požadavky na rychlosť zpracování signálů a na složitost, lze volit kompromisy.

S postupně rostoucí složitostí číslicových systémů nabývají na důležitosti prostředky pro jejich počítačový návrh a simulaci EDA (Electronic Design Aids). Počítačové zpracování vychází z popisu systému některým ze speciálních jazyků HDL (Hardware Description Language). Ve světě jsou nejrozšířenější dva z těchto jazyků – VHDL a Verilog.

Úspěšné využívání počítačových návrhových prostředků není možné bez dobré znalosti základních stavebních bloků číslicových systémů, jejich vlastností a problematiky jejich vzájemné spolupráce. V následujících kapitolách je proto tradiční popis jednotlivých číslicových součástek a funkčních bloků schématy, logickými výrazy, tabulkami a mapami doplněn o popis v jazyce VHDL. Čtenář tak získá znalosti o reálných obvodech současně s představou o jejich formálním popisu speciálním jazykem. To bylo cílem této knihy.

Přehled základních vlastností jazyka VHDL je podán ve třetí kapitole. Vzhledem k rozsáhlosti podrobného výkladu jazyka VHDL, který by zabral několik set stran, jsou v této kapitole popsány pouze základní prvky a část syntaxe nejčastěji používaných příkazů jazyka. Nepředpokládá se, že by čtenář nejprve důkladně prostudoval tuto kapitolu, a teprve pak se věnoval ukázkám použití VHDL při popisu funkčních bloků, probíraných v dalších kapitolách. Vhodnější přístup je informativní přečtení kapitoly 3, intuitivní porozumění jazykovým konstrukcím v navazujících kapitolách, a pak návrat k detailnímu porozumění syntaxi jazyka. V případě hlbššího zájmu o jazyk VHDL se nabízí řada kvalitních literárních pramenů (viz seznam na konci knihy).



SIGNÁLY V ČÍSLICOVÝCH SYSTÉMECH

V číslicových systémech se pracuje se signály, které mají jen **konečný** počet diskrétních hodnot. Tím se liší od systémů analogových, u kterých jsou signály spojité a mohou ve vymezeném rozsahu nabývat nekonečného počtu hodnot. V číslicovém systému může i **čas** být veličinou diskrétní, tj. signály se mohou měnit jen v určitých okamžicích. Takovéto číslicové systémy se pak nazývají **synchronní** – na rozdíl od systémů asynchronních, u kterých ke změnám signálů může docházet kdykoliv. Synchronní systémy jsou podstatně častější, neboť existence přesně stanovených okamžiků změn signálů zavádí „pořádek“ do **časování signálů** v systému a tím usnadňuje jeho konstrukci i výrobu v podobě integrovaných obvodů. Přesné časování je zajištěno hodinovými (taktovacími) impulzy, což je velmi významný signál systému.

Číslicové systémy se dělí na dvě skupiny – **kombinační** systémy a **sekvenční** systémy. U kombinačních systémů jsou výstupní signály závislé pouze na současných vstupních signálech. U sekvenčních systémů jsou výstupní signály závislé nejen na současných vstupních signálech, ale i na vstupních signálech v minulosti. Systém tedy má vnitřní paměť.

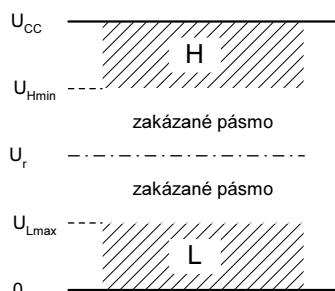
2.1 Dvojstavové signály

Jak již bylo řečeno, číslicové signály mají jen konečný počet diskrétních hodnot. V naprosté většině jsou to jen dvě hodnoty. Dvouhodnotové nebo dvoustavové signály snižují nároky na výrobní tolerance. Bylo tak možné zavést výrobní postupy, které umožňují hromadnou a levnou výrobu součástek.

Předpokládejme, že číslicové součástky jsou napájeny kladným napětím $+U_{CC}$. Jedna hodnota bude vyjádřena nižším napětím, druhá vyšším napětím. Dvě možné hodnoty signálu označíme jako '0' a '1' (v souladu se značením v Booleově algebře). Jejich přiřazení nižšímu či vyššímu napětí je možné dvojím způsobem:

- u **pozitivní logiky** odpovídá nižší napětí hodnotě „0“ a vyšší napětí hodnotě „1“
- u **negativní logiky** odpovídá vyšší napětí hodnotě „0“ a nižší napětí hodnotě „1“

Při záporném napájecím napětí by platilo stejné pravidlo. Tak např. přiřazení napětí – 10 V hodnotě „0“ a –2 V hodnotě „1“ by odpovídalo pozitivní logice (což může být poněkud překvapivé).



Obr. 2.1 Pásma napětí L a H u číslicových signálů

Zmírnění nároků na tolerance napětí je dosaženo tak, že logickým hodnotám jsou přiřazena celá **pásma napětí** – viz obr. 2.1. Existuje pásmo nižších napětí L (angl. *Low*) a pásmo vyšších napětí H (angl. *High*). Všechny hodnoty napětí, spadající do pásmu L, jsou označeny jako U_L a všechny hodnoty napětí, spadající do pásmu H, jsou označeny jako U_H . Mezními hodnotami jsou U_{Lmax} a U_{Hmin} .

V pozitivní logice by všechny hodnoty napětí $\leq U_{Lmax}$ odpovídaly hodnotě „0“ a všechny hodnoty napětí $\geq U_{Hmin}$ by odpovídaly hodnotě „1“. Mezi těmito mezemi je **zakázané pásmo**, přes které signál přechází jen při změně stavu obvodu (v přechodných dějích). Dlouhodobé setrvání napětí v tomto pásmu může vést k nepředvídanému chování obvodu – zpravidla k jeho **rozkmitání**.

Mezi krajními mezemi U_{Hmin} a U_{Lmax} leží **referenční napětí** U_r (též „rozhodovací napětí“) – zpravidla ve středu zakázaného pásmu. K tomuto napětí se vztahují některé údaje o dynamických parametrech obvodu aj.

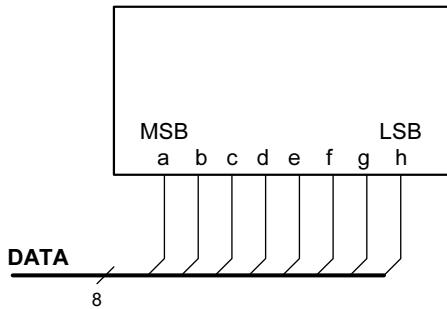
Za **ideálních podmínek**, tj. při hodnotách napětí zaručeně uvnitř přípustných pásem a při nekonečně rychlých přechodech mezi nimi, lze místo s hodnotami napětí pracovat s logickými hodnotami „0“ a „1“. Tím se velmi usnadní návrh systému, neboť lze používat aparát Booleovy algebry.

Ve skutečnosti však je třeba brát v úvahu ještě další vlivy:

- Přechodné děje nelze zanedbat.
- Správné úrovně napětí mohou být porušeny nadmernou zátěží číslicových součástek.
- Do systému mohou pronikat rušivé signály z okolí, nebo si je může systém generovat svojí činností sám. Rušivé signály se superponují na původní signály.

Jestliže jeden signál může nabývat dvou hodnot, pak dvojice signálů může nabývat čtyř hodnot, a skupina n signálů může nabývat 2^n hodnot. Seskupení signálů je v praxi velmi časté – např. adresy nebo data v počítači, apod. Skupina signálů se stejným určením se nazývá **vektor** a skupina vodičů, vedoucích tyto signály, se nazývá **sběrnice**. Vektor má n složek. Vektor o $n+1$ signálech bude nabývat $2^{n+1} = 2 \times 2^n$ hodnot. Přidáním jednoho signálu se tedy **zdvojnásobí** rozlišení. To je důležitá vlastnost číslicových systémů. Jednotlivé složky vektoru je nutné **označit** tak, aby bylo zřejmé, který signál odpovídá **nejvyššímu bitu** čísla (dat, adres) a který **nejnižšímu bitu**. Jednotlivé signály se běžně označují indexy. Např. skupina signálů (vektor) X má 8 složek x_7, \dots, x_0 . Je ovšem nutné předem zavést dohodu, že signál x_7 bude mít význam nejvyššího (nebo třeba naopak nejnižšího) bitu a zavedenou konvenci **dodržovat** během celého návrhu. Jednotlivé signály lze samozřejmě označovat i jinak – např. písmeny a, b, c, \dots, h , atd. Opět je nutné stanovit, které z písmen má u daného vektoru význam nejvyššího bitu čísla a které naopak nejnižšího. K tomu se používá označení **MSB** pro nejvyšší bit (angl. *Most Significant Bit*) a **LSB** pro nejnižší bit (angl. *Least Significant Bit*). Obr. 2.2 ukazuje výsek ze schématu s vyznačením nejvyššího a nejnižšího bitu.

Dosud jsme si všímali signálů **napěťových**. Je to nejčastější případ. Méně často se využívají i signály **proudové**. Proud může téci dvěma směry a tím rozlišit logickou hodnotu „0“ a „1“. Výhodou proudových signálů je to, že nemusí docházet ke změně napětí (nebo jen k velmi malé) a tím je značně omezen nežádoucí vliv **parazitních kapacit** v obvodu.



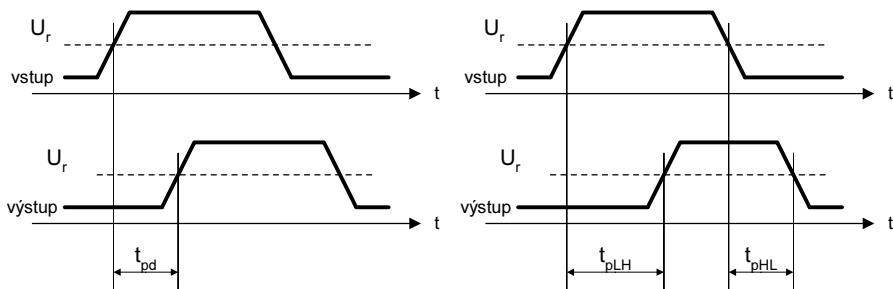
Obr. 2.2 Vektor a jeho složky

2.2 Třístavové signály

Kromě hodnot „0“ a „1“ mohou třístavové signály nabývat ještě hodnoty „Z“. Hodnota „Z“ odpovídá stavu, kdy obvod, který signál generuje, nedodává žádný proud (respektive proud jen zcela zanedbatelný) a je tedy jakoby „odpojen“. Stav „Z“ se nazývá též **vysokoimpedanční**. Součástky s třístavovými výstupy lze spojovat paralelně, tj. do uzlu – z celé skupiny paralelně spojených výstupů však nejvýše jeden může být ve stavu „0“ nebo „1“, všechny ostatní musí být ve stavu „Z“. Obvod s výstupem ve stavu „Z“ nijak neovlivňuje napětí v uzlu – kdyby byl obvod samostatný, nebylo by jeho výstupní napětí vůbec definováno. Napětí v uzlu však zpravidla je definováno některou z ostatních součástek v uzlu, např. rezistorem spojeným se zemí, napájecím napětím, nebo jiným napětím.

2.3 Dynamické parametry číslicových signálů

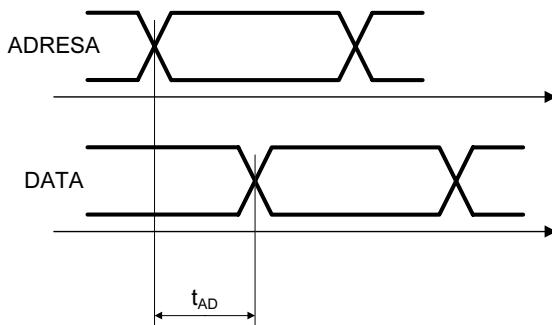
Pro číslicové signály je typický lichoběžníkový časový průběh. Proto je stanovení dynamických parametrů relativně velmi jednoduché. Nejdůležitější je stanovení **doby zpoždění průchodu signálu** (součástkou) jako zpoždění od průchodu signálu **referenční úrovni** na vstupu a na výstupu součástky. Tato doba se označuje jako t_{pd} (z angl. *propagation delay*).



Obr. 2.3 Hlavní dynamické parametry číslicového signálu

V časovém diagramu na levé části obr. 2.3 je zpoždění zhruba stejné při změně vstupního signálu z pásmo L do H i z H do L. Ve skutečnosti se tato zpoždění mohou lišit, jak ukazuje pravý obrázek. Pak rozlišujeme dobu t_{PLH} (při změně výstupního signálu z L do H) a t_{PHL} (při změně opačné). Pro hrubou orientaci se pak uvádí doba t_{pd} jako průměr obou.

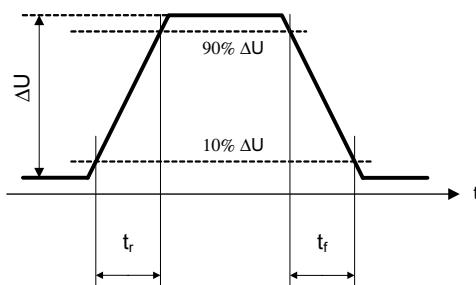
V mnoha případech je třeba časovým diagramem vyjádřit jen **okamžik změny** signálu bez ohledu na to, zda je to z L do H nebo naopak. Obr. 2.4 ukazuje takový příklad, kdy je sledováno zpoždění t_{AD} mezi vydáním adresy a vydáním dat v počítači. Adresu i data je třeba chápat jako vektory.



Obr. 2.4 Znázornění okamžiků změn signálu

Při současné úrovni technologie jsou doby zpoždění běžných číslicových součástek (integrovaných obvodů) rádu ns (nanosekund, 10^{-9} s), zpoždění uvnitř integrovaného obvodu jsou o jeden až dva řády kratší.

Při tak krátkých dobách se musí brát v úvahu i zpoždění signálu při **průchodu signálu spojem**. Na běžném plošném spoji je zpoždění kolem 1 ns na 10 cm délky cesty. To může hrát významnou roli zvláště u rozvodu hodinových impulzů. Časování událostí na jednom konci plošného spoje je pak jiné, než na jeho druhém konci. Existence zpoždění ve spojích ukazuje, že současnost událostí je relativní nejen v kosmických rozměrech.



Obr. 2.5 Definice doby náběhu a doběhu

Dalšími významnými parametry číslicových signálů jsou **doba náběhu a doběhu** impulzu. Ty jsou všeobecně definovány při protnutí signálu úrovní 10 % a 90 % z ustáleného stavu – viz obr. 2.5.

Doba náběhu t_r (angl. *rise time*) a doba doběhu t_f (angl. *fall time*) se u běžných číslicových obvodů udává v jednotkách nebo zlomcích ns. Ve specifikacích výrobců číslicových součástek mohou být doby náběhu a doběhu u některých signálů definovány i jinak.

Doba náběhu a doběhu záleží jednak na vlastnostech samotné součástky, jednak na její zátěži – hlavně na **kapacitní složce** zátěže. Proud, potřebný pro nabíjení a vybíjení zatěžovací kapacity, je přibližně dán jako:

$$I_C = C \cdot \frac{\Delta U}{\Delta t},$$

kde za Δt je třeba dosadit celou dobu přechodného děje (tedy $1/0,8 t_r$ nebo t_f). Výstupní proud součástky I_O je však omezen, zpravidla na několik jednotek až desítek mA. Proto i **strmost** impulzu $\Delta U/\Delta t$ bude omezena. Např. u součástky s napěťovým skokem 2 V, s výstupním proudem omezeným na 2 mA a se strmostí impulzu 2 V/ns v nezatiženém stavu, by se při zátěži $C = 10 \text{ pF} (= 10^{-11} \text{ F})$ doba přechodného děje prodloužila na

$$\Delta t = C \cdot \frac{\Delta U}{I_o} = 10^{-11} \cdot \frac{2}{2 \cdot 10^{-3}} = 10^{-8} \text{ [s]},$$

tedy 10 ns. To je významné zhoršení.

Kapacitní zátěži součástky jsou jednak **vstupní kapacity** připojených součástek, jednak parazitní **kapacity vodičů** na plošném spoji. Vstupní kapacity se běžně pohybují mezi 5 až 10 pF, kapacity vodičů na plošném spoji kolem 1 pF na 1 cm délky (hrubým odhadem, záleží samozřejmě na provedení plošného spoje). Výše uvedený číselný příklad je tedy zcela realistický – odpovídá zátěži jedním vstupem další součástky, vzdálené jen o 5 cm. Přesto došlo k výraznému vlivu.

Pro větší kapacitní zátěž, ať již od velkého počtu vstupů dalších součástek, nebo od dlouhého spoje, je třeba volit součástky s **velkým výstupním proudem** – běžně přes 50 mA. Jedná se vždy o krátkodobé proudové impulzy, nikoliv o trvalý stav. Není proto nutné uvažovat o podstatném zvýšení příkonu. Dalším opatřením je **zkrácení spojů**. Moderní metody návrhu desek s **plošnou montáží** to umožňují. Přesto je nutné počítat s tím, že systém, konstruovaný ze vzájemně propojených obvodů menší integrace, bude vždy značně pomalejší než systém, **integrovaný** v jedné součástce, ve které jsou délky spojů a i parazitní kapacity o několik rátů menší.



JAZYK VHDL

Jazyk VHDL je jazyk vysoké úrovně navržený speciálně pro účely popisu (návrhu) a simulace velmi rozsáhlých číslicových obvodů a systémů. Jedná se tedy o jazyk určený pro popis hardware. Jazyky pro popis hardware jsou označovány v cizojazyčné literatuře zkratkou HDL, což je akronym z *Hardware Description Language*. Výhodou tohoto jazyka jsou jeho bohaté vyjadřovací schopnosti a značná nezávislost číslicového systému popsaného jazykem VHDL na cílové technologii jeho realizace nebo výroby. Vzhledem k rozsahu knihy jsou v této kapitole popsány pouze základní prvky a část syntaxe nejčastěji používaných příkazů jazyka VHDL, neboť úplný podrobný výklad jazyka VHDL by vydal na samostatnou knihu (vlastní IEEE standard jazyka VHDL má cca 300 stran). Syntaxe jazyka VHDL uvedená v této kapitole je popisovaná dle v současnosti nejvíce používaného standardu jazyka VHDL, jímž je standard *IEEE Std 1076-1993*. V případě hlubšího zájmu o jazyk VHDL existuje mnoho kvalitní literatury, většinou cizojazyčné (viz např. literatura použitá při psaní této knihy).

Je třeba zdůraznit, že popis číslicového systému (hardware) v jazyce VHDL je činnost značně odlišná od programování v klasických programovacích jazyčích typu C nebo Pascal. Při popisu číslicového systému jazykem VHDL je důležité myslit na to, že popisujeme číslicový systém, který většinou nakonec chceme realizovat v hardware. To znamená, že nám vytvořený kód bude muset nakonec projít syntézou, jejímž výsledkem bude zapojení z hradel a klopných obvodů určené pro programovatelný logický obvod nebo jiný typ zákaznického obvodu. Konstrukce, které v jazyce VHDL při popisu navrhovaného číslicového systému vytváříme, musí tedy být většinou **syntetizovatelné** (s výjimkou testovacího programu nebo modelů určených pouze pro simulaci na simulátoru). Testovací program (angl. *testbench*) je program v jazyce VHDL, jímž generujeme vstupní signály pro nám navrhovaný číslicový systém a kontrolujeme správnost jeho funkce (simulací a testbenchům se dále věnuje podkapitola 3.13). Kód testbenche přitom není třeba syntetizovat, neboť se používá pouze pro simulaci na simulátoru jazyka VHDL. Značná část příkazů a vyjadřovacích prostředků, které nám jazyk VHDL nabízí, nelze nebo zatím nelze syntetizovat do hardware. Lze je tedy pouze simulovat na simulátoru. V kódu, který je třeba syntetizovat, tedy nelze používat všechny konstrukce a výmožnosti, které nám jazyk VHDL nabízí. Ve většině příkladů výpisů kódu v jazyce VHDL, uváděných v následujících kapitolách, se jedná o syntetizovatelný kód (tedy kód, který jde jak simulovat na simulátoru, tak realizovat dnešními syntezátorami v hardware). V případech, kdy se jedná o kód, který nelze syntetizovat do hardware, ale pouze simulovat na simulátoru, je na tuto skutečnost upozorněno v komentáři nebo v doprovodném vysvětlujícím textu. Dále je také třeba podotknout, že se syntezátoru jednotlivých výrobců drobně liší v tom, co dokáží syntetizovat. Projevuje se to hlavně při syntéze složitějších konstruktů jazyka VHDL, kde některé syntezátoru mohou selhávat, zatímco jiné syntézu stejného kódu úspěšně zvládají. Obecně však lze říci, že schopnosti syntezátorů se neustále zvětšují a zvládají dnes již syntézu poměrně složitých konstrukcí jazyka VHDL.

3.1 Historie, současnost, budoucnost a vlastnosti jazyka VHDL

Vývoj jazyka VHDL byl zahájen v roce 1981 v rámci výzkumného projektu VHSIC (angl. *Very High Speed Integrated Circuits*) ministerstva obrany Spojených států amerických (U.S. Department of Defense). V rámci tohoto projektu byl řešen mimo jiné problém

efektivního popisu velmi rozsáhlých integrovaných obvodů, neboť vývoj a simulace navrhovaných obvodů byl tehdy řešen pomocí různých vzájemně nekompatibilních jazyků a nástrojů. Také dokumentace funkcí takto navrhovaných obvodů byla velmi neefektivní a nedostatečná. Jedním z cílů výzkumného projektu VHSIC bylo navrhnout jazyk vysoké úrovně s velmi rozsáhlými vyjadřovacími schopnostmi, který bude umožňovat simulaci a návrh číslicových obvodů nezávisle na cílové technologii a metodice návrhu. Vlastní vývoj základů jazyka byl v roce 1983 zadán firmám, které se výzkumnému projektu VHSIC také účastnily. Jednalo se o firmy IBM, Intermetrics a Texas Instruments. Tyto firmy v letech 1983 až 1985 vytvořily základ definice jazyka VHDL, který byl poprvé veřejně publikován v roce 1985. Syntaxe jazyka VHDL vycházela z jazyka ADA. Označení **VHDL** vzniklo jako akronym z názvu *VHSIC Hardware Description Language*. Pro další rozvoj, vývoj a standardizaci jazyka postoupilo ministerstvo obrany jazyk VHDL v roce 1986 organizaci IEEE (angl. *Institute of Electrical and Electronics Engineers*). V roce 1987 byl organizací IEEE poprvé publikován standard jazyka VHDL pod označením *IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1987)*. Tento standard je v literatuře někdy označován jako VHDL-87. Dle zvyků standardizačního procesu organizace IEEE by měl být standard jazyka VHDL v pravidelných pětiletých intervalech revidován, což se i děje (někdy se zpožděním). Standard jazyka VHDL od svého prvního vydání v roce 1987 prošel zatím celkem třemi revizemi a jedním rozšířením.

Historie jazyka VHDL v letopočtech:

- 1981: zahájení vývoje jazyka, projekt VHSIC U.S. DoD
- 1983–1985: vývoj základů jazyka firmami (IBM, Intermetrics, TI)
- 1986: postoupení jazyka organizaci IEEE
- 1987: publikování standardu IEEE Std 1076-1987 (VHDL-87)
- 1994: publikování standardu IEEE Std 1076-1993 (VHDL-93)
- 1999: publikování standardu IEEE Std 1076.1-1999 (VHDL-AMS)
- 2000: publikování standardu IEEE Std 1076-2000 (VHDL-2000)
- 2002: publikování standardu IEEE Std 1076-2002 (VHDL-2002)

První verzi standardu jazyka VHDL IEEE Std 1076-1987 a její následnou revizi IEEE Std 1076-1993 implementovali do svých návrhových systémů postupně všichni výrobci návrhových systémů pracujících v této oblasti. Obě verze standardu jsou tak dnes podporovány ve všech návrhových a simulačních systémech všech výrobců. V roce 1999 byl vydán standard *IEEE Standard VHDL Analog and Mixed-Signal Extensions (IEEE Std 1076.1-1999)*, který bývá označován jako VHDL-AMS. VHDL-AMS je nadstavbou jazyka VHDL, která obsahuje rozšíření jazyka VHDL pro popis a simulaci analogových a smíšených (analogově-číslicových) systémů. V roce 2000 byla publikována v pořadí druhá revize jazyka VHDL označená IEEE Std 1076-2000. Tato revize však nepřinesla mnoho významných změn. Díky tomu se zřejmě také nesetkala s odesvou ze strany firem. Podporu tohoto standardu tak nenajdeme téměř v žádném současném návrhovém systému. Poslední revize standardu jazyka VHDL, označená IEEE Std 1076-2002, nalezla zatím odesvou pouze u některých výrobců simulátorů jazyka VHDL (např. Mentor Graphics). Díky tomu se v současnosti používá pro návrh i simulaci v jazyce VHDL většinou standard IEEE Std 1076-1993. Tento standard je

v literatuře většinou označován jako VHDL-93. Vzhledem k tomu, že byl vydán v roce 1994, jedná se dnes již o poměrně starý standard.

Další revize standardu jazyka VHDL je očekávána v historicky blízké době (v době psaní tohoto textu se blíží konec pětiletého intervalu revize). Tato očekávaná revize jazyka VHDL bývá často označována jako VHDL-200X. Nová revize standardu jazyka VHDL by měla přinést mnoho významných vylepšení. Z očekávaných vylepšení jmenujme například knihovní balíky pro podporu matematických operací v pevné řádové čárce (*ieee.fixed_pkg*) a plovoucí řádové čárce (*ieee.fphdl_pkg*), konverzní funkce pro převod většiny předdefinovaných typů a typů z knihoven IEEE na řetězec, ochranu IP (*Intellectual Property*) funkcí pomocí šifrování VHDL kódu standardními šifrovacími metodami, možnost použít podmíněné a výběrové přiřazení v procesu a další mnohá vylepšení. Pokud se očekávaná vylepšení do další revize jazyka VHDL opravdu dostanou, dá se předpokládat její implementace většinou výrobců návrhových a simulačních systémů.

Kromě jazyka VHDL existují samozřejmě i další jazyky pro popis číslicových systémů. Jmenujme proto alespoň další dva HDL jazyky standardizované organizací IEEE, a to Verilog a SystemVerilog. Standard jazyka Verilog byl poprvé publikován v roce 1995 pod označením *IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language (IEEE Std 1364-1995)*. V roce 2001 pak byla publikována jeho první revize pod označením *IEEE Standard Verilog® Hardware Description Language (IEEE Std 1364-2001)*. Jazyk Verilog má syntaxi podobnou známému jazyku C a má také podobné vyjadřovací schopnosti jako má jazyk VHDL. Verilog je společně s VHDL dnes používán k návrhu většiny rozsáhlých číslicových systémů (např. procesorů, čipsetů, programovatelných logických obvodů, aj.). Mnoho dnešních návrhových a simulačních systémů proto podporuje oba jazyky – Verilog i VHDL. Návrh jednoho číslicového systému se tak dnes (a není to výjimkou) může skládat z částí systému napsaných v jazyce Verilog i VHDL. Jen spíše pro zajímavost uvedeme, že používání jazyka VHDL a Verilog je také geograficky odlišné. Používání jazyka VHDL je rozšířeno zejména na univerzitách a firmách v Evropě, zatímco Verilog je rozšířen zejména v asijských zemích. V USA je v univerzitním i firemním prostředí rozšířeno používání obou jazyků. Dalším výše zmíněným jazykem je SystemVerilog. Tento jazyk je založen na jazyce Verilog a silně rozšířil možnosti původního jazyka. Standard jazyka SystemVerilog byl publikován v roce 2005 pod označením *IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language (IEEE Std 1800-2005)*.

V současné době se velmi rozvíjejí jazyky určené pro popis číslicových systémů, které jsou založené přímo na jazyce C nebo C++. Mezi tyto jazyky patří například Handel-C a SystemC. Jazyk Handel-C je založen na jazyce C, jazyk SystemC pak na objektovém jazyce C++. Standard jazyka SystemC byl již dokonce nedávno publikován (rok 2006) pod označením *IEEE Standard SystemC Language Reference Manual (IEEE Std 1666-2005)*. Tyto jazyky umožňují buď přímou syntézu číslicového systému navrženého v těchto jazycích do cílové technologie, nebo syntézu založenou na překladu do **RTL** (*Register Transfer Level* – blíže viz podkapitola 15.5) popisu systému v jazycích VHDL nebo Verilog a následnou syntézu do cílové technologie. Tyto nové jazyky byly vytvořeny z důvodu dosažení ještě větší abstrakce od hardware, než jaké dosahují dnes již „klasické“ jazyky pro popis hardware jako je VHDL nebo Verilog. Tyto nové jazyky již tedy nejsou koncipovány pro přímý popis hardware (ve smyslu jako jazyk VHDL nebo Verilog). Nejedná se tedy o HDL jazyky, ale jedná

se spíše o jazyky vysoké úrovni, jejichž výstupem je hardware. Návrhář číslicového systému v těchto nových jazycích tak již nemusí klást takovou pozornost a důraz na to, jak bude syntézu jednotlivých sekvencí kódu implementovat na úrovni hradel a klopných obvodů, a může se mnohem více věnovat vývoji a optimalizaci návrhu systému na algoritmické úrovni. Optimalizace navrhovaného systému na úrovni hradel, klopných obvodů a podobně je v těchto jazycích již hodně skryta a ponechána na komplikátoru a syntezátoru těchto jazyků. Z uvedeného je vidět, že vývoj v oblasti jazyků vhodných pro popis číslicových systémů se velmi rozvíjí a jeho tempo i úroveň abstrakce se bude v budoucnosti zřejmě dále zvětšovat.

Výše uvedené jazyky se samozřejmě mezi sebou navzájem liší vyjadřovacími schopnostmi, úrovní abstrakce, striktní typovostí nebo naopak typovou tolerancí, úsporností nebo naopak rozvláčnosti zápisu, mají však i mnoho společných vlastností. Jmenujme proto z našeho pohledu alespoň čtyři nejdůležitější vlastnosti:

- Číslicové systémy navržené v těchto jazycích jsou značně nezávislé na cílové implementaci v reálném hardware. Číslicový systém navržený pomocí těchto jazyků tak lze implementovat v programovatelných obvodech různých typových řad od různých výrobců. V případě potřeby výroby velkých sérií lze takto navržený systém použít i pro implementaci plně zákaznického obvodu ASIC.
- Uvedené jazyky umožňují simulovat vytvořený číslicový systém na simulátorech příslušných jazyků a také v těchto jazycích psát vlastní testovací prostředí pro navrhovaný číslicový systém.
- Většina uvedených jazyků byla publikována formou otevřeného IEEE standardu. Nejedná se tedy o uzavřené proprietární firemní jazyky, jejichž využití jinými firmami by tím bylo značně ztížené. Díky tomu je možné používat pro návrh v těchto jazycích simulátory, syntezátory, případně celá vývojová prostředí různých výrobců. Výsledkem toho je také vznik konkurenčního prostředí na poli výrobců těchto návrhových nástrojů.
- Některé jazyky lze používat pro návrh, simulaci a implementaci číslicového systému společně a využít tak naplno možností a vhodnosti použití jednotlivých jazyků pro jednotlivé části navrhovaného systému.

3.2 Použité formátování pro výpisy kódu a syntaxi jazyka VHDL

Syntaxe, výpisy kódu a příklady v jazyce VHDL jsou sázeny neproporcionálním písmem. Klíčová slova jazyka VHDL jsou vyznačena tučně. Komentáře jsou sázeny kurzívou. Pro výklad syntaxe je v textu použito následující formátování:

[]	volitelné
tučně	povinné
{ }	lze použít opakováně
	alternativa
()	seskupení
ID	identifikátor, návěští

3.3 Komentáře a identifikátory

Komentář v jazyce VHDL začíná **dvěma pomlčkami** a může začínat kdekoli na rádku. Komentář končí koncem řádku, na kterém začal. Komentář nelze jiným způsobem ukončit. Jazyk VHDL tak nepodporuje vícezádkové komentáře. V případě, že potřebujeme v jazyce VHDL komentovat celý úsek kódu, musíme komentovat každou řádku zvlášť. S většinou vyspělých editorů lze však toto omezení snadno řešit. Komentáře jsou také často využívané samotnými návrhovými systémy a syntezátory. Ve formě komentářů se tak například dají používat speciální příkazy pro ovládání chodu syntezátoru při syntéze zdrojového kódu. Tyto příkazy však nemají se syntaxí jazyka VHDL nic společného, liší se a jsou specifické pro každý jednotlivý syntezátor nebo návrhový systém určitého výrobce. Některé firmy zvolily jiné řešení a chod syntezátoru umožňují ovládat pomocí uživatelem definovaných atributů v jazyce VHDL. Ve formě komentářů si také do zdrojového textu ukládají některé návrhové systémy různá nastavení – např. při automaticky generovaném zdrojovém kódu, atd. Ve zdrojovém textu v jazyce VHDL se doporučuje nepoužívat české znaky, a to ani v komentářích, protože značné množství návrhových nástrojů nedokáže takový zdrojový text zpracovat.

Příklady použití komentářů:

```
-- toto je obyčejny komentar v jazyce VHDL  
-- nasledujici komentare jsou prikazy ovladajici syntezator pri synteze  
-- synopsys synthesis_off  
-- synopsys synthesis_on  
-- synthesis translate_off  
-- synthesis translate_on
```

Identifikátory slouží, podobně jako v ostatních jazycích, k pojmenování typů, proměnných, signálů a dalších objektů.

Pro identifikátory a návěští platí následující syntaxe:

```
pismeno{[_]pismeno_nebo_cislice}
```

Z uvedené syntaxe vyplývají následující omezení. Identifikátory se mohou skládat pouze z písmen, číslic a podtržítka. První znak identifikátoru musí být písmeno a poslední znak identifikátoru nesmí být podtržítko. Dvě podtržítka za sebou nejsou dovolena. Jako identifikátor nelze použít klíčová slova (viz podkapitola 3.14). Protože jazyk **VHDL** nerozlišuje **velká a malá písmena**, je například *rx_clk*, *RX_CLK*, *Rx_clk*, *Rx_Clk* tentýž identifikátor. Malá a velká písmena se však rozlišují v některých speciálních případech, např. v řetězcích nebo v označení hodnot u výčtových typů (např. 'X' nebo 'Z' u typu std_logic).

Příklady povolených identifikátorů:

```
rx_clk  
sel_8  
Ser2Par  
Bitova_Pozice
```

Příklady nepovolených identifikátorů:

```
_rx_clk      -- nezacina pismenem  
8B12B        -- nezacina pismenem  
datove#okno  -- obsahuje nepovoleny znak #
```

```
pocet_bitu -- obsahuje dve podtrzitka  
rx_clk_ -- konci podtrzitkem
```

3.4 Zápis čísel, znaků a řetězců

Čísla lze v jazyce VHDL zapisovat v několika soustavách a formátech. Jazyk VHDL rozlišuje čísla celá a čísla reálná. Čísla lze v jazyce VHDL zapisovat v dekadické soustavě, případně v dalších soustavách (např. dvojkové, osmičkové nebo šestnáctkové soustavě). Zvláštním způsobem lze v jazyce VHDL také zapisovat bitové řetězce v různých soustavách.

3.4.1 Zápis čísel v dekadické soustavě

Syntaxe zápisu čísel v dekadické soustavě:

```
cislo{[_]cislo}[.cislo{[_]cislo}][E[+|-]cislo]
```

Podtržitko mezi jednotlivými čísly nemá žádný vliv na hodnotu čísla. Lze ho používat k lepší čitelnosti zápisu čísla (např. k oddělování tisíců, apod). Tečka označuje desetinnou část čísla a písmeno E označuje exponent a lze ho psát i jako malé e. Exponent celých čísel nesmí být záporný.

Příklady:

```
112      0      1E5      567_890      -- cela cisla  
112.0    0.1    0.123    3.141_592_65  -- realna cisla  
112.2E-3 1.0E5  1.23E10   3.141_593E5  -- realna cisla s exponentem
```

3.4.2 Zápis čísel v dalších soustavách

Jazyk VHDL umožňuje zapisovat celá a reálná čísla ve dvojkové až šestnáctkové soustavě. Syntaxe zápisu je částečně podobná jako u zápisu čísel v dekadické soustavě. Jako první je třeba uvést dekadicky soustavu, v níž je číslo zapsáno, následovanou znakem #. Jako číslo lze (podle použité soustavy) použít číslice 0 až 9 a znaky A až F. Znaky A až F a znak exponentu lze psát i malými písmeny. Podtržitko mezi jednotlivými čísly nemá žádný vliv na hodnotu čísla a je určeno k lepší čitelnosti zápisu čísla. Tečka odděluje desetinnou část čísla. Písmeno E označuje exponent a lze ho psát i jako malé e. Exponent musí být, stejně jako soustava, zapsán dekadicky a nesmí být záporný. Exponentem je mocněna soustava v níž je číslo zapsáno.

Syntaxe zápisu čísel v soustavách:

```
soustava#cislo{[_]cislo}[.cislo{[_]cislo}]#[Ecislo]
```

Příklady:

```
2#1010_0101#     8#245#     16#A5#      -- cela cisla  
2#101_1010#E5    8#55#E2    16#B4#E1    -- cela cisla s exponentem  
2#1.0#E4         8#2.0#E1    16#1.0#E1    -- realna cisla s exponentem
```

3.4.3 Zápis znaku

Znaky se v jazyce VHDL, podobně jako v ostatních jazycích, zapisují mezi apostrofy.

Syntaxe zápisu znaku:

```
'z'
```

Příklady zápisu znaku:

```
'a' -- znak a  
'*' -- znak *  
' '' -- znak ' (apostrof)  
' ' -- znak mezera
```

3.4.4 Zápis textových řetězců

Textové řetězce se v jazyce VHDL, podobně jako v ostatních jazycích, zapisují mezi uvozovky.

Syntaxe zápisu řetězce:

```
"retezec"
```

Pokud se v řetězci má vyskytnout znak uvozovka ", musí se zapsat dvakrát za sebou. Prázdný řetězec neobsahuje mezi uvozovkami žádné znaky.

Příklady zápisu řetězce:

```
"VHDL je jazyk urcený pro popis hardware" -- retezec o 39 znacích  
"" -- prázdný řetězec  
" " -- řetězec obsahující jeden znak (mezera)  
"**" -- řetězec obsahující jeden znak (*)  
""" -- řetězec obsahující jeden znak ("")
```

Řetězce lze spojovat pomocí operátoru spojení & (angl. *concatenation*).

Příklad spojení řetězců:

```
"VHDL je jazyk urcený pro popis hardware" &  
"a je definovan normou IEEE Std 1076." -- jeden řetězec o 76 znacích
```

3.4.5 Zápis bitových řetězců

Bitové řetězce slouží k reprezentaci zápisu bitových polí ve dvojkové, osmičkové a šestnáctkové soustavě.

Syntaxe zápisu bitových řetězců:

```
znak_soustavy"cislo{[_]cislo}"
```

Znak soustavy může být **B** pro dvojkovou soustavu, **O** pro osmičkovou soustavu a **X** pro šestnáctkovou soustavu. Jako znaky bitového řetězce lze podle použité soustavy použít znaky 0 až 9 a znaky A až F. Znaky soustavy i bitového řetězce lze psát malými písmeny. Podtržítko mezi jednotlivými znaky nemá žádný vliv na hodnotu bitového řetězce a je určeno k lepší čitelnosti zápisu bitového řetězce.

Příklady zápisu bitového řetězce:

B"1111_1111_1111"	-- retezec o 12 znacích, ekvivalent retezci "111111111111"
O"7777"	-- retezec o 12 znacích, ekvivalent retezci "111111111111"
X"FFF"	-- retezec o 12 znacích, ekvivalent retezci "111111111111"
O"77_7"	-- retezec o 9 znacích, ekvivalent retezci "111111111"
X"77_7"	-- retezec o 12 znacích, ekvivalent retezci "011101110111"

3.5 Entita, architektura a další návrhové jednotky

Jednou z **primárních** (hlavních) návrhových jednotek v jazyce VHDL je tzv. **entita**. Entita popisuje rozhraní (vstupy a výstupy) objektu, který může představovat pouhé logické hradlo, celý obvod nebo velký systém. Entita také může obsahovat identifikátory, jimiž lze tuto entitu parametrizovat. Entitu lze přirovnat ke schematické značce, která pojmenovává vstupy a výstupy, definuje jejich typ a směr přenosu dat.

Syntaxe entity:

```
entity ENTITYID is
    [generic ({ID : TYPEID [:= výraz];});]
    [port ({ID : [in | out | inout | buffer | linkage] TYPEID [:= výraz];});]
    [{deklarace}]
    [begin
        {paralelni_prikazy}]
    end [entity] [ENTITYID];
```

Volitelná sekce **generic** umožňuje definovat tzv. generické konstanty entity. Hodnotu každé generické konstanty je pak možné zvenku entity, při každém použití entity jako komponenty (tzv. port mapu), měnit pomocí tzv. generického mapu. Generické konstanty lze s výhodou použít pro parametrizaci entity (např. pro nastavení její funkce, bitové šířky dat, operandů, atd). Více informací o použití entity jako komponenty a parametrizaci lze nalézt v podkapitolách 3.9.6. a 3.9.5.

Volitelná sekce **port** umožňuje definovat porty (vstupy a výstupy) entity. Každý port musí mít definováno své jméno (identifikátor), režim (angl. *mode*) přenosu a typ dat, která bude moci přenášet. Režim přenosu dat může být: **in**, **out**, **inout**, **buffer** a **linkage**. Režim přenosu dat nemusí být v deklaraci portu uveden. Pokud není směr uveden, je implicitně zvolen směr **in**.

- **in** – Port tohoto typu může být buzen pouze z vnějšku (data mohou tímto portem do entity pouze vstupovat). Režim **in** se používá nejčastěji pro vstupy, jako jsou např. hodiny, nulování, nastavování, povolování, datové vstupy, atd.
- **out** – Port tohoto typu může být buzen pouze z vnitřku entity (data mohou tímto portem z entity pouze vystupovat). Vlastní entita může port tohoto typu pouze budit, nikoliv však číst. Režim **out** se používá například pro výstupy, jako jsou např. řídicí signály, datové výstupy, třístanovkové výstupy, atd.
- **buffer** – Port tohoto typu může být buzen pouze z vnitřku entity (data mohou tímto portem z entity pouze vystupovat), vlastní entita však může výstupní data i zpětně číst (narozdíl od režimu **out**, ve kterém zpět číst nejdou). Režim **buffer** lze použít například pro výstupy čítačů, neboť stav výstupu čítače se musí uvnitř entity použít pro výpočet

následujícího stavu čítače (výstupní hodnotu je tedy třeba čist zpět). V některých případech (například při provádění časových simulací), nebo u některých návrhových systémů lze občas narazit na problémy s porty v režimu **buffer**. Řešením je deklarace nového vnitřního signálu, do něhož provedeme přiřazení namísto do portu (tentot signál pak lze bez problémů uvnitř entity dále čist). Problematický port pak deklarujeme v režimu **out** a nově vytvořený signál do tohoto portu uvnitř entity přiřadíme.

- **inout** – Port tohoto typu může být buzen signálem vnějším i vnitřním (data mohou tímto portem vstupovat i vystupovat). Režim **inout** se používá nejčastěji pro obousměrné vstupy / výstupy, jako jsou např. obousměrné datové sběrnice, atd.
- **linkage** – Port tohoto typu se v běžné praxi téměř nepoužívá. Navíc je pravděpodobné, že v příští revizi standardu jazyka VHDL bude tento režim ze standardu vypuštěn. Nebudem se jím tedy z výše uvedených důvodů zde dále zabývat.

Příklad deklarace:

```
entity scitacka is
  port (
    a : in std_logic;      -- 1. scitanec
    b : in std_logic;      -- 2. scitanec
    cin : in std_logic;    -- vstup prenosu (carry in)
    sum : out std_logic;   -- součet
    cout : out std_logic;  -- vystup prenosu (carry out)
  );
end;
```

V deklaraci portů entity je použit typ **std_logic**. Typ **std_logic** je výčtový typ, který může nabývat celkem 9 hodnot. Kromě klasických logických hodnot '0' a '1' může tento typ nabývat také hodnot 'U', 'X', 'Z', 'W', 'L', 'H', '-'). Více informací o tomto typu lze nalézt v podkapitole 3.12., věnující se knihovnám a knihovním balíkům.

Architektura definuje vlastní chování a funkci entity (např. vztahy mezi porty entity). Architektura tedy definuje vnitřek entity. Architektura je tzv. **sekundární** (závislá) návrhová jednotka. Každá entita musí mít alespoň jednu architekturu. Ke každé entitě tedy lze definovat více architektur. Každá architektura stejně entity pak ale musí mít jiné jméno (identifikátor). Protože má jazyk VHDL velmi bohaté vyjadřovací schopnosti, lze většinou popsat tutéž funkci architektury různými styly zápisu. Jazyk VHDL umožňuje tyto styly popisu architektury – popis **behaviorální** (chování), popis **toku dat** (dataflow), popis **strukturální** a **RTL** popis. Stylovým popisu se věnuje kapitola 15.

Syntaxe architektury:

```
architecture ARCHID of ENTITYID is
  [{definice_a_deklarace}]
begin
  [{paralelni_prikazy}]
end [architecture] [ARCHID];
```

Následující úsek kódu ukazuje tělo architektury, které patří k entitě z předchozího příkazu. Kód realizuje úplnou jednobitovou sčítačku. V kódu jsou použity logické operátory **and**, **or** a **xor**. Operátory jazyka VHDL je věnována podkapitola 3.7. Symbol **<=** se v jazyce VHDL používá pro přiřazení do signálu.

```

architecture a_scitacka of scitacka is
begin
    -- realizace uplné scítacky
    -- součet
    sum <= a xor b xor cin;
    -- přenos
    cout <= (a and b) or (cin and (a xor b));
end;

```

Základní zdrojový text v jazyce VHDL se sestává z deklarace entity a příslušné architektury (případně více architektur). Jméno souboru, ve kterém je uložen zdrojový text entity, by mělo být stejně jako jméno této entity. V souboru by měla být definována pouze jedna entita a příslušná architektura nebo architektury, pokud jich má mít entita definováno více. Celý zdrojový text realizující úplnou jednobitovou sčítáčku pak ukazuje následující výpis kódu. Jméno souboru by mělo být **scitacka.vhd** nebo **scitacka.vhdl**.

```

library ieee;
use ieee.std_logic_1164.all;

entity scitacka is
port (
    a : in std_logic;      -- 1. scitanec
    b : in std_logic;      -- 2. scitanec
    cin : in std_logic;    -- vstup přenosu (carry in)
    sum : out std_logic;   -- součet
    cout : out std_logic   -- výstup přenosu (carry out)
);
end;

architecture a_scitacka of scitacka is
begin
    -- realizace uplné scítacky
    -- součet
    sum <= a xor b xor cin;
    -- přenos
    cout <= (a and b) or (cin and (a xor b));
end;

```

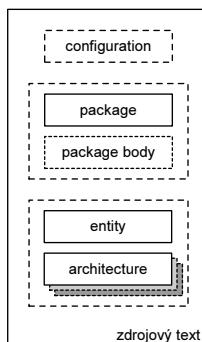
Výpis 3.1 Úplný zdrojový text realizující v jazyce VHDL úplnou sčítáčku

První řádka výpisu kódu s klíčovým slovem **library** připojuje knihovnu s názvem *ieee*. Druhá řádka s klíčovým slovem **use** pak připojuje z této knihovny vše, co je definováno v knihovním balíku s názvem *std_logic_1164*. Připojení všeho, co je v balíku definováno, je dáno použitím klíčového slova **all**. Knihovní balík *std_logic_1164* je připojován, protože je v příkladu použit typ **std_logic** pro vstupy a výstupy entity. Tento balík definuje typ **std_logic** a také definuje nad tímto typem veškeré operace (např. v architektuře použité logické operátory **and**, **or** a **xor**). Knihovní balíky se definují pomocí klíčových slov **package** a **package body**. V bloku uvozeném klíčovým slovem **package** se provádí deklarace obsahu knihovního balíku (jedná se o primární návrhovou jednotku). Tělo knihovního balíku je definováno klíčovými slovy **package body**. Tělo knihovního balíku je, podobně jako architektura, sekundární návrhovou jednotkou. Narozdíl od architektury je však deklarace těla nepovinná

(nemusí tedy být deklarováno) a pokud je deklarováno, tak pouze jedno. Každý knihovní balík tak může mít pouze jedno tělo. Tělo knihovního balíku nemusí následovat bezprostředně za deklarací a může být dokonce umístěno i v jiném souboru. Knihovnám a knihovním balíkům se podrobněji věnuje podkapitola 3.12.

Poslední návrhovou jednotkou používanou v jazyce VHDL je tzv. konfigurační deklarace. Konfigurace se definuje pomocí klíčového slova **configuration**. Konfigurací lze např. určit která z architektur se použije v případě, že má entita definováno více architektur. Konfigurační deklarace je nepovinná a zdrojový text v jazyce VHDL ji tedy nemusí obsahovat. V případě, že entita má definováno více architektur a konfigurační deklarace neexistuje, použije se pro simulaci nebo syntézu poslední architektura.

Vztahy primárních a sekundárních (závislých) návrhových jednotek ukazuje obr. 3.1.



Obr. 3.1 Vztahy primárních a sekundárních návrhových jednotek

Zdrojové texty návrhových jednotek lze kompilovat nezávisle na sobě a lze je ukládat do knihoven.

3.6 Základní datové typy

Jazyk VHDL má několik základních datových typů, z nichž většina je skalárních a několik kompozitních.

Mezi **skalární** typy patří typ:

- výčтовý
- celočíselný
- fyzický
- s plovoucí řádovou čárkou

Mezi **kompozitní** typy patří typ:

- pole
- záznam

Speciálním typem je pak typ **soubor**. Jazyk VHDL umožňuje, podobně jako ostatní jazyky vysoké úrovně, uživateli definovat vlastní datové typy. Standard jazyka VHDL-93 má několik datových typů **předdefinovaných**. Všechny předdefinované datové typy, uváděné v následujících podkapitolách, jsou deklarovány v knihovně *std* v balíku *standard*. Knihovní balíky jsou v české terminologii někdy označovány jako **slohy**. Knihovna *std* je vždy automaticky přístupná a není ji tedy třeba nijak připojovat. Jazyk VHDL je typově velmi striktní, není tedy možné spolu míchat různé datové typy při matematických operacích, přiřazených a podobně. Pokud potřebujeme spolu různé datové typy nějakým způsobem kombinovat, např. v matematických operacích, přiřazených a podobně, musíme je nejdříve buď **přetykovat**, nebo použít příslušnou **konverzní funkci** (pokud existuje), měnící jeden typ na druhý. Konverzní funkce jsou zpravidla definovány v knihovních balících, ve kterých jsou definovány příslušné typy.

3.6.1 Výčtový typ

Výčtový typ patří mezi diskrétní skalární typy. Protože se jedná o skalární typ, jsou pro něj definovány veškeré relační operátory. Hodnoty, jichž může výčtový typ nabývat, jsou definovány mezi závorkami při deklaraci typu. Protože se jedná navíc o diskrétní typ, lze pro každou jeho hodnotu určit i předchozí a následující hodnotu. Hodnota, uvedená ve výčtu jako první zleva, má nejnižší hodnotu. Hodnota, uvedená nejvíce vpravo, má pak hodnotu nejvyšší.

Syntaxe výčtového typu:

```
type TYPEID is (ID {, ID});
```

Předdefinované výčtové typy v knihovním balíku *std.standard*:

```
bit, boolean, character, file_open_kind, file_open_status, severity_level
```

Definice předdefinovaných výčtových typů v knihovním balíku *std.standard*:

```
type bit is ('0', '1');
type boolean is (false, true);
type file_open_kind is (read_mode, write_mode, append_mode);
type file_open_status is (open_ok, status_error, name_error, mode_error);
type severity_level is (note, warning, error, failure);
```

Typ **bit** tedy podle své definice může nabývat hodnot '0' a '1'. Lze s ním tedy například reprezentovat běžné dvouhodnotové číslicové signály. Typ **boolean** pak definuje hodnoty **false** a **true**, atd. Výčtový typ **character** definuje 256 hodnot (znaků). Znaky typu **character** jsou kódovány jako osmibitové dle normy ISO 8859-1:1987 (Latin-1). Definici předdefinovaného typu **character** zde tedy neuvádíme.

Příklady použití:

```
-- definice
type jidlo is (snidane, obed, svacina, vecere);
type std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

-- deklarace
signal a : jidlo;
signal b : std_logic;
```

```

signal c : bit;
signal d : boolean;

-- pouziti
a <= svacina;
b <= 'Z';
c <= '1';
d <= false;

```

3.6.2 Celočíselný typ

Definice celočíselného typu definuje celočíselný rozsah hodnot, jichž může tento typ nabývat. Standard jazyka VHDL definuje, že celočíselný typ musí mít minimálně 32bitovou implementaci. Celočíselný typ tedy musí vždy podporovat minimální rozsah $-(2^{31} - 1)$ až $+(2^{31} - 1)$ tj. $-2\ 147\ 483\ 647$ až $+2\ 147\ 483\ 647$. Operátorům, jež jsou definovány pro celočíselný typ, se věnují podkapitoly 3.7.2., 3.7.4., 3.7.5. a 3.7.6.

Syntaxe definice celočíselného typu:

```
type ID is range cele_cislo downto | to cele_cislo;
```

Předdefinované celočíselné typy v knihovním balíku std.standard:

```

integer má definován rozsah  $-2\ 147\ 483\ 647$  až  $+2\ 147\ 483\ 647$  včetně
natural má definován rozsah 0 až  $+2\ 147\ 483\ 647$  včetně
positive má definován rozsah 1 až  $+2\ 147\ 483\ 647$  včetně

```

Definice předdefinovaných celočíselných typů v knihovním balíku std.standard:

```

type integer is range -2147483647 to 2147483647
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;

```

Ve výše uvedených definicích je použit příkaz **'high** a **subtype**. Apostrof a následující slovo (zde **high**) značí atribut jazyka VHDL. Atribut **'high** vrací jako návratovou hodnotu maximální rozsah typu, u něhož byl použit. Ve výše uvedených dvou případech tedy vrací hodnotu 2 147 483 647. Atributům se věnuje podrobněji podkapitola 3.11. Pomocí příkazu **subtype** se definuje podtyp odvozený od určitého typu.

Příklady použití:

```

-- definice
type byte is range 0 to 255;
type sig16 is range -32768 to 32767;
type index is range 31 downto 0;

-- deklarace
signal a : byte;
signal b : sig16;
signal c : index;
signal d : integer;

-- pouziti
a <= 168;
b <= -28741;
c <= 23;
d <= -386544278;

```

3.6.3 Fyzický typ

Fyzický typ je určen k vyjádření množství vztaženého k základní jednotce. Hodnota fyzického typu je tedy celočíselný násobek základní jednotky fyzického typu. Základní jednotka definuje rozlišovací schopnost daného fyzického typu. Jakkoliv hodnota menší než základní jednotka je oříznuta na hodnotu 0 základní jednotky. Standard jazyka VHDL definuje, že fyzický typ musí mít minimálně 32bitovou implementaci. Fyzický typ tedy musí vždy podporovat minimální rozsah $-(2^{31} - 1)$ až $+(2^{31} - 1)$ tj. $-2\ 147\ 483\ 647$ až $+2\ 147\ 483\ 647$. Pro fyzický typ jsou definovány stejné operátory jako pro celočíselný typ.

Syntaxe definice fyzického typu:

```
type ID is range cislo downto | to cislo
  units
    ZAKLADNI_JEDNOTKA;
    {DALSI_JEDNOTKA;};
end units [ID];
```

Předdefinované fyzické typy v knihovním balíku *std.standard*:

```
time má definován rozsah -2 147 483 647 až +2 147 483 647 včetně
delay_length má definován rozsah 0 až +2 147 483 647 včetně
```

Základní jednotkou typu **time** je femtosekunda **fs** ($1\ \text{fs} = 10^{-15}\ \text{s}$).

Definice předdefinovaných fyzických typů v knihovním balíku *std.standard*:

```
type time is range -2147483647 to 2147483647
  units
    fs;          -- femtosekunda
    ps = 1000 fs; -- pikosekunda
    ns = 1000 ps; -- nanosekunda
    us = 1000 ns; -- mikrosekunda (mikro se píše jako u)
    ms = 1000 us; -- milisekunda
    sec = 1000 ms; -- sekunda
    min = 60 sec; -- minuta
    hr = 60 min; -- hodina
  end units;

  subtype delay_length is time range 0 fs to time'high;
```

Implementace typu **time** je závislá na výrobci simulátoru. Standard jazyka VHDL definuje, že typ **time** musí mít minimálně 32bitovou implementaci. Většina výrobců simulátorů však implementuje typ **time** (z důvodu podpory delších simulačních časů) pomocí 64bitové implementace, takže má pak většinou definován rozsah jako $-(2^{63} - 1)$ až $+(2^{63} - 1)$. V knihovním balíku *std.standard* je definována například funkce **now**, která vrádí hodnotu současného simulačního času (typ **delay_length**).

Příklady použití:

```
-- definice
type delka is range 0 to 1E18
  units
    A; -- Angstrom
    nm = 10 A; -- nanometr
    um = 1000 nm; -- mikrometr
    mm = 1000 um; -- milimetr
```

```

cm = 10 mm;      -- centimetr
dm = 10 cm;      -- decimetr
m = 10 dm;       -- metr
end units;

-- deklarace
signal a : delka;
signal b : integer;
signal c : delay_length;

-- pouziti
a <= 3 cm - 7 A + 2 mm;
b <= cm / mm;
c <= now;      -- naplni c aktualni hodnotou simulacniho casu

```

3.6.4 Typ s plovoucí řádovou čárkou

Definice typu s plovoucí řádovou čárkou definuje rozsah hodnot, jichž může tento typ nabývat. Vlastní implementace typu s plovoucí řádovou čárkou je závislá na výrobci simulátoru nebo syntezátoru. Je třeba mít na paměti, že současné syntezátory většinou neumí provést syntézu matematických operací s plovoucí řádovou čárkou do hardware.

Syntaxe definice typu s plovoucí řádovou čárkou:

```
type ID is range desetinne_cislo downto | to desetinne_cislo;
```

Předdefinovaný typ s plovoucí řádovou čárkou v knihovním balíku *std.standard*:

```
real má definován rozsah  $-1,0 \cdot 10^{308}$  až  $+1,0 \cdot 10^{308}$  včetně
```

Standard jazyka VHDL definuje, že typ **real** musí podporovat přesnost na minimálně šest desetinných míst a minimální rozsah $-1,0 \cdot 10^{38}$ až $+1,0 \cdot 10^{38}$ včetně. Vlastní vnitřní implementace typu s plovoucí řádovou čárkou je závislá na výrobci simulátoru nebo syntezátoru. Čísla s plovoucí řádovou čárkou jsou výrobci simulátorů a syntezátorů implementována dle standardu *IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985)* případně dle standardu *IEEE Standard for Radix-Independent Floating-Point Arithmetic (IEEE Std 854-1987)*.

Definice typu s plovoucí řádovou čárkou v knihovním balíku *std.standard*:

```
type real is range -1.0E308 to 1.0E308
```

Příklady použití:

```

-- definice
type float is range -1.0E38 to 1.0E38;

-- deklarace
signal a : float;
signal obvod : real;

-- pouziti
a <= 0.123456E10;
obvod <= 2 * 3.141592654 * 34;

```

3.6.5 Typ pole

Typ pole patří mezi kompozitní datové typy (skládá se z více prvků). Pole je datová struktura, složená z více **stejných** prvků (elementů). Význam a práce s poli jsou v jazyce VHDL podobné jako v jiných programovacích jazycích. Index prvku, k němuž přistupujeme, se uvádí v závorkách. Jazyk VHDL podporuje **vícerozměrná** pole.

Syntaxe definice typu pole:

```
type ID is array ({rozsah | TYPEID,}) of TYPEID;
```

Element rozsah má následující syntaxi:

```
rozsah (cislo | ENUMID downto | to cislo | ENUMID) |  
OBJID[reverse_]range | TYPEID range <>
```

Předdefinované typy pole v knihovním balíku std.standard:

```
bit_vector je definován jako jednorozměrné pole typu bit  
string je definován jako jednorozměrné pole typu character
```

Definice typů pole v knihovním balíku std.standard:

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

Symbol $\langle\rangle$ v definici rozsahu pole znamená, že rozsah pole není při definici určen (omezen).

Příklady použití:

```
-- definice  
type slovo is array (15 downto 0) of bit;  
type pamet128x16 is array (0 to 127) of slovo;  
type pamet_x16 is array (natural range <>) of slovo;  
  
-- deklarace  
signal a, b : slovo;  
signal ram1 : pamet128x16;  
signal ram2 : pamet_x16(0 to 1023);  
signal retezec : string(1 to 4);  
  
-- použití  
a <= "0110001101110011";  
b(15) <= a(0);  
ram2(76) <= ram1(72);  
retezec <= "AHOJ";
```

3.6.6 Typ záznam

Typ záznam (**record**) patří stejně jako pole mezi kompozitní typy. Na rozdíl od pole, které patří mezi homogenní datové typy (všechny jeho prvky jsou stejného typu), může záznam obsahovat prvky **různých** typů (záznam je heterogenní datový typ). Práce v jazyce VHDL s typem záznam je podobná jako v jiných programovacích jazycích. Přistupovat lze buď k jednotlivým prvkům záznamu, a nebo k záznamu jako celku.

Syntaxe definice typu záznam:

```
type ID is record
    ID : TYPEID;
    {ID : TYPEID;}
end record [ID];
```

Příklady použití:

```
-- definice
type datum is record
    den : integer range 1 to 31;
    mesic : integer range 1 to 12;
    rok : integer range 0 to 3000;
end record;

-- deklarace
signal den_a, den_b, den_c : datum;

-- pouziti
den_a.den <= 11;           -- priazeni hodnot do jednotlivych prvku zaznamu
den_a.mesic <= 1;
den_a.rok <= 2006;
den_b.rok <= den_a.rok;   -- priazeni prvku jednoho zaznamu do druhoho
den_c <= den_a;           -- priazeni celeho zaznamu do druhoho
```

3.6.7 Typ soubor

Typ soubor (**file**) slouží ve VHDL k přístupu k souborům v režimu pro čtení nebo pro zápis.

Syntaxe definice typu soubor:

```
type FTYPEID is file of TYPEID;
```

Pro typ file jsou implicitně definovány následující procedury a funkce:

```
procedure file_open (file f : FTYPEID; external_name : in string;
                     open_kind : in file_open_kind := read_mode);

procedure file_open (status : out file_open_status; file f : FTYPEID;
                     external_name : in string;
                     open_kind : in file_open_kind := read_mode);

procedure file_close (file f : FTYPEID);

procedure read (file f : FTYPEID; value : out TYPEID);

procedure write (file f : FTYPEID; value : in TYPEID);

function endfile (file f : FTYPEID) return boolean;
```

Procedura **file_open** slouží k otevření souboru pro čtení (**read_mode**), zápis (**write_mode**) nebo zápis na konec již existujícího souboru (**append_mode**). Procedura **file_open** je před-definovaná jako přetížená (její deklarace existuje ve dvou verzích). Lze ji volat buď bez prvního parametru, který je typu **file_open_status** a nebo s ním. Pokud je procedura **file_open** s tímto parametrem zavolána, pak funkce do tohoto parametru uloží stav otevření. Stav otevření

je výčtový typ, jež může nabývat hodnot **open_ok**, **status_error**, **name_error** a **mode_error**. Procedura **file_close** slouží k zavření souboru. Případné neuzavřené soubory uzavře simulátor při ukončení simulátoru, doporučuje se však všechny otevřené soubory uzavřít. Procedury **read** a **write** slouží ke čtení a zápisu dat z/do souboru. Funkce **endfile** slouží ke zjišťování, zda jsme narazili na konec souboru. Funkce vrací booleovskou hodnotu **true**, pokud jsme na konci souboru.

Příklady definice typu soubor:

```
-- definice
type tf_word is file of integer;
type tf_text is file of string;
```

Deklaraci a použití typu soubor se dále věnuje podkapitola 3.8.5. Typ soubor (**file**) není syntetizovatelný a je tedy použitelný pouze pro simulaci na simulátoru jazyka VHDL.

3.7 Operátory

Operátory, které jazyk VHDL definuje, lze rozdělit do několika tříd – viz tab. 3.1. Třídy operátorů jsou v tabulce uvedeny se vzrůstající prioritou (shora dolů). Nejnižší prioritu mají logické operátory, nejvyšší prioritu pak má operátor mocniny ******, absolutní hodnoty **abs** a logický operátor **not**. V rámci třídy mají všechny operátory stejnou prioritu. Znak *L* značí levý operand, znak *P* značí pravý operand.

Tab. 3.1 Operátory jazyka VHDL (nejnižší prioritu mají logické operátory)

logické op.	<i>L and P</i>	<i>L or P</i>	<i>L nand P</i>	<i>L nor P</i>	<i>L xor P</i>	<i>L xnor P</i>
relační op.	<i>L = P</i>	<i>L /= P</i>	<i>L < P</i>	<i>L <= P</i>	<i>L > P</i>	<i>L >= P</i>
op. posuvu	<i>L sll P</i>	<i>L srl P</i>	<i>L sla P</i>	<i>L sra P</i>	<i>L rol P</i>	<i>L ror P</i>
op. scítání	<i>L + P</i>	<i>L - P</i>	<i>L & P</i>			
znaménkové op.	<i>+P</i>	<i>-P</i>				
násobící op.	<i>L * P</i>	<i>L / P</i>	<i>L mod P</i>	<i>L rem P</i>		
různé op.	<i>L ** P</i>	abs P	not P			

3.7.1 Logické operátory

V jazyce VHDL jsou definovány logické operátory **and**, **or**, **nand**, **nor**, **xor**, **xnor** a **not**. Funkce těchto operátorů je zřejmá z jejich názvů. Tyto operátory jsou definovány pro předdefinované typy **bit** a **boolean**. Tyto operátory jsou rovněž definovány i pro jednorozměrná pole typu **bit_vector**. V knihovním balíku **std_logic_1164** (viz podkapitola 3.12.1) jsou definovány tyto operátory také pro typy **std_ulogic** a **std_logic** a pro jednorozměrná pole těchto dvou typů (tj. pro typy **std_ulogic_vector** a **std_logic_vector**). Vyjma operátoru **not**, který má nejvyšší prioritu, mají všechny zbývající logické operátory stejnou prioritu, a to nejnižší. V jazyce VHDL je tedy třeba při psaní logických výrazů dát na tuto vlastnost zvýšený pozor a používat závorek, jimiž se určí pořadí vyhodnocení výrazu explicitně. Jazyk VHDL vyhodnocuje výrazy, tak jako většina jazyků, zleva doprava.

Příklad:

```
-- deklarace
signal a1, a2, a3, b, c, d, e ,f : bit;
signal pole_a, pole_b, pole_c, pole_d : bit_vector (7 downto 0);

-- priklad pouziti
a1 <= b or c and d or e;    -- odpovida výrazu ((b or c) and d) or e
a2 <= b or c or d and e;    -- odpovida výrazu ((b or c) or d) and e
a3 <= b and c or d and e;   -- odpovida výrazu ((b and c) or d) and e
f <= not b;
pole_a <= pole_b and pole_c;
pole_d <= not pole_b;
```

3.7.2 Relační operátory

V jazyce VHDL jsou definovány tyto relační operátory $=$, \neq , $<$, \leq , $>$, \geq . Operátory rovnosti a nerovnosti ($=$ a \neq) jsou definovány pro všechny typy mimo typu soubor. Operátor rovnosti vrací booleovskou hodnotu **true**, pokud jsou oba operandy shodné, jinak vrací hodnotu **false**. Podobně je tomu i u dalších operátorů, všechny vrací booleovskou hodnotu **true**, pokud je relační podmínka splněna a hodnotu **false**, pokud splněna není. Operátory menší, menší nebo rovno, větší a větší nebo rovno ($<$, \leq , $>$, \geq) jsou definovány pro všechny skalární typy a pro jednorozměrná pole, jejichž prvky jsou diskrétního typu.

Příklad:

```
-- deklarace
signal a, b : integer;
signal rov, ruz, men, vet : boolean;

-- priklad pouziti
rov <= a = b;
ruz <= a /= b;
men <= a <= b;
vet <= a >= b;
```

3.7.3 Operátory posuvu

Operátory posuvu **sll**, **srl**, **sla**, **sra**, **rol** a **ror** jsou definovány pro jednorozměrná pole, jejichž prvky jsou předdefinovaného typu **bit** nebo **boolean**. Levý operand operátoru posuvu musí být jednorozměrné pole typu **bit** nebo **boolean**. Pravý operand operátoru posuvu musí být celočíselný typ **integer**. Výsledkem operace posuvu je stejný typ, jako je typ levého operantu. Pokud je pravý operand roven 0, vrací tyto operátory hodnotu levého operantu.

- $L \text{ sll } P$ – provádí logický posuv L vlevo o P pozic. Pokud má pravý operand zápornou hodnotu, vrátí operátor hodnotu, jako kdyby se vykonával výraz $L \text{ srl } -P$.
- $L \text{ srl } P$ – provádí logický posuv L vpravo o P pozic. Pokud má pravý operand zápornou hodnotu vrátí operátor hodnotu, jako kdyby se vykonával výraz $L \text{ sll } -P$.
- $L \text{ sla } P$ – provádí aritmetický posuv L vlevo o P pozic. Pokud má pravý operand zápornou hodnotu, vrátí operátor hodnotu, jako kdyby se vykonával výraz $L \text{ sra } -P$.

- $L \text{ sra } P$ – provádí aritmetický posuv L vpravo o P pozic. Pokud má pravý operand zápornou hodnotu, vrátí operátor hodnotu, jako kdyby se vykonával výraz $L \text{ sla } -P$.
- $L \text{ rol } P$ – provádí rotaci L vlevo o P pozic. Pokud má pravý operand zápornou hodnotu, vrátí operátor hodnotu, jako kdyby se vykonával výraz $L \text{ ror } -P$.
- $L \text{ ror } P$ – provádí rotaci L vpravo o P pozic. Pokud má pravý operand zápornou hodnotu, vrátí operátor hodnotu, jako kdyby se vykonával výraz $L \text{ rol } -P$.

Příklad:

```
-- deklarace
signal a : bit_vector (7 downto 0) := "10000001";
signal b, c, d, e, f, g : bit_vector (7 downto 0);

-- priklad pouziti
b <= a sll 1; -- vysledek: "00000010"
c <= a srl 1; -- vysledek: "01000000"
d <= asla 2; -- vysledek: "00000111"
e <= asra 2; -- vysledek: "11100000"
f <= arol 1; -- vysledek: "00000011"
g <= aror 1; -- vysledek: "11000000"
```

3.7.4 Sčítací operátory a operátor spojení

Mezi sčítací operátory patří operátor sčítání a odčítání ($+ -$). Tyto operátory jsou definovány pro všechny číselné typy. Operátor spojení $\&$ je definován pro jednorozměrná pole (tj. např. i pro veškeré řetězce). Pro operátor sčítání a odčítání platí, že levé a pravé operandy musí být stejného typu a stejného typu je i výsledek operace. Pro levý a pravý operand operátoru spojení mohou nastat čtyři možnosti:

- Oba operandy jsou jednorozměrná pole stejného typu, stejné nebo různé délky. Výsledkem operace bude jednorozměrné pole stejného typu o délce rovné součtu délek obou operandů. Prvky tohoto pole sestávají z prvků levého operandu, následované prvky pravého operandu (při pohledu zleva doprava).
- Levý operand je jednorozměrné pole a pravý operand je jeden prvek stejného typu jako jsou prvky pole levého operandu. Výsledkem operace je jednorozměrné pole stejného typu jako původní pole, které má zprava přidán jeden prvek (při pohledu zleva doprava).
- Levý operand je jeden prvek stejného typu jako jsou prvky pole pravého operandu a pravý operand je jednorozměrné pole. Výsledkem operace je jednorozměrné pole stejného typu jako původní pole, které má zleva přidán jeden prvek (při pohledu zleva doprava).
- Levý i pravý operand jsou prvky stejného typu. Výsledkem je jednorozměrné pole o dvou prvcích stejného typu jako operandy, přičemž prvek levého operandu se nachází vlevo a prvek pravého operandu se nachází vpravo (při pohledu zleva doprava).

Příklad použití:

```
-- deklarace
signal a : bit_vector (7 downto 0) := "10000001";
signal b, c, d : bit_vector (7 downto 0);
signal e : bit_vector (1 downto 0);
```

```
-- priklad pouziti
b <= a(3 downto 0) & a(7 downto 4); -- vysledek: "00011000"
c <= a(6 downto 0) & '0';           -- vysledek: "00000010"
d <= '0' & a(7 downto 1);          -- vysledek: "01000000"
e <= '1' & '0';                  -- vysledek: "10"
```

3.7.5 Znaménkové operátory

Znaménka + a - mají normální matematický význam (tj. funkci identity a negace) a jsou definovány pro všechny číselné typy. Výsledek operace má stejný typ jako operand. Protože mají znaménkové operátory menší prioritu než násobící operátory (*, /, **mod** a **rem**) a operátory **abs** a ******, musí se v takových případech vhodně použít závorky.

Příklad použití:

```
-- deklarace
signal a, b, c, d, e : integer;

-- priklad pouziti
a <= b / (-c);
d <= -c * (-b);
e <= b / -c;    -- neplatny vyraz !!! nebyly pouzity zavorky !!!
```

3.7.6 Násobící operátory

Operátor násobení a dělení (* a /) jsou definovány pro celočíselný typ a typ s plovoucí řádovou čárkou. Operátory dělení modulo (**mod**) a zbytek po dělení (**rem**) jsou definovány pouze pro celočíselné typy. V případě použití operátorů této třídy pro celočíselný typ a typ s plovoucí řádovou čárkou platí, že operandy musí být stejného typu a stejného typu je i výsledek. Operátor násobení a dělení je také definován pro všechny fyzické typy. V případě použití fyzického typu jako jednoho z operandů násobení nebo dělení platí následující omezení:

- V případě operátoru násobení platí, že pokud je jako jeden z operandů (lhostejno jestli jako levý nebo pravý operand) použit fyzický typ, druhý operand musí být celočíselný typ nebo typ s plovoucí čárkou. To znamená že, fyzický typ lze násobit pouze s celočíselným typem nebo s typem s plovoucí řádovou čárkou. Výsledkem takové operace je stejný fyzický typ. Nelze tedy násobit fyzický typ s fyzickým typem (ani když jsou oba operandy stejného fyzického typu), vznikal by totiž nový typ (fyzický typ na druhou), a to je nepřípustné.
- V případě operátoru dělení se musí fyzický typ použít jako levý operand. Pravý operand může být celočíselného typu, typu s plovoucí řádovou čárkou nebo stejného typu jako levý operand. V případě použití celočíselného typu a typu s plovoucí řádovou čárkou má pak výsledek operace stejný fyzický typ jako levý operand. V případě dělení fyzického typu fyzickým typem má výsledek celočíselný typ.

3.7.7 Různé operátory

Nejvyšší prioritu mají v jazyce VHDL operátory mocniny ******, absolutní hodnoty **abs** a logický operátor **not**. Funkci logického operátoru **not** jsme se věnovali již v podkapitole

3.7.1. Operátor absolutní hodnoty je definován pro všechny číselné typy. Výsledkem je stejný typ jako typ operandu. Operátor mocniny vyžaduje jako levý operand celočíselný typ nebo typ s plovoucí rádovou čárkou, pravý operand musí být typu **integer**. Výsledek má stejný typ jako levý operand. Exponent může nabývat záporné hodnoty pouze v případě, že je levý operand typem s plovoucí rádovou čárkou.

3.8 Základní objekty

Mezi základní objekty jazyka VHDL patří konstanty, signály, proměnné, aliasy a soubořy. Objekty lze deklarovat na místech k tomu určených, tj. v deklaračních částech architektury, procesů, bloků, atd. Podobně jako v jiných jazycích, lze deklarovat jeden nebo více objektů stejného typu najednou (při deklaraci jsou od sebe jednotlivé identifikátory odděleny čárkou). Každému objektu lze rovnou při deklaraci přiřadit implicitní hodnotu.

3.8.1 Konstanty

Hodnotu konstanty nelze dále v kódu měnit (identifikátor konstanty se tedy může vyskytovat pouze na pravé straně přiřazení).

Syntaxe deklarace konstanty:

```
constant ID : TYPEID := vyraz;
```

Příklady:

```
constant pi : real := 3.141592;
constant sirka : integer := 8;
constant tclk : time := 12.5 ns;
```

3.8.2 Signály

Signály slouží k přenosu dat mezi jednotlivými komponentami, apod. Funkci signálů lze přirovnat k funkci vodičů v číslicovém systému. Signály lze deklarovat například v deklarační části architektury.

Syntaxe deklarace signálu:

```
signal ID : TYPEID [:= vyraz];
```

Příklady:

```
-- deklarace
signal polomer, obvod : real;
signal sirka : integer := 10;
signal th : time;

-- pouziti
polomer <= 25;
obvod <= 2 * pi * polomer;
sirka <= sirka + 1;
th <= 10 ns;
```

Přiřazení do signálu se provádí pomocí symbolu `<=` za nímž následuje výraz. Implicitní přiřazení do signálu (při deklaraci) se však provádí pomocí symbolu `:=` podobně jako u přiřazení do proměnné (viz dále).

3.8.3 Proměnné

Nesdílenou proměnnou (bez **shared**) lze deklarovat pouze uvnitř procesu (v jeho deklarační části), sdílenou proměnnou (**shared variable**) lze deklarovat pouze mimo proces. Sdílenou proměnnou tedy lze, podobně jako signály, deklarovat například v deklarační části architektury.

Syntaxe deklarace proměnné:

```
[shared] variable ID : TYPEID [:= vyraz];
```

Příklady:

```
-- deklarace v deklarační casti architektury
shared variable vlajka : boolean;

-- deklarace v deklarační casti procesu
variable citac : integer range 0 to 199 := 0;
variable pamet : array (0 to 127) of integer;

-- použití uvnitř procesu
citac := citac + 1;
pamet(8) := -20;
vlajka := '1';
```

Přiřazení do proměnné se provádí pomocí symbolu `:=` za nímž následuje výraz. Přiřazovat do proměnné lze pouze uvnitř procesů a při jejich deklaraci.

3.8.4 Aliasy

Alias není objekt v pravém slova smyslu. Nejedná se totiž o nový objekt, ale pouze o alternativní identifikátor na již existující objekt nebo jeho část. Použití aliasu umožňuje zjednodušit zápis kódu a zpřehlednit ho.

Syntaxe deklarace aliasu:

```
alias ID [: TYPEID] is OBJID;
```

Příklady:

```
-- deklarace
signal adr : std_logic_vector (31 downto 0);
alias top_adr : std_logic_vector (3 downto 0) is adr(31 downto 28);
alias bank : std_logic_vector (3 downto 0) is adr(27 downto 24);
alias row : std_logic_vector (11 downto 0) is adr(23 downto 12);

-- použití
top_adr <= "1000";
bank <= "1001";
row <= "111100001110";
```

3.8.5 Soubory

Jak již bylo řečeno v podkapitole 3.6.7, existuje v jazyce VHDL typ **file** umožňující přístup k souborům. Definici typu **file** lze definovat datový typ, jenž je nebo bude v souboru uložen (viz kap. 3.6.7). Pro přístup k souboru je třeba dále provést deklaraci tzv. rukověti k souboru, pomocí níž lze otevřít konkrétní soubor pro čtení nebo zápis.

Syntaxe deklarace souboru:

```
file ID : FTYPED [ [open read_mode | write_mode | append_mode] is "jmeno"];
```

Příklady:

```
-- definice
type tf_word is file of integer;
type tf_text is file of string;

-- deklarace
file soubor1 : tf_word open write_mode is "soubor1.bin";
file soubor2 : tf_text;

-- pouziti
-- zapis hodnoty 123456 typu integer do binarniho souboru (otevren deklaraci)
write (soubor1, 123456);
file_close (soubor1);

-- zapis retezce do textoveho souboru (soubor nebyl otevren pri deklaraci)
file_open (soubor2, "soubor2.txt", write_mode);
write (soubor2, "Zapise tento text do souboru");
file_close (soubor2);
```

Typ **file** není syntetizovatelný. Je tedy použitelný pouze pro simulaci na simulátoru jazyka VHDL.

3.9 Paralelní příkazy

Všechny paralelní příkazy jazyka VHDL se vykonávají souběžně (tj. provádějí se najednou v čase). Na pořadí uvedení paralelních příkazů ve zdrojovém textu tedy nezáleží. Mezi paralelní příkazy patří:

- nepodmíněné přiřazení
- podmíněné přiřazení
- výběrové přiřazení
- příkaz process
- generující příkaz
- použití komponenty
- volání procedury
- příkaz bloku
- příkaz assert

3.9.1 Nepodmíněné přiřazení

Nepodmíněné přiřazení je přiřazení přímé hodnoty nebo výrazu do signálu. Přiřazení se provádí pomocí symbolu `<=` přičemž vlevo od symbolu je cílový identifikátor a vpravo je přiřazovaná hodnota s volitelnými prefixovými a postfixovými příkazy, určujícími charakter a velikost zpoždění.

Syntaxe nepodmíněného přiřazení:

```
[NAVESTI :] ID <= [transport | [reject cas] inertial] vyraz_elem {, vyraz_elem};
```

Výrazový element `vyraz_elem` má následující syntaxi:

```
vyraz [after cas] | null [after cas]
```

Není-li uvedena žádná volitelná prefixová část nebo je uvedena volitelná prefixová část **inertial**, má přiřazení vždy charakter inerčního (setrvačného) zpoždění. Setrvačné zpoždění je pro číslicové obvody charakteristické. Pokud má přiřazení charakter setrvačného zpoždění a má dojít k přenosu impulzu stejného nebo kratšího než časový limit stanovený pomocí **reject**, bude impulz potlačen a nedojde k jeho přenosu. Pokud není časový limit **reject** uveden, budou potlačeny všechny impulzy kratší než čas zpoždění prvního výrazového elementu (čas uvedený za **after** prvního výrazu v přiřazení). Pokud není uveden postfixový příkaz **after**, je implicitně uvažováno nulové zpoždění (tj. jako kdyby bylo uvedeno **after 0 ns**).

Použití volitelné části **transport** způsobí, že přiřazení bude mít charakter chování transportního zpoždění. Transportní zpoždění je charakteristické nekonečnou frekvenční odezvou, což ve výsledku znamená, že přiřazení přenese libovolně krátký impulz.

Pokud je použito přiřazení bez volitelných částí (v holém tvaru bez použití **transport**, **inertial** a **after**), je přiřazení syntetizovatelné (tj. lze ho realizovat v hardware, např. v programovatelném obvodu). Pokud je v přiřazení použita některá volitelná část (kromě návěstí), půjde pouze simulovat v simulátoru jazyka VHDL. Přiřazení hodnoty **null** neprovede žádnou akci. Použití různých variant nepodmíněného přiřazení ukazuje následující příklad.

Příklad:

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

entity nepodminene is
    port (
        a, b : out std_logic;
        sirka : out integer;
        sig1, sig2, sig3, sig4, sig5 : out std_logic
    );
end;

architecture a_nepodminene of nepodminene is
    -- deklarace vnitřních signálů
    signal c, d, sig : std_logic;
begin
    -- použití nepodmíněných přiřazení (syntetizovatelná)
    a <= not d;
    b <= c xor d;
    c <= '0';
    d <= '1';
```

```

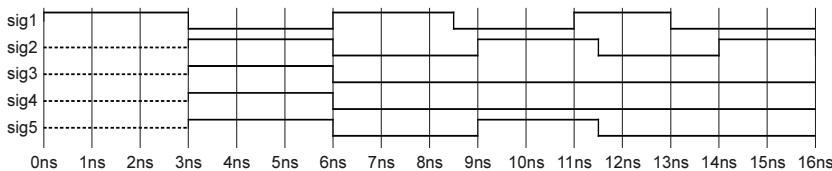
sirka <= 10;
sig1 <= sig;

-- pouziti nepodminenych prirazeni (nesyntetizovateln)
sig <= '1', '0' after 3 ns, '1' after 6 ns, '0' after 8.5 ns, '1' after 11 ns,
  '0' after 13 ns;
sig2 <= transport sig after 3 ns;
sig3 <= sig after 3 ns;
sig4 <= inertial sig after 3 ns;
sig5 <= reject 2 ns inertial sig after 3 ns;
end;

```

Výpis 3.2 Použití různých variant nepodmíněného přiřazení

Chování transportního a setrvačného zpoždění ukazuje obr. 3.2.



Obr. 3.2 Nepodmíněné přiřazení s transportním a setrvačným zpožděním

Zobrazené průběhy signálů *sig1* až *sig5* jsou výsledkem simulace výše uvedeného příkladu. Na signálu *sig1* jsou zřetězenými příkazy **after** postupně generovány impulzy dlouhé 3 ns, 2,5 ns a 2 ns (doba, kdy je signál v logické 1). Signál *sig2* má charakter transportního zpoždění, přenese tedy libovolně krátký impuls. Signál *sig3* je tedy přímou kopí signálu *sig1*, zpožděnou o 3 ns. Signál *sig4* nemá uveden typ zpoždění a má tedy implicitně charakter setrvačného zpoždění. Protože tento signál nemá uveden typ zpoždění a tedy ani časový limit **reject**, budou jím potlačeny všechny impulzy kratší než je čas zpoždění prvního výrazového elementu, tj. impulzy kratší než 3 ns. Signál *sig3* je tedy kopií signálu *sig1*, zpožděnou opět o 3 ns, neobsahuje však druhý a třetí impulz, protože tyto impulzy byly kratší než 3 ns. Signál *sig4* má uveden charakter setrvačného zpoždění. Jelikož není uveden časový limit **reject**, budou přiřazením potlačeny všechny impulzy kratší než čas zpoždění prvního výrazového elementu (stejně jako u signálu *sig3*). Oba signály *sig3* a *sig4* mají tedy stejný charakter zpoždění a dokonce stejný časový limit **reject**, jsou tedy rovnocenné. Signál *sig5* má uveden charakter setrvačného zpoždění. Vzhledem k tomu, že má uveden časový limit **reject**, budou jím potlačeny všechny impulzy stejné nebo kratší než tento časový limit. Signál *sig5* je tedy kopií signálu *sig1* zpožděnou opět o 3 ns, neobsahuje však třetí 2 ns impulz, protože tento impulz byl stejně dlouhý jako časový limit nastavený pomocí **reject**.

3.9.2 Podmíněné přiřazení

Podmíněné přiřazení je přiřazení přímé hodnoty nebo výrazu do signálu při splnění booleovské podmínky (má-li podmínka hodnotu **true**). Podmíněné přiřazení se provádí pomocí symbolu **<=** a konstrukce příkazu **when** volitelně s **else**. Podmíněná přiřazení lze zřetězit.

Syntaxe podmíněného přiřazení:

```
[NAVESTI :] ID <= [transport | [reject cas] inertial] {vyraz_elem when podminka  
else} vyraz_elem [when podminka];
```

Výrazový element *vyraz_elem* má následující syntaxi:

```
vyraz [after cas] | null [after cas]
```

Stejně jako u nepodmíněného přiřazení lze i u podmíněného přiřazení ve výrazu použít prefixové a postfixové příkazy, určující charakter a velikost zpoždění. Za předpokladu, že není použito příkazů **transport**, **inertial**, **reject** a **after**, je podmíněné přiřazení syntetizovatelné. Podmíněné přiřazení lze výhodně použít i pro jednoduché povolování signálu nebo skupiny signálů, realizaci třístavových výstupů, multiplexoru, prioritního kodéru, hladinového klopného obvodu (latch) a dalších jednoduchých obvodů. Zřetězený příkaz **when-else** vede při syntéze většinou na kaskádní zapojení, není proto příliš vhodný pro tvorbu vícevstupových multiplexorů, neboť takové multiplexory pak nemají stejné zpoždění ze vstupů na výstup. Pro tvorbu více vstupových multiplexorů je vhodnější použít paralelní příkaz „výběrové přiřazení“ (**with-select**) nebo příkaz **process** se sekvenčním příkazem **case** (viz další text). Další praktické příklady použití podmíněného přiřazení lze najít v kapitole 8 (základní funkční bloky).

Příklady:

```
-- deklarace  
signal d1, d2, d3, d4, en, oe, p, y1, y2, y3, y4 : std_logic;  
signal sel, y5 : std_logic_vector (1 downto 0);  
  
-- použití podmíněných přiřazení  
y0 <= d1 when en = '1'; -- hladinový klopný obvod (latch)  
y1 <= d1 when en = '1' else '0'; -- povolování signálu  
y2 <= d2 when oe = '1' else 'Z'; -- ovládání třístavového signálu  
y3 <= d3 when p = '1' else d4; -- multiplexer 2 na 1  
y4 <= d1 when sel = "00" else  
      d2 when sel = "01" else  
      d3 when sel = "10" else  
      d4;  
y5 <= "11" when d1 = '1' else -- prioritní kódér  
      "10" when d2 = '1' else  
      "01" when d3 = '1' else  
      "00";
```

3.9.3 Výběrové přiřazení

Výběrové přiřazení umožňuje přiřazení přímé hodnoty nebo výrazu do signálu na základě hodnoty jiného signálu (jedná se vlastně o obdobu sekvenčního příkazu **case**, viz další text).

Syntaxe výběrového přiřazení:

```
[NAVESTI :] with ID select  
      SIGID <= [transport | [reject cas] inertial] vyraz_elem when vyber [{,  
vyraz_elem when vyber}];
```

Výrazový element *vyraz_elem* má následující syntaxi:

```
vyraz [after cas] | null [after cas]
```

Element *vyber* má následující syntaxi:

vyraz | rozsah_hodnot | **others**

Stejně jako u nepodmíněného přiřazení, lze i u výběrového přiřazení použít prefixové a postfixové příkazy určující charakter a velikost zpoždění. Opět platí, že pokud není použito příkazů **transport**, **inertial**, **reject** a **after**, je výběrové přiřazení syntetizovatelné do hardware. Příkaz **others** zastupuje všechny ostatní hodnoty, jež nebyly předchozích příkazech **when** vyjmenovány.

Příklad použití:

```
-- deklarace
signal d1, d2, d3, d4, y1 : std_logic;
signal sel : std_logic_vector (1 downto 0);

-- multiplexer 4 na 1
with sel select
    y1 <= d1 when "00",
    d2 when "01",
    d3 when "10",
    d4 when others;
```

3.9.4 Proces

Příkazy uvedené uvnitř procesu se vykonávají na rozdíl od příkazů uváděných v předchozích podkapitolách sekvenčně, podobně jako v běžných programovacích jazycích. Navenek se však proces chová jako paralelní příkaz. O příkazech, jež lze použít uvnitř procesu, pojednává podkapitola 3.10. Procesy lze velmi dobře použít pro realizaci klopných obvodů, stavových automatů, sériového zpracování dat a mnoha dalších sekvenčních obvodů.

Syntaxe procesu:

```
[NAVESTI :] process [(SIGID {, SIGID})]
    [{definice}]
    [{deklarace}]
begin
    [{volani_procedur}]
    [{sekvencni_ridici_prikazy}]
    [{sekvencni_prirazovaci_prikazy_pro_signaly_nebo_promenne}]
end process [NAVESTI];
```

Každý proces je spuštěn automaticky na začátku simulace. Další spuštění procesu je možné dvěma způsoby:

- Proces lze dále spouštět na základě signálů uvedených v **citlivostním seznamu**. Citlivostní seznam signálů je uveden v závorkách za klíčovým slovem **process**. Pokud dojde ke změně hodnoty jakéhokoliv signálu, který je uveden v citlivostním seznamu (jednotlivé signály se v citlivostním seznamu oddělují čárkou), dojde ke spuštění procesu. Proces, který má citlivostní seznam, nesmí v těle procesu obsahovat řídicí příkazy **wait**.

Příklad:

```
-- deklarace
signal d, en, q : std_logic;
```

```
-- proces se spusti pri zmene signalu d nebo en
process (d, en)
begin
    -- do q se zapise d v pripade ze en = 1 (hladinovy klopny obvod)
    if (en = '1') then
        q <= d;
    end if;
end process;
```

- Druhá možnost, jak spouštět proces, je na základě příkazů **wait**. Proces, jehož tělo obsahuje řídící příkazy **wait**, nesmí již mít uveden citlivostní seznam. Pomocí příkazů **wait** lze vykonávání procesu pozastavit na určitou dobu (příkazem **wait for**), nebo zastavit do splnění určité podmínky (**wait until**), a nebo až do změny na nějakém signálu (příkaz **wait on**), případně lze tyto možnosti kombinovat. Jakmile proces bez citlivostního seznamu skončí, okamžitě se znova spouští. Pokud nechceme, aby se proces znova spustil, musíme použít příkaz **wait** (bez parametru). Všechny varianty příkazu **wait** jsou podrobněji popsány v podkapitole 3.10.3.

3.9.5 Generující příkaz

Příkaz **generate** lze s výhodou použít např. pro úsporný popis rovnic, daných nějakým rekurentním zápisem. Také lze příkaz **generate** použít pro popis struktur, které se opakují a jsou mezi sebou propojeny tak, že je lze popsat pomocí cyklů (např. hradla s mnoha vstupy, sčítáčky, násobičky, atd.).

Syntaxe generate:

```
NAVESTI : gener_schema generate
    [{blok_deklaraci}
begin
    {paralelni_vyrazy}
end generate [NAVESTI];
```

Element gener_schema má následující syntaxi:

```
gener_schema:
    for ID in od to | downto do
        if podminka
```

Pokud je použita forma příkazu **for generate**, pak je tělo příkazu **generate** prováděno opakováně. Řídící proměnná, označená identifikátorem ID, není nikde deklarována a je přístupná pouze v těle cyklu **for**. Hodnota proměnné je při každé iteraci cyklu automaticky zvětšena / zmenšena o 1. Tento příkaz lze s výhodou použít např. k indexaci prvků polí, apod.

Následující výpis kódu ukazuje realizaci parametrizovatelného hradla AND.

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

entity siroky_and is
    generic (
        sirka : positive := 64  -- sirka slova
    );
```

```

port (
    data : in std_logic_vector (sirka - 1 downto 0); -- vstupni vektor dat
    y : out std_logic -- vystup hradla AND
);
end siroky_and;

architecture struct of siroky_and is
    signal tmp : std_logic_vector (sirka - 1 downto 0);
begin
    tmp(0) <= data(0);
    wand: for i in 1 to sirka - 1 generate
        tmp(i) <= tmp(i - 1) and data(i);
    end generate wand;
    y <= tmp(sirka - 1);
end struct;

```

Výpis 3.3 Úplný zdrojový text realizující v jazyce VHDL parametrizovatelný AND

Bez použití **generate** by musel VHDL kód při šířce slova 64 bitů obsahovat 63 rovnic (v každé rovnici jeden **and**), a nebo jednu velmi dlouhou rovnici s 63 operátory **and**. Díky použití generické konstanty a příkazu **generate** je kód univerzální pro libovolně široký vstupní vektor dat (musí být alespoň dvoubitový).

Druhá forma příkazu **if generate** lze využít pro podmíněné generování příkazu. U příkazu **if generate** nelze použít **else**. Toto omezení však lze velmi jednoduše vyřešit např. použitím dvou příkazů **if generate**, přičemž druhý má uvedenu podmírkou negovanou.

```

and_data : if prepinac = '1' generate
    q <= a and b and c and d;
end generate;

xor_data : if not (prepinac = '1') generate
    q <= a xor b xor c xor d;
end generate;

```

3.9.6 Použití komponenty

V jazyce VHDL lze jednotlivé entity použít jako komponenty v dalších entitách a tak vytvářet hierarchické struktury. Návrh systému lze v jazyce VHDL vytvářet buď odshora dolů, nebo zdola nahoru, případně kombinovat oba způsoby (viz kapitola 15). Před použitím entity jako komponenty je třeba komponentu deklarovat. Deklarace komponenty se provádí pomocí příkazu **component**.

Syntaxe deklarace komponenty:

```

component ENTITYID is
    [generic ({ID : TYPEID [:= výraz];});]
    [port ({ID : [in | out | inout | buffer | linkage] TYPEID [:= výraz];});]
end component [ENTITYID];

```

Komponentu lze použít pomocí příkazu **port map**. Pomocí tohoto příkazu lze připojit na vstupy a výstupy použité komponenty signály. Volitelně lze pomocí příkazu **generic map** měnit hodnoty generických konstant.

Syntaxe použití komponenty:

```
NAVESTI : COMPID
[generic map ([GENCID =>] vyraz {, [GENCID =>] vyraz})]
port map ([PORTID =>] vyraz {, [PORTID =>] vyraz});
```

Pokud má generická konstanta nastavenu v deklaraci entity implicitní hodnotu (viz podkapitola 3.5), nemusí se její hodnota při použití entity jako komponenty v generickém mapu nastavovat. Pokud nám ale implicitní hodnota z nějakého důvodu nevyhovuje, můžeme ji samozřejmě v generickém mapu změnit najinou. Mají-li všechny generické konstanty dané entity implicitní hodnotu definovanou, může v takovém případě generický map úplně chybět. V případě, že hodnota generické konstanty nemá při deklaraci nastavenu implicitní hodnotu, musí se při každém použití takové entity jako komponenty (port mapu) hodnota této generické konstanty v generickém mapu definovat. Identifikátory nebo hodnoty lze na porty komponentu přiřazovat buď pomocí přiřazení \Rightarrow nebo bez něj. V případě, že uvádíme připojované identifikátory nebo hodnoty v pořadí, v jakém jsou uvedeny v deklaraci komponentu, není pak třeba přiřazovat jednotlivé parametry identifikátorům portů pomocí přiřazení \Rightarrow . Pokud však nechceme některé výstupní porty komponentu připojovat nebo chceme uvádět připojované porty v jiném pořadí, něž jak byly uvedeny v deklaraci komponentu (např. proto, že si pořadí nepamatujeme), musíme přiřazovat jednotlivé identifikátory nebo hodnoty na porty pomocí přiřazení \Rightarrow . Přitom vlevo od symbolu přiřazení je jméno portu, na které se přiřazení provádí a vpravo je přiřazovaný identifikátor nebo hodnota.

Příklad použití:

```
-- architektura obsahuje dve 32 vstupa hradla AND
architecture struct of tb_siroky_and is
  -- deklarace komponentu z predchozi podkapitoly (vypis 3.2.)
  component siroky_and is
    generic (
      sirka : positive := 64
    );
    port (
      data : in std_logic_vector (sirka - 1 downto 0);
      y : out std_logic
    );
  end component;

  -- deklarace signalu
  signal d1, d2 : std_logic_vector (31 downto 0);
  signal y1, y2 : std_logic;

begin
  -- pouziti komponentu
  and32a : siroky_and
    generic map (32)
    port map (d1, y1);

  -- dalsi pouziti komponentu
  and32b : siroky_and
    generic map (sirka => 32)
    port map (y => y2, data => d2);
end struct;
```

3.9.7 Volání procedury

Proceduru lze v jazyce VHDL volat podobně jako v jiných jazycích, tj. identifikátorem (jménem) procedury s tím, že případné parametry uvedeme v závorce. Samotná procedura je uvnitř, podobně jako proces, vykonávána sekvenčně. Procedurám se věnuje podkapitola 3.10.13.

Syntaxe volání procedury:

```
[NAVESTI :] PROCID([(PARID =>] vyraz [, [PARID =>] vyraz)]);
```

Pro pořadí parametrů při volání procedury platí stejný princip přiřazování jako pro pořadí portů při port mapu komponentů (viz předchozí podkapitola).

Příklad použití:

```
-- predpokladana deklarace procedure
procedure clock (num : in integer; signal clk : out std_logic)

-- volani procedury
clock (150, hodiny);
clock (clk => hodiny, num => 150);
```

3.9.8 Příkaz bloku

Definice bloku slouží k dekompozici návrhu a tím lepší čitelnosti zdrojového kódu. Uvnitř bloku lze definovat generické konstanty, mapovat generické konstanty, definovat porty a mapovat porty.

Syntaxe bloku:

```
BLOCKID : block [is]
  [generic ({ID : TYPEID [:= vyraz];});
   [generic map ([GENCID =>] vyraz [, [GENCID =>] vyraz)]);
   [port ({ID : [in | out | inout | buffer] TYPEID [:= vyraz];});
    [port map ([PORTID =>] vyraz [, [PORTID =>] vyraz]);]
   [{deklarace}]
 begin
  [{paralelni_prikazy}]
 end block [BLOCKID];
```

3.9.9 Příkaz assert

Příkaz **assert** vyhodnotí podmínsku a pokud má podmínka hodnotu **false**, provede příkaz **report** (viz podkapitola 3.10.12), který zajistí výpis výrazu (řetězce) na textový výstup (většinou okna simulátoru). Příkaz lze použít jako paralelní příkaz i jako sekvenční příkaz (častější použití). Podrobný popis syntaxe a chování příkazu **assert** je uvedeno v podkapitole 3.10.11.

3.10 Sekvenční příkazy

Těla procesů a procedur jsou v jazyce VHDL vykonávána sekvenčně. Sekvenční příkazy jazyka VHDL lze používat v procesech a procedurách a lze je dobře použít pro popis sekvenčních (algoritmických) částí návrhu.

3.10.1 Sekvenční přiřazení do proměnné

Přiřazení do proměnné se provádí pomocí symbolu `:=`, přičemž vlevo od symbolu je cílový identifikátor a vpravo je přiřazovaná hodnota.

Syntaxe přiřazení do proměnné:

```
VARID := vyraz;
```

Přiřazení do proměnné lze použít pouze v procesu nebo proceduře. Přiřazení hodnoty do proměnné se uskuteční okamžitě (na rozdíl od sekvenčního přiřazení do signálu).

Příklad použití:

```
process
    variable a, b, c : integer := 0;
begin
    a := 1;      -- po provedení příkazu je a = 1 ; b = 0 ; c = 0
    b := 2;      -- po provedení příkazu je a = 1 ; b = 2 ; c = 0
    c := a + b; -- po provedení příkazu je a = 1 ; b = 2 ; c = 3
    a := 4;      -- po provedení příkazu je a = 4 ; b = 2 ; c = 3
    wait;
end process; -- po ukončení procesu je a = 4 ; b = 2 ; c = 3
```

3.10.2 Sekvenční přiřazení do signálu

Sekvenční přiřazení do signálu je přiřazení přímé hodnoty nebo výrazu do signálu. Syntaxe a použití sekvenčního přiřazení je stejně jako paralelního nepodmíněného přiřazení. Přiřazení se tedy provádí pomocí symbolu `<=` přičemž vlevo od symbolu je cílový identifikátor a vpravo je přiřazovaná hodnota s volitelnými prefixovými a postfixovými příkazy, určujícími charakter a velikost zpoždění. Podrobněji o syntaxi viz podkapitola 3.9.1. Sekvenční přiřazení do signálu však, na rozdíl od přiřazení do proměnné, neprovede zápis nové hodnoty okamžitě po provedení příkazu, ale až po ukončení procesu a nebo ve chvíli, kdy se při vykonávání procesu provede příkaz `wait`. Pokud bylo zapsáno do signálu v průběhu provádění procesu několikrát, bude do signálu zapsána posledně přiřazená hodnota.

Syntaxe přiřazení do signálu:

```
[NAVESTI :] SIGID <= [transport | [reject cas] inertial] {vyraz_elem};
```

Příklad použití:

```
-- deklarace
signal a, b, c : integer := 0;

-- přiřazení do signálu uvnitř procesu
process
begin
    a <= 1;      -- po provedení příkazu je a = 0 ; b = 0 ; c = 0
    b <= 2;      -- po provedení příkazu je a = 0 ; b = 0 ; c = 0
    c <= a + b; -- po provedení příkazu je a = 0 ; b = 0 ; c = 0
    a <= 4;      -- po provedení příkazu je a = 0 ; b = 0 ; c = 0
    wait;
end process; -- po ukončení procesu je a = 4 ; b = 2 ; c = 0
```

3.10.3 Příkaz wait

Příkaz **wait** slouží k pozastavení a nebo zastavení běhu procesu. Příkaz má několik forem, které lze navíc vzájemně kombinovat.

Syntaxe příkazu **wait**:

```
wait [on SIGID {, SIGID}] [until výraz] [for doba];
```

- Pokud je použita citlivostní forma **wait on seznam signálů**, pozastaví příkaz běh procesu, dokud nenastane změna na jednom z vyjmenovaných signálů.
- Pokud je použita podmínková forma **wait until výraz**, pozastaví příkaz běh procesu, dokud není booleovská podmínka rovna hodnotě **true**. Podmínka je testována při každé změně hodnoty signálů uvedených v podmínce nebo v seznamu signálů citlivostní formy.
- Pokud je použita časová forma **wait for doba**, pozastaví příkaz běh procesu na určenou dobu. Pokud je uvedena současně citlivostní nebo podmínková forma, mají tyto ve vyhodnocování přednost. Časová forma příkazu (**wait for**) není syntetizovatelná a je tedy použitelná pouze pro simulaci na simulátoru jazyka VHDL.

Příklad:

```
-- deklarace
signal clk, q : std_logic;

-- proces se bude znova spustit každých 10 ns
-- výsledkem tohoto procesu je hodinový signál s periodou 20 ns
process
    variable clk_tmp : std_logic := '0';
begin
    if q = 'H' then -- pokud je q = 'H' zastavi proces
        wait;
    end if;
    clk_tmp := not clk_tmp;
    clk <= clk_tmp;
    wait for 10 ns; -- pozastavi proces na dobu 10 ns
end process;

-- proces se vykona pouze jednou (obsahuje wait bez parametru)
process
begin
    q <= '1';
    wait for 21 ns; -- pozastavi proces na dobu 21 ns
    wait on clk; -- pozastavi proces do doby než se změní signál clk
    q <= '0';
    wait on clk; -- pozastavi proces do doby než se změní signál clk
    q <= '1';
    wait until clk = '1'; -- pozastavi proces do doby než se
                           -- změní signál clk na hodnotu '1'
    q <= '0';
    wait on clk until q = '1' for 30 ns; -- pozastavi proces do doby než
                                         -- se změní signál clk a q má hodnotu '1'
                                         -- pokud není tato podmínka splněna po dobu 30 ns
                                         -- proces po uplynutí doby pokracuje dále
    q <= 'H'; -- zajistí zastavení prvního procesu
    wait; -- zastavi proces
end process;
```

3.10.4 Příkaz if

Příkaz **if** slouží pro vykonání sekvence příkazů v závislosti na vyhodnocení podmínky, případně dalších podmínek.

Syntaxe příkazu **if**:

```
[NAVESTI :]
    if podminka1 then
        {sekvence_prikazu}
    [{elsif podminka2 then
        {sekvence_prikazu2}}]
    [else
        {sekvence_prikazu}]
end if [NAVESTI];
```

Funkce příkazu **if** je podobná jako u jiných jazyků. Je-li splněna podmínka *podminka1*, provedou se příkazy obsažené v *sekvenci_prikazu1*. Je-li přítomna další podmínka *podminka2* a je-li splněna, provedou se příkazy obsažené v *sekvenci_prikazu2*, a tak dále. Je-li přítomna poslední část s **else** a nebyla-li splněna žádná předcházející podmínka, provedou se příkazy obsažené v této *sekvenci_prikazu*. Podmínky jsou tedy testovány v pořadí od první k poslední, z toho vyplývá jejich prioritní výběr mezi ostatními.

Příkaz **if** je vhodný např. pro tvorbu prioritních kodérů, klopných obvodů, registrů, atd.

Příklad použití:

```
-- deklarace
signal a, b, c : std_logic;
signal y : std_logic_vector (1 downto 0);

-- prioritni koder
process (a, b, c)
begin
    if a = '1' then      -- nejvetsi priorita
        y <= "00";
    elsif b = '1' then
        y <= "10";
    elsif c = '1' then
        y <= "01";
    else                  -- nejmensi priorita
        y <= "11";
    end if;
end process;
```

3.10.5 Příkaz case

Příkaz **case** slouží podobně jako předchozí příkaz **if** pro vykonání sekvence příkazů v závislosti na vyhodnocení podmínek, avšak bez priorit.

Syntaxe příkazu **case**:

```
[NAVESTI :] case vyraz is
    when hodnota1 | hodnota2 =>
        {sekvence_prikazu}
    when hodnota3 | hodnota4 => {sekvence_prikazu}
end case [NAVESTI];
```

Ze syntaxe příkazu je zřejmé, že pokud se některá sekvence příkazů má vykonávat pro několik hodnot, stačí hodnoty od sebe oddělit v příkazu **when** symbolem |. Pomocí příkazu **case** lze snadno realizovat dekodéry, multiplexery, stavové automaty, atd.

Příklad použití:

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

entity mux_case is
    port (
        a, b, c, d : in std_logic;
        sel : in std_logic_vector (1 downto 0);
        y : out std_logic
    );
end mux_case;

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

Výpis 3.4 Úplný zdrojový text realizující v jazyce VHDL multiplexer 4 na 1

Uvedený výpis kódu realizuje multiplexer 4 na 1 pomocí příkazu **process** a **case**. Možnost **when others** je vhodné v příkazu **case** testovat vždy. V případě, že opomeneme otestovat příkazy **when** všechny hodnoty, jichž může testovaný signál nabývat, dojde pak ke splnění podmínky **when others** a vykoná se příslušná sekvence příkazů. Ta může přiřadit nějakou implicitní hodnotu nebo realizovat nějaký chybový výpis pomocí příkazu **report** nebo **assert** (viz podkapitola 3.10.11. a 3.10.12.), apod.

3.10.6 Příkaz loop

Příkaz **loop** slouží pro opakování vykonání sekvence příkazů v závislosti na vyhodnocení podmínky nebo příkazu cyklu **for**.

Syntaxe příkazu **loop**:

```
[NAVESTI :] [iteracni_schema] loop
    {sekvence_prikazu}
end loop [NAVESTI];
```

Element *iteracni_schema* má následující syntaxi:

```
while podminka
for ID in od to | downto do
```

V případě použití iteračního schématu **while** je na začátku každého cyklu testována uvedená podmínka. Pokud má podmínka hodnotu **true**, je vykonáno tělo smyčky a pokud má hodnotu **false**, je vykonávání příkazu **loop** ukončeno. V případě použití iteračního schématu **for** je uvedený identifikátor automaticky deklarován jako konstanta diskrétního rozsahu (není ho tedy třeba deklarovat). Jedná se o lokální deklaraci, která je viditelná pouze uvnitř cyklu. Protože se jedná o konstantu, lze ji použít pouze na pravé straně přiřazení, připojovat na vstupy, atd. Vykonávání smyčky končí dosažením pravé hodnoty uvedeného rozsahu. Ukončení vykonávání právě běžící iterace a skok na další iteraci příkazu **loop** lze realizovat pomocí příkazu **next**. Vykonávání příkazu **loop** lze ukončit vykonáním příkazu **exit** nebo **return** (viz další podkapitoly).

Příklad použití příkazu loop:

```
architecture struct of siroky_and is
begin
    process (data)
        variable tmp : std_logic;
    begin
        tmp := data(0);
        for i in 1 to sirka - 1 loop
            tmp := tmp and data(i);
        end loop;
        y <= tmp;
    end process;
end;
```

Další, tentokrát celý výpisu kódu počítá pomocí příkazu **loop** počet bitů ve vstupních datech, které mají hodnotu log. 1.

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

entity poc_jedn is
    port (
        data : in std_logic_vector (7 downto 0);
        pocet : out integer range 0 to 8
    );
end poc_jedn;

architecture struct of poc_jedn is
begin
    process (data)
        variable poc_bitu : integer range 0 to 8;
    begin
        poc_bitu := 0;
        for i in data'range loop
            if (data(i) = '1') then
                poc_bitu := poc_bitu + 1;
            end if;
        end loop;
        pocet <= poc_bitu;
    end process;
end struct;
```

Výpis 3.5 Počítadlo jedničkových bitů

3.10.7 Příkaz next

Příkaz **next** lze použít pro ukončení vykonávání právě běžící iterace příkazu **loop** a skok na další iteraci. Příkaz **next** se implicitně vztahuje na nejbližší příkaz **loop**. Pokud se má příkaz **next** vztahovat k jinému než nejbližšímu příkazu **loop**, je třeba uvést návštětí příslušného **loop** příkazu. V případě, že je uvedena podmínka **when**, lze vykonávání příkazu **next** podmínit (příkaz **next** je proveden, pokud má podmínka hodnotu **true**).

Syntaxe příkazu **next**:

```
[NAVESTI :] next [LOOPID] [when podminka];
```

3.10.8 Příkaz exit

Příkaz **exit** slouží pro ukončení vykonávání příkazu **loop**. Příkaz **exit** se implicitně vztahuje na nejbližší příkaz **loop**. Pokud se má příkaz **exit** vztahovat k jinému než nejbližšímu příkazu **loop**, je třeba uvést návštětí příslušného **loop** příkazu. V případě, že je uvedena podmínka **when**, lze vykonání příkazu **exit** podmínit (příkaz je proveden, pokud má podmínka hodnotu **true**).

Syntaxe příkazu **exit**:

```
[NAVESTI :] exit [LOOPID] [when podminka];
```

3.10.9 Příkaz return

Příkaz **return** lze použít pouze uvnitř procedury nebo funkce a slouží pro ukončení vykonávání této procedury nebo funkce. V případě použití příkazu uvnitř procedury není dovoleno uvést v příkazu výraz (návratovou hodnotu). Naopak v případě použití příkazu uvnitř funkce je uvedení výrazu (návratové hodnoty) nutné.

Syntaxe příkazu **return**:

```
[NAVESTI :] return [výraz];
```

3.10.10 Příkaz null

Příkaz **null** neprovádí žádnou akci. Použít jej lze například v příkazu **case** pro hodnoty, při kterých nechceme provádět žádná přiřazení apod. Viz například výpis 3.4.

Syntaxe příkazu **null**:

```
[NAVESTI :] null;
```

3.10.11 Příkaz assert

Příkaz **assert** vyhodnotí podmínu. Pokud má podmínka hodnotu **false**, provede příkaz **report**, který zajistí výpis výrazu (řetězce) na textový výstup (většinou okno simulátoru). Výraz v příkazu **report** musí být typu **string**. Dále lze nastavit úroveň závažnosti vypisovaného řetězce (chyby) pomocí příkazu **severity**. Možné úrovně závažnosti jsou dány předdefinovaným výčtovým typem **severity_level**, který může nabývat hodnot **note**, **warning**,

error a **failure**. V případě, že není úroveň závažnosti uvedena, je implicitně nastavena na hodnotu **error**. V konfiguraci simulátoru lze pak většinou nastavit, při které úrovni závažnosti chyby má simulátor zastavit simulaci a předat řízení uživateli. Většina simulátorů má tuto hodnotu nastavenu implicitně na hodnotu **failure**. Příkaz **assert** není syntetizovatelný a je většinou syntezátorů při syntéze ignorován, takže jeho výskyt nebrání syntéze.

Syntaxe příkazu assert:

```
[NAVESTI :] assert podminka  
    [report vyraz_typu_retezec]  
    [severity hodnota_typu_severity_level];
```

Příklad použití:

```
process  
begin  
    assert false;      -- ERROR: Assertion violation  
    wait for 10 ns;  
    assert false report "Text";                      -- ERROR: Text  
    wait for 10 ns;  
    assert false report "Poznamka" severity note;   -- NOTE: Poznamka  
    wait for 10 ns;  
    assert false report "Varovani" severity warning; -- WARNING: Varovani  
    wait for 10 ns;  
    assert false report "Chyba" severity error;       -- ERROR: Chyba  
    wait for 10 ns;  
    assert now < 50 ns report "Porucha" severity failure; -- FAILURE: Porucha  
end process;
```

V posledním příkazu **assert**, ve výše uvedeném příkladu, je použita funkce **now**, která vrací aktuální simulační čas (v době vykonání příkazu má hodnotu 50 ns). Podmínka má tedy hodnotu **false** a simulátorem je vypsán uvedený textový řetězec s úrovní závažnosti **failure** a zastavena simulace.

3.10.12 Příkaz report

Jak již bylo uvedeno v předchozí podkapitole, příkaz **report** zajistí výpis výrazu (řetězce) na textový výstup do informačního okna simulátoru. Výraz v příkazu **report** musí být typu **string** a lze nastavit úroveň závažnosti vypisovaného řetězce (chyby) pomocí příkazu **severity** (hodnoty **note**, **warning**, **error** a **failure**). V případě, že není úroveň závažnosti uvedena, je implicitně nastavena na hodnotu **note**. Stejně jako v případě příkazu **assert** zastaví simulátor simulaci ve chvíli, kdy je závažnost vypisované zprávy rovna nastavené hodnotě (implicitně hodnota **failure**). Příkaz **report** není syntetizovatelný a je většinou syntezátorů při syntéze ignorován, takže jeho výskyt nebrání syntéze.

Syntaxe příkazu report:

```
[NAVESTI :] report vyraz_typu_retezec  
    [severity hodnota_typu_severity_level];
```

Příklad použití:

```
process  
begin  
    report "Text";                                -- NOTE: Text
```

```

wait for 10 ns;
report "Poznamka" severity note;      -- NOTE: Poznamka
wait for 10 ns;
report "Varovani" severity warning; -- WARNING: Varovani
wait for 10 ns;
report "Chyba" severity error;        -- ERROR: Chyba
wait for 10 ns;
report "Porucha" severity failure;   -- FAILURE: Porucha
end process;

```

Poslední příkaz **report** ve výše uvedeném příkladu vypisuje uvedený textový řetězec s úrovní závažnosti **failure**, simulátor tedy po výpisu řetězce zastaví simulaci.

3.10.13 Procedury

Procedura představuje, podobně jako proces, sekvenční prostředí. Příkazy uvnitř procedury jsou tedy vykonávány sekvenčně. Procedury lze volat z procesů.

Syntaxe procedury:

```

procedure ID[({constant | variable | signal} ID : in | out | inout TYPEID
    [:= výraz];})] [is
begin
    {sekvenční_příkazy}
end [procedure] [ID]];

```

Příklad:

```

procedure clock (num : in integer; signal clk : out std_logic) is
begin
    for i in 1 to num loop
        clk <= '0';
        wait for 20 ns;
        clk <= '1';
        wait for 80 ns;
    end loop;
end clock;

```

Výše uvedená procedura má dva parametry. První parametr je vstupní typu **integer**, druhý parametr je výstupní signál typu **std_logic**. Tělo procedury vygeneruje do výstupního signálu obdélníkový průběh s periodou 100 ns, střídou 80:20 o počtu period udaných prvním vstupním parametrem.

3.10.14 Funkce

Funkce představuje podobně jako procedura sekvenční prostředí. Funkce lze volat z procesů a příkazy těla funkce jsou vykonávány sekvenčně.

Syntaxe funkce:

```

function ID [({constant | variable | signal | file} ID : [in] TYPEID [:=
    výraz];})] return TYPEID [is
begin
    {sekvenční_příkazy}
end [function] [ID]];

```

Příklad:

```
function hyst_prah (data : integer; souc_stav : boolean) return boolean is
begin
    if data < 10000 then
        return false;
    elsif data > 15000 then
        return true;
    end if;
    return souc_stav;
end;
```

Výše uvedená funkce má dva vstupní parametry. První vstupní parametr s identifikátorem *data* je typu **integer**, druhý vstupní parametr s identifikátorem *souc_stav* je typu **boolean**. Funkce vrací návratovou hodnotu typu **boolean** a realizuje prahování s hysterézí. Pokud je hodnota prvního parametru menší než hodnota 10 000, vrací funkce hodnotu **false**. Pokud je hodnota větší než 15 000, vrací hodnotu **true**, a pokud se nachází vstupní hodnota mezi těmito hodnotami, vrací funkce návratovou hodnotu rovnou druhému vstupnímu parametru *souc_stav* (tj. současnou hodnotu).

3.11 Atributy

Pojmem atributy se v jazyce VHDL rozumí speciální funkce, jež poskytuje informaci o objektech, na nichž je atribut použit. Atributy lze použít na identifikátory typů, podtypů, konstant, signálů, proměnných, entit, architektur a dalších objektů. Název atributu se uvádí za apostrofem. Apostrof s atributem se uvádí za identifikátorem (prefixem), k němuž se atribut vztahuje. Atributy mohou mít parametr. Parametr atributu se uvádí, podobně jako u funkcí a procedur, v závorkách. Jazyk VHDL má velké množství atributů předdefinovaných a uživatel si také může definovat vlastní atributy. Předdefinované atributy jazyka VHDL lze rozdělit do čtyř skupin – atributy typů, polí, signálů a pojmenovaných objektů. V dalším textu se budeme věnovat každé skupině předdefinovaných atributů zvlášť.

Syntaxe použití atributu:

```
PREFIX'jmeno_atributu[(vyraz)]
```

Příklady použití:

```
clk'event          pole'range         pole;left(2)
```

3.11.1 Atributy typů

Předdefinovaných atributů určených k získávaní informací o typech je celkem 14. Funkce jednotlivých atributů ukazuje příklad na konci této podkapitoly.

T'ascending

Popis: vrátí booleovskou hodnotu **true**, pokud je typ nebo podtyp deklarován se vzestupným rozsahem, jinak vrátí hodnotu **false**

Prefix T: jakýkoliv skalární typ nebo podtyp

Výsledek: hodnota typu **boolean**

T'base

Popis: vrátí základní typ daného typu nebo podtypu T
Prefix T: jakýkoliv typ nebo podtyp
Výsledek: typ
Omezení: atribut lze použít pouze jako prefix jiného atributu (např. $T'base'left$)

T'left

Popis: vrátí hodnotu daného typu nebo podtypu T , která je nejvíce vlevo
Prefix T: jakýkoliv skalární typ nebo podtyp
Výsledek: hodnota stejného typu jako byl typ prefixu T

T'right

Popis: vrátí hodnotu daného typu nebo podtypu T , která je nejvíce vpravo
Prefix T: jakýkoliv skalární typ nebo podtyp
Výsledek: hodnota typu jako typ prefixu T

T'high

Popis: vrátí nejvyšší hodnotu daného typu nebo podtypu T
Prefix T: jakýkoliv skalární typ nebo podtyp
Výsledek: hodnota typu jako typ prefixu T

T'low

Popis: vrátí nejnižší hodnotu daného typu nebo podtypu T
Prefix T: jakýkoliv skalární typ nebo podtyp
Výsledek: hodnota typu jako typ prefixu T

T'pos(x)

Popis: vrátí pozici hodnoty x v daném typu nebo podtypu T
Prefix T: jakýkoliv diskrétní nebo fyzický typ nebo podtyp
Parametr: výraz, jehož typ je základní typ T
Výsledek: hodnota typu jako typ **integer**

T'val(x)

Popis: vrátí hodnotu na pozici x daného typu T
Prefix T: jakýkoliv diskrétní nebo fyzický typ nebo podtyp
Parametr: výraz, jehož typ je jakýkoliv celočíselný typ
Výsledek: hodnota typu jako je základní typ prefixu T
Omezení: chyba nastane, pokud výsledek neodpovídá rozsahu $T'low$ **to** $T'high$

T'succ(x)

Popis: vrátí hodnotu na pozici o jednu větší než je pozice hodnoty x
Prefix T: jakýkoliv diskrétní nebo fyzický typ nebo podtyp
Parametr: výraz, jehož typ je základní typ T
Výsledek: hodnota typu jako je základní typ prefixu T
Omezení: chyba nastane pokud x je rovno $T'high$ nebo pokud je x mimo rozsah $T'low$ **to** $T'high$

T'pred(x)

Popis: vrátí hodnotu na pozici o jednu menší než je pozice hodnoty x
Prefix T: jakýkoliv diskrétní nebo fyzický typ nebo podtyp
Parametr: výraz, jehož typ je základní typ T
Výsledek: hodnota typu jako je základní typ prefixu T
Omezení: chyba nastane, pokud x je rovno $T'low$ nebo pokud je x mimo rozsah $T'low$ **to** $T'high$

T'leftof(x)

Popis: vráti hodnotu vlevo od hodnoty x daného typu T
Prefix T: jakýkoliv diskrétní nebo fyzický typ nebo podtyp
Parametr: výraz, jehož typ je základní typ T
Výsledek: hodnota typu jako je základní typ prefixu T
Omezení: chyba nastane, pokud x je rovno T'left nebo pokud je x mimo rozsah T'low to T'high

T'rightof(x)

Popis: vráti hodnotu vpravo od hodnoty x daného typu T
Prefix T: jakýkoliv diskrétní nebo fyzický typ nebo podtyp
Parametr: výraz, jehož typ je základní typ T
Výsledek: hodnota typu jako je základní typ prefixu T
Omezení: chyba nastane, pokud x je rovno T'right nebo pokud je x mimo rozsah T'low to T'high

T'image(x)

Popis: převede výraz x, který je typu T, na textový řetězec
Prefix T: jakýkoliv skalární typ nebo podtyp
Parametr: výraz, jehož typ je základní typ T
Výsledek: typu string

T'value(x)

Popis: převede textový řetězec na hodnotu typu T
Prefix T: jakýkoliv skalární typ nebo podtyp
Parametr: typu string
Výsledek: hodnota typu jako je základní typ prefixu T

Příklad použití:

```
-- definice
type mlv7 is ('0','1','X','Z','H','L','W');
subtype mlv4 is mlv7 range '0' to 'Z';

-- deklarace
signal s1 : mlv7 := mlv4'left;           -- s1 ma hodnotu '0'
signal s2 : mlv7 := mlv4'base'left;       -- s2 ma hodnotu '0'
signal s3 : mlv7 := mlv4'right;          -- s3 ma hodnotu 'Z'
signal s4 : mlv7 := mlv4'base'right;      -- s4 ma hodnotu 'W'
signal s5 : mlv7 := mlv4'high;           -- s5 ma hodnotu 'Z'
signal s6 : mlv7 := mlv4'low;            -- s6 ma hodnotu '0'
signal i1 : integer := mlv7'pos('W');    -- I1 ma hodnotu 6
signal i2 : integer := mlv4'pos('Z');    -- I2 ma hodnotu 3
signal s7 : mlv7 := mlv4'val(2);        -- s7 ma hodnotu 'X'
signal s8 : mlv7 := mlv4'succ('1');      -- s8 ma hodnotu 'X'
signal s9 : mlv7 := mlv4'pred('1');      -- s9 ma hodnotu '0'
signal s10 : mlv7 := mlv4'leftof('1');   -- s10 ma hodnotu '0'
signal s11 : mlv7 := mlv4'rightof('1');  -- s11 ma hodnotu 'X'
signal s12 : boolean := mlv4'ascending;  -- s12 ma hodnotu true
signal s13 : string(1 to 3) := mlv4'image('Z'); -- s13 ma hodnotu "Z"
signal s14 : mlv7 := mlv7'value("L");    -- s14 ma hodnotu 'L'
```

3.11.2 Atributy polí

Předdefinovaných atributů určených k získávaní informací o polích je celkem 8. Funkce jednotlivých atributů ukazuje opět příklad na konci podkapitoly. U všech atributů polí platí, že jako prefix *A* může být použito jakékoli omezené pole nebo jeho alias.

A'ascending[(n)]

Popis: vrátí booleovskou hodnotu **true**, pokud má *n*-tá dimenze pole vzestupný (**to** rozsah, jinak vrátí hodnotu **false**)
Parametr: statický výraz typu **integer**, jehož hodnota musí být \leq dimenzi pole *A*
Výsledek: hodnota typu **boolean**
Poznámka: pokud je *n* vynecháno, je implicitně *n* = 1

A'left[(n)]

Popis: vrátí index prvku *n*-té dimenze pole, který je nejvíce vlevo (levou mez *n*-té dimenze pole)
Parametr: statický výraz typu **integer**, jehož hodnota musí být \leq dimenzi pole *A*
Výsledek: hodnota typu **integer**
Poznámka: pokud je *n* vynecháno, je implicitně *n* = 1

A'right[(n)]

Popis: vrátí index prvku *n*-té dimenze pole, který je nejvíce vpravo (pravou mez *n*-té dimenze)
Parametr: statický výraz typu **integer**, jehož hodnota musí být \leq dimenzi pole *A*
Výsledek: hodnota typu **integer**
Poznámka: pokud je *n* vynecháno, je implicitně *n* = 1

A'high[(n)]

Popis: vrátí index prvku *n*-té dimenze pole, který je horní mezi *n*-té dimenze pole
Parametr: statický výraz typu **integer**, jehož hodnota musí být \leq dimenzi pole *A*
Výsledek: hodnota typu **integer**
Poznámka: pokud je *n* vynecháno, je implicitně *n* = 1

A'low[(n)]

Popis: vrátí index prvku *n*-té dimenze pole, který je dolní mezi *n*-té dimenze pole
Parametr: statický výraz typu **integer**, jehož hodnota musí být \leq dimenzi pole *A*
Výsledek: hodnota typu **integer**
Poznámka: pokud je *n* vynecháno, je implicitně *n* = 1

A'range[(n)]

Popis: vrátí rozsah indexů *n*-té dimenze pole (A'left **to** A'right nebo A'left **downto** A'right)
Parametr: statický výraz typu **integer**, jehož hodnota musí být \leq dimenzi pole *A*
Výsledek: rozsah
Poznámka: pokud je *n* vynecháno, je implicitně *n* = 1

A'reverse_range[(n)]

Popis: vrátí obrácený rozsah indexů *n*-té dimenze pole (zamění **to** za **downto** a **downto** za **to**)
Parametr: statický výraz typu **integer**, jehož hodnota musí být \leq dimenzi pole *A*
Výsledek: rozsah
Poznámka: pokud je *n* vynecháno, je implicitně *n* = 1

A'length|(n)|

Popis: vráti počet prvků n -té dimenze pole

Parametr: statický výraz typu **integer**, jehož hodnota musí být \leq dimenzi pole A

Výsledek: hodnota rovna výrazu $T'pos(A'high(n)) - T'pos(A'low(n)) + 1$

Poznámka: pokud je n vynecháno, je implicitně $n = 1$

Příklad použití:

```
-- definice
type bitove_pole is array (0 to 15, 7 downto 0) of std_logic;
-- deklarace
signal pole : bitove_pole;
signal s1 : integer := pole'left;          -- s1 = 0
signal s2 : integer := pole'right;         -- s2 = 15
signal s3 : integer := pole'left(2);        -- s3 = 7
signal s4 : integer := pole'right(2);       -- s4 = 0
signal s5 : integer := pole'high;           -- s5 = 15
signal s6 : integer := pole'low;            -- s6 = 0
signal s7 : integer := pole'high(2);        -- s7 = 7
signal s8 : integer := pole'low(2);          -- s8 = 0
signal s9 : integer := pole'length;         -- s9 = 16
signal s10 : integer := pole'length(2);      -- s10 = 8
signal s11 : boolean := pole'ascending;      -- s11 = true
signal s12 : boolean := pole'ascending(2);    -- s12 = false
signal s13 : integer range pole'range;       -- s13 = 0, integer rozsah 0 to 15
signal s14 : integer range pole'range(2);     -- s14 = 7, integer rozsah 7 downto 0
```

3.11.3 Atributy signálů

Předdefinovaných atributů, určených k získávaní informací o signálech, je celkem 11. U všech atributů pro signály platí, že jako prefix S může být použit jakýkoliv signál.

S'active

Popis: vráti booleovskou hodnotu **true**, pokud byl signál S v daném simulačním cyklu aktivní (nastala-li na signálu transakce).

Výsledek: hodnota typu **boolean**

S'last_active

Popis: vráti čas od poslední aktivity (transakce) na signálu s . Pokud transakce na daném signálu nikdy nenastala, pak vráti hodnotu 0 ns.

Výsledek: hodnota typu **time**

S'delayed|(t)|

Popis: vytvoří kopii signálu S zpožděnou o čas t .

Parametr: nezáporná hodnota nebo výraz typu **time**

Výsledek: nový signál stejného typu, jako je základní typ prefixu S

Poznámka: pokud není parametr t udán, je signál zpožděn o tzv. delta zpoždění.

Příklad použití:

```
-- deklarace
signal clk, data : std_logic;

-- Test dodržení hold time vstupních dat
process (clk'delayed(10 ns))
```

```

begin
  if (clk = '1' and clk'stable(10 ns)) then
    assert(data'stable(10 ns))
      report "Data hold time nedodrzen!"
      severity warning;
  end if;
end process;

```

S'stable[(t)]

Popis: vrací booleovskou hodnotu **true**, pokud signál *S* nezměnil hodnotu (nenastala na něm událost) po dobu *t*, jinak vrací hodnotu **false**.

Parametr: výraz, jehož výsledek je nezáporný a je typu **time**

Výsledek: nový signál typu **boolean**

Poznámka: je-li je *t* vynecháno: vrátí **true**, pokud signál v daném simulačním cyklu nezměnil hodnotu, v opačném případě vrátí **false**

Příklad použití:

```

-- deklarace
signal data : std_logic;

process
  variable a : boolean;
begin
  wait for 30 ns;
  data <= '1' after 30 ns;
  wait for 10 ns;
  a := data'stable(20 ns); -- true (na data 20 ns zpatky nenastala udalost)
  wait for 30 ns;
  a := data'stable(20 ns); -- false (pred 10 ns nastala na data udalost)
  wait;
end process;

```

S'quiet[(t)]

Popis: vrátí booleovskou hodnotu **true**, pokud byl signál *S* po dobu *t* neaktivní (nastala-li na signálu transakce), v opačném případě vrátí hodnotu **false**.

Parametr: výraz, jehož výsledek je nezáporný a je typu **time**

Výsledek: nový signál typu **boolean**

Poznámka: pokud je *t* vynecháno: vrátí **true**, pokud byl signál v daném simulačním cyklu aktivní, v opačném případě vrátí **false**.

S'transaction

Popis: při každém výskytu transakce (aktivitě) na signálu *S* mění hodnotu (každá transakce na signálu *S* způsobí negaci výsledku oproti předchozí hodnotě).

Výsledek: nový signál typu **bit**

S'event

Popis: vrátí booleovskou hodnotu **true**, pokud nastala na signálu *S* v daném simulačním cyklu událost (signál *S* změnil hodnotu).

Výsledek: hodnota typu **boolean**

Příklad použití:

```
-- deklarace
signal clk, data, q : std_logic;

process (clk)
begin
    -- zapis do q se provede pokud signal clk změnil hodnotu
    -- a jeho současná hodnota je '1' (tj. nastala-li na clk vstupna hrana)
    if (clk'event and clk = '1') then
        q <= data;
    end if;
end process;
```

S'last_event

Popis: vrátí dobu od poslední události na signálu *S*.

Výsledek: hodnota typu **time**

Poznámka: pokud událost nikdy nenastala, vrátí hodnotu 0 ns

S'last_value

Popis: vrátí hodnotu signálu *S* před jeho poslední změnou.

Výsledek: hodnota typu jako je základní typ prefixu *S*

Poznámka: pokud změna hodnoty (událost) signálu nikdy nenastala, vrátí současnou hodnotu signálu.

S'driving

Popis: vrátí booleovskou hodnotu **true**, pokud je signál *S* buzen, v opačném případě vrátí hodnotu **false**

Výsledek: hodnota typu **boolean**

S'driving_value

Popis: vrátí budící hodnotu signálu *S*

Výsledek: hodnota typu, jako je základní typ prefixu *S*

3.11.4 Atributy pojmenovaných objektů

Předdefinované atributy určené k získávaní informací o pojmenovaných objektech jsou celkem 3.

O'simple_name

Popis: vrátí řetězec se jménem objektu

Výsledek: hodnota typu **string**

Poznámka: jako prefix *O* lze použít jakýkoliv pojmenovaný objekt z následujících objektů (**architecture**, **component**, **configuration**, **constant**, **entity**, **file**, **function**, **group**, **label**, **literal**, **package**, **procedure**, **signal**, **subtype**, **type**, **variable**, **units**)

O'instance_name

Popis: vrátí řetězec obsahující celou hierarchii cesty k objektu *O* od kořenového objektu (tzv. top level, většinou se jedná o entitu) včetně jeho jména objektu *O*.

Výsledek: hodnota typu **string**

O'path_name

Popis: vráti řetězec obsahující celou hierarchii cesty k objektu O od kořenového objektu (bez jména objektu O)
Výsledek: hodnota typu **string**

3.11.5 Uživatelem definované atributy

V jazyce VHDL lze definovat uživatelské atributy. Uživatelem definované atributy se používají například pro ovládání chodu syntezátorů při syntéze VHDL kódu nebo simulátorů při simulaci. Ovládání chodu syntezátorů řeší výrobci těchto nástrojů dvěma způsoby. Buď pomocí uživatelských atributů, nebo pomocí komentářů. Ovládání chodu syntezátoru pomocí komentářů je uvedeno v podkapitole 3.3.

Syntaxe definice uživatelského atributu:

```
attribute ID : TYPEID;
```

Syntaxe deklarace uživatelského atributu:

```
attribute ATTRID of OBJID : objekt {, objekt} | others | all is vyraz;
```

Element *objekt* může být jakýkoliv z následujících objektů: **architecture**, **component**, **configuration**, **constant**, **entity**, **file**, **function**, **group**, **label**, **literal**, **package**, **procedure**, **signal**, **subtype**, **type**, **variable**, **units**.

Pro příklad uvedeme použití atributu **enum_encoding**. Tento uživatelský atribut je používán syntezátory (např. firmy Synopsys) pro definici binárního kódování hodnot výčtových typů.

Příklady:

```
-- definice atributu
attribute enum_encoding : string;
-- definice výctového typu barvy
type barvy is (CERNA, CERVENA, ZELENA, MODRA, SVMODRA, FIALOVA, ZLUTA, BILA);
-- deklarace atributu k typu barvy
attribute enum_encoding of barvy : type is "000 100 010 001 011 101 110 111";

-- definice výctového typu stav_y_automatu
type stav_y_automatu is (ST0, ST1, ST2, ST3);
-- deklarace atributu k typu stav_y_automatu
attribute enum_encoding of stav_y_automatu : type is "01 10 00 11";
```

Pokud syntezátor, který podporuje atribut **enum_encoding** narazí při syntéze na tento atribut, zakóduje hodnoty výčtového typu podle uvedených binárních hodnot. Pokud na atribut narazí syntezátor, který nepodporuje tento atribut nebo je kód simulován v simulátoru, jednoduše tento atribut ignoruje. Výrobci syntezátorů i simulátorů definují další uživatelské atributy, jimž lze ovládat běh jejich syntezátorů a simulátorů. Jména a funkce těchto uživatelských atributů lze nalézt v příslušné dokumentaci od výrobců těchto softwarových nástrojů.

3.12 Knihovny a knihovní balíky

V knihovně se mohou nacházet tzv. primární a sekundární návrhové jednotky. Mezi primární jednotky patří **entity**, **package** a **configuration**, mezi sekundární jednotky patří **architecture** a **package body**. Primární jednotka **configuration** sekundární jednotku nemá.

Primární a jím odpovídající sekundární jednotky se musí nacházet ve stejné knihovně. Primární jednotka **entity** může mít v knihovně více odpovídajících sekundárních jednotek **architecture**. V případě výskytu více sekundárních jednotek k primární jednotce je třeba provést výběr sekundární jednotky v jednotce **configuration**. Pokud není výběr proveden, případně se k primární jednotce poslední překládaná odpovídající sekundární jednotka. V knihovně se většinou nachází **package** tzv. knihovní balíky (někdy jsou v české terminologii označovány jako slohy), případně **package body**. Knihovny se připojují pomocí příkazu **library**, knihovní balíky se zpřístupňují pomocí příkazu **use**.

Příkazem **library** lze připojit jednu nebo více knihoven najednou jedním příkazem.

Syntaxe příkazu library:

```
library jmeno_knihovny [{, jmeno_knihovny}];
```

Pomocí příkazu **use** lze zpřístupnit buď obsah celého jednoho knihovního balíku, a nebo pouze uvedený jednotlivý objekt z knihovního balíku. Pokud potřebujeme zpřístupnit obsah více knihovních balíků z jedné knihovny, je třeba použít příkaz **use** několikrát za sebou.

Syntaxe příkazu use:

```
use jmeno_knihovny.jmeno_baliku.[all | identifikator_nebo_operator];
```

Příklad použití:

```
library ieee; -- pripoji knihovnu ieee
use ieee.std_logic_1164.all; -- zpristupni cely knihovni balik std_logic_1164
use ieee.numeric_std.abs; -- zpr. funkci abs z knihovniho baliku numeric_std
```

Jazyk VHDL definuje dvě třídy knihoven, a to knihovnu pracovní – **work** a knihovny zdrojové. Pracovní knihovna může být připojena vždy jen jedna a jsou do ní automaticky ukládány zpracovávané (překládané) objekty. Zdrojových knihoven může být připojen neomezený počet. Standard jazyk VHDL definuje předdefinovanou zdrojovou knihovnu **std**. Knihovna **std** obsahuje dva knihovní balíky **standard** a **textio**. Každá návrhová jednotka má implicitně zpřístupněné knihovny **work** a **std** způsobem, jako kdyby obsahovala následující příkazy:

```
library work, std; -- pripoji knihovnu work a std
use std.standard.all; -- zpristupni cely knihovni balik standard
```

Knihovny **work** a **std** jsou tedy vždy automaticky připojeny a není je tedy třeba připojovat. Z knihovny **std** je automaticky zpřístupněn obsah celého balíku **standard**, není ho tedy třeba zpřístupňovat pomocí příkazu **use**. Naopak balík **textio** není automaticky zpřístupněn a je ho třeba v případě potřeby zpřístupnit pomocí příkazu **use**. Knihovní balík **standard** v sobě obsahuje definice všech předdefinovaných typů a podtypů jazyka VHDL (viz pokapitola 3.6) Knihovní balík **textio** v sobě obsahuje definice a deklarace typů, souborů, procedur a funkcí pro pokročilejší práci s textovými řetězci. Tento balík definuje hlavně procedury pro formátovaný zápis různých datových typů do textového souboru, a procedury pro čtení různých datových typů z textových souborů.

Existuje také několik knihoven a knihovních balíků různých firem, zabývajících se vývojem návrhových a simulačních systémů (např. od *Synopsys* a *Mentor Graphics*), a dále také knihovních balíků standardizovaných organizací IEEE. V první řadě zde jmenujme alespoň ty nedůležitější knihovní balíky, které jsou při návrhu číslicových systémů nejvíce používány a které jsou standardizovány organizací IEEE. Všechny tyto balíky jsou sdruženy v knihovně **ieee**. Knihovna se tedy připojuje pomocí příkazu „**library ieee;**“.

- Knihovní balík **std_logic_1164** – obsah tohoto balíku je definován ve standardu *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)* (*IEEE Std 1164-1993*).
- Knihovní balík **numeric_bit** a **numeric_std** – obsah tohoto balíku je definován ve standardu *IEEE Standard VHDL Synthesis Packages* (*IEEE Std 1076.3-1997*).
- Knihovní balík **math_real** – obsah tohoto balíku je definován ve standardu *IEEE Standard VHDL Language Math Packages* (*IEEE Std 1076.2-1996*).

Dále existují knihovní balíky **std_logic_arith**, **std_logic_unsigned** a **std_logic_signed** od firmy *Synopsys*, definující celočíselné matematické operace nad typem **std_logic_vector**. Tyto knihovní balíky vznikly v době, kdy ještě neexistovaly standardizované knihovní balíky **numeric_bit** a **numeric_std**. Tyto balíky byly v minulosti zařazeny do knihovny IEEE, nejsou však standardizovány organizací IEEE. Vzhledem k tomu, že dnes již existují knihovní balíky **numeric_bit** a **numeric_std**, které definují veškeré aritmetické a logické operace nad typy **unsigned** a **signed**, doporučuje se důsledně používat pouze tyto standardizované balíky. S knihovními balíky **std_logic_arith**, **std_logic_unsigned** a **std_logic_signed** se lze často setkat např. ve zdrojových textech starších číslicových návrhů nebo ve starší literatuře. V nových návrzích číslicových systémů se používá prakticky výhradně knihovní balík **numeric_std**, který definuje matematické a logické operace nad typy **unsigned** a **signed**, které jsou odvozeny od typu **std_logic**. Pro všechny výše jmenované knihovní balíky (vyjma balíku **math_real**, který definuje matematické operace v plovoucí řádové čárce), umí dnešní syntezátory provést syntézu do hardware. Příští revize jazyka VHDL, která je očekávána v blízké době, by měla přinést syntetizovatelné knihovní balíky pro podporu matematických operací v pevné řádové čárce (balík **fixed_pkg**) a plovoucí řádové čárce (balík **fphdl_pkg**). Vývojové verze zdrojových textů těchto dvou balíků jsou již v současné době volně dostupné na internetu a lze je využívat pro simulaci a některé návrhové systémy dokonce zvládají částečně i jejich syntézu.

Většina návrhových systémů a simulátorů obsahuje zdrojové texty knihovních balíků, ve kterých je uvedena alespoň deklarační část **package**. Nahlédnutím do těchto zdrojových textů lze snadno zjistit jaké typy, podtypy, konverzní funkce, atd. balíky obsahují i bez přístupu k IEEE standardům, definujícím obsah těchto knihovních balíků. Uvedené platí i pro knihovní balíky firem zabývajících se vývojem návrhových nástrojů a simulátorů. Nejvíce používaným knihovním balíkům **std_logic_1164** a **numeric_std**, a dále knihovně parametrizovatelných modulů **lpm** se podrobněji věnují následující podkapitoly.

3.12.1 Knihovní balík **std_logic_1164**

V tomto knihovním balíku jsou definovány typy **std_ulogic**, **std_logic**, **std_ulogic_vector** a **std_logic_vector**. Typ **std_ulogic_vector** je jednorozměrné neomezené pole typu **std_ulogic**. Podobně **std_logic_vector** je jednorozměrné neomezené pole typu **std_logic**. Typy **std_ulogic** a **std_logic** jsou výčtové typy, jenž definují devět hodnot. Kromě klasických logických hodnot '0' a '1' mohou tyto typy nabývat také hodnot 'U', 'X', 'Z', 'W', 'L', 'H', '-'. Typ **std_ulogic** nedisponuje tzv. rezoluční funkcí a z tohoto důvodu na signál tohoto typu nelze připojit více budičů najednou. Rezoluční funkce slouží k řešení konfliktů v případě, že je signál buzen více zdroji. Typ **std_logic** je definován jako podtyp typu **std_ulogic** s definovanou rezoluční

funkcí. Díky tomu lze na signál tohoto typu připojit více budičů najednou. Rezoluční funkce pak provádí výpočet hodnoty signálu v případě, že je signál buzen více budiči. Hodnoty '0' a '1' reprezentují tzv. tvrdé logické hodnoty (dodávané zdrojem signálu s malým výstupním odporem). Hodnota 'U' reprezentuje hodnotu proměnné nebo signálu, který nebyl doposud buzen (nebyla do něj ještě zapsána žádná hodnota). Tato hodnota je implicitní počáteční hodnota. Hodnota 'X' reprezentuje neznámou hodnotu. Tato hodnota vzniká při konfliktu. To je např. ve chvíli, kdy je signál buzen ze dvou tvrdých zdrojů signálu, přičemž každý zdroj dává jinou logickou hodnotu ('0' a '1'). Hodnoty 'L' a 'H' reprezentují tzv. měkké logické hodnoty (dodávané zdrojem signálu s větším výstupním odporem). Hodnota 'W' reprezentuje tzv. měkkou neznámou hodnotu. Tato hodnota vzniká opět při konfliktu, ale pro signál buzený ze dvou měkkých zdrojů signálu, přičemž každý zdroj dává jinou logickou hodnotu ('L' a 'H'). Hodnota 'Z' reprezentuje vysoko impedanční stav zdroje signálu. Poslední hodnota '-' vyjadřuje, že na dané hodnotě nezáleží. Tuto hodnotu lze s výhodou použít v situacích, kdy může zjednodušit výsledné rovnice, pravdivostní tabulky, apod.

Definice výčtového typu std_ulogic z balíku std_logic_1164:

```
type std_ulogic is (
    'U', -- Neinicializovano (tuto hodnotu má signál který nebyl dosud buzen)
    'X', -- Neznáma hodnota (hodnota vznika např. konfliktem '0' a '1')
    '0', -- Log. 0 z tvrdého zdroje
    '1', -- Log. 1 z tvrdého zdroje
    'Z', -- Vysoká impedance
    'W', -- Neznáma hodnota (hodnota vznika např. konfliktem 'L' a 'H')
    'L', -- Log. 0 z měkkého zdroje
    'H', -- Log. 1 z měkkého zdroje
    '-'
);
```

Následující výpis kódu ukazuje definici rezoluční tabulky, podle níž rezoluční funkce provádí výpočet hodnoty pro typ **std_logic** v případě připojení více budičů.

```
type stdlogic_table is array (std_ulogic, std_ulogic) of std_ulogic;

constant resolution_table : stdlogic_table := (
    -- U   X   0   1   Z   W   L   H   -
    ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- U
    ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- X
    ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- 0
    ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- 1
    ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- Z
    ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- W
    ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- L
    ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- H
    ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- -
);
```

V knihovním balíku **std_logic_1164** jsou dále definovány typy **X01**, **X01Z**, **UX01** a **UX01Z**. Jedná se o podtypy typu **std_ulogic** používající, jak již z jejich názvu vyplývá, pouze určité podmnožiny hodnot typu **std_ulogic**. Všechny tyto podtypy mají definovanou rezoluční funkci a jejich použití je stejně jako použití typu **std_logic**.

V balíku jsou dále definovány všechny přetížené logické operátory (tj. **and**, **nand**, **or**, **nor**, **xor**, **xnor** a **not**) pro typy a podtypy **std_ulegic**, **std_ulegic_vector** a **std_logic_vector**. Dále jsou v balíku definovány konverzní funkce mezi předdefinovanými typy **bit** a **bit_vector** a balíkem definovanými typy a podtypy **std_ulegic**, **std_ulegic_vector** a **std_logic_vector**. Dále pak také konverzní funkce mezi typy a podtypy **bit**, **bit_vector**, **std_ulegic**, **std_ulegic_vector**, **std_logic_vector** a podtypy **X01**, **X01Z** a **UX01**. V knihovním balíku jsou dále definovány funkce **faling_edge(s)** a **rising_edge(s)**, určené pro detekci vzestupné a sestupné hrany na signálu typu **std_ulegic**.

3.12.2 Knihovní balík numeric_std

V tomto knihovním balíku jsou definovány typy **unsigned** a **signed**. Tyto dva nové typy jsou neomezená pole typu **std_logic**. Knihovní balík definuje nad těmito typy celočíselné matematické operace, dále logické operace, operace rotací a posunu, konverzní funkce.

Knihovní balík numeric_std definuje následující funkce a přetížené matematické operátory:

```

abs(arg)    -- absolutni hodnota (arg je typu signed, vysledek je typu signed)
-(arg)      -- zmena znamenka (arg je typu signed, vysledek je typu signed)
+          -- scitani (operand typu unsigned, natural, signed, integer)
-          -- odcitani (operand typu unsigned, natural, signed, integer)
*          -- nasobeni (operand typu unsigned, natural, signed, integer)
/          -- deleni (operand typu unsigned, natural, signed, integer)
rem        -- zbytek po deleni (operand typu unsigned, natural, signed, integer)
mod        -- deleni modulo (operand typu unsigned, natural, signed, integer)

```

Dále definuje přetížené relační operátory:

```

>          -- vetsi (operand typu unsigned, natural, signed, integer)
<          -- mensi (operand typu unsigned, natural, signed, integer)
<=         -- mensi nebo rovno (operand typu unsigned, natural, signed, integer)
>=         -- vetsi nebo rovno (operand typu unsigned, natural, signed, integer)
=          -- rovno (operand typu unsigned, natural, signed, integer)
/=         -- ruzne (operand typu unsigned, natural, signed, integer)

```

Dále definuje přetížené funkce pro posuny a rotace:

```

shift_left (arg, cnt)   -- posun vlevo (arg : unsigned/signed, cnt : natural)
shift_right (arg, cnt)  -- posun vpravo (arg : unsigned/signed, cnt : natural)
rotate_left (arg, cnt)  -- rotace vlevo (arg : unsigned/signed, cnt : natural)
rotate_right (arg, cnt) -- rotace vpravo (arg : unsigned/signed, cnt : natural)
sll (arg, cnt)         -- posun vlevo (arg : unsigned/signed, cnt : natural)
srl (arg, cnt)         -- posun vpravo (arg : unsigned/signed, cnt : natural)
rol (arg, cnt)         -- rotace vlevo (arg : unsigned/signed, cnt : natural)
ror (arg, cnt)         -- rotace vpravo (arg : unsigned/signed, cnt : natural)

```

Dále definuje přetíženou funkci pro změnu počtu bitů:

```
resize (arg, size)  -- zmena poctu bitu (arg : unsigned/signed, size : natural)
```

Dále definuje konverzní funkce:

```

to_integer (arg)      -- konverze na integer/positive (arg : signed/unsigned)
to_unsigned (arg, size) -- konverze na unsigned (arg, size : positive)
to_signed (arg, size)  -- konverze na signed (arg : integer, size : positive)

```

Dále definuje přetížené logické operátory:

```
not      -- log. negace (operand typu unsigned/signed)
and     -- log. součin (operandy typu unsigned/signed)
or       -- log. součet (operandy typu unsigned/signed)
nand    -- log. negovaný součin (operandy typu unsigned/signed)
nor     -- log. negovaný součet (operandy typu unsigned/signed)
xor     -- log. ex-or (operandy typu unsigned/signed)
xnor    -- log. negovaný ex-or (operandy typu unsigned/signed)
```

Stejné funkce a operátory definuje také knihovní balík **numeric_bit**, tento balík však definuje typy **unsigned** a **signed** jako neomezené pole typu **bit**. Jak již bylo řečeno na začátku kapitoly o knihovnách, standardizované knihovní balíky **numeric_bit** a **numeric_std** plně nahrazují nestandardizované knihovní balíky **std_logic_arith**, **std_logic_unsigned** a **std_logic_signed** od firmy Synopsys. Pro nové návrhy se doporučuje používat pouze knihovnu **numeric_std**.

3.12.3 Knihovna LPM

Při návrhu číslicového systému je velmi užitečné, aby návrhář vytvořil systém, který je co možná nejvíce technologicky nezávislý. Dosáhnout toho se současnými EDA (angl. *Electronic Design Automation*) návrhovými nástroji není jednoduché. Schematický návrh číslicového systému návrhovým nástrojem je poměrně rychlý a efektivní, protože je realizován funkcemi, kterými disponuje použitý návrhový systém. Takto realizovaný systém je ovšem složen z technologicky závislých funkcí daného návrhového prostředku. Migrovat s takto navrženým systémem mezi různými cílovými technologiemi pak může být značně problematické. Naproti tomu použití pouze HDL jazyků, jako VHDL nebo Verilog, dává návrháři velkou technologickou nezávislost, ovšem za cenu prodloužení doby návrhu. Překlenut tuto mezeru mezi technologickou nezávislostí a efektivitou je velmi obtížné, protože zde dříve neexistovala množina standardních funkcí, kterou by podporovali všichni výrobci EDA nástrojů a výrobci obvodů. Právě tyto důvody vedly k vytvoření knihovny parametrizovatelných modulů (angl. *Library of Parameterized Modules – LPM*) do standardu **EDIF** (angl. *Electronic Design Interface Format*).

Standard LPM byl navržen v roce 1990 jako jedna z možností pro efektivní návrh číslicových systémů do odlišných technologií, jako jsou např. obvody PLD, hradlová pole a standardní buňky. Předběžná verze standardu vyšla poprvé v roce 1991, další úprava předběžné verze pak v roce 1992. Standard byl přijat organizací EIA (angl. *Electronic Industries Alliance*) v dubnu roku 1993 jako doplněk do standardu EDIF.

EDIF je formát pro přenos návrhu mezi návrhovými nástroji různých výrobců. Formát EDIF popisuje syntaxi, která reprezentuje logický netlist. LPM do něj pak přidává množinu funkcí, která popisuje logické operace netlistu. Před rozšířením o LPM musel každý EDIF netlist typicky obsahovat technologicky specifické logické funkce, které zabraňovaly tomu, aby byl návrh ve větší míře nezávislý na cílové technologii.

V současné době je standard LPM podporován většinou velkých výrobců nástrojů, zde můžeme jmenovat firmy jako je Cadence, Mentor Graphics, Viewlogic a Intergraph. Z největších výrobců programovatelných logických obvodů podporují v současné době standard LPM firmy Altera, Actel a Lattice Semiconductor.

Poslední verze standardu (LPM 2 2 0) obsahuje 29 parametrizovatelných funkcí, jako jsou hradla, čítače, multiplexory, klopné obvody, aritmetická a paměťové funkce. Do dalších verzí standardu budou snad časem přibývat i složitější funkce z oblasti číslicového zpracování signálu, atd. Většina funkcí je parametřitelná několika parametry (bitová šířka, zobrazení dat, zřetězení, atd). Díky tomu lze ve výsledku např. jedinou funkcí (entitou) realizující čítač *LPM_COUNTER* realizovat přes 30 druhů čítačů známých z řady 74xxx. Knihovna je dostupná zdarma a je volně šířitelná.

Následující kus kódu ukazuje deklaraci entity *LPM_COUNTER* z knihovny LPM:

```
entity LPM_COUNTER is
  generic (
    LPM_WIDTH : positive;
    LPM_MODULUS : natural := 0;
    LPM_DIRECTION : string := "UNUSED";
    LPM_AVALUE : string := "UNUSED";
    LPM_SVALUE : string := "UNUSED";
    LPM_PVALUE : string := "UNUSED";
    LPM_TYPE: string := L_COUNTER;
    LPM_HINT : string := "UNUSED"
  );
  port (
    DATA : in std_logic_vector (LPM_WIDTH - 1 downto 0):= (others => '0');
    CLOCK : in std_logic;
    CLK_EN : in std_logic := '1';
    CNT_EN : in std_logic := '1';
    UPDOWN : in std_logic := '1';
    SLOAD : in std_logic := '0';
    SSET : in std_logic := '0';
    SCLR : in std_logic := '0';
    ALOAD : in std_logic := '0';
    ASET : in std_logic := '0';
    ACLR : in std_logic := '0';
    CIN : in std_logic := '0';
    COUT : out std_logic := '0';
    Q : out std_logic_vector (LPM_WIDTH - 1 downto 0)
  );
end LPM_COUNTER;
```

Parametrisace generických konstant při použití komponentu lze realizovat požadované funkce čítače (měnit bitovou šířku čítače, realizovat funkci modulo, měnit směr čítání, definovat hodnoty čítače po resetu, nastavení, atd.)

Funkce realizované v knihovně LPM lze rozdělit do následujících kategorií:

Kategorie hradla:

<i>LPM_CONSTANT</i>	-- parametřitelná konstantní hodnota
<i>LPM_INV</i>	-- parametřitelný invertor
<i>LPM_AND</i>	-- parametřitelný AND
<i>LPM_OR</i>	-- parametřitelný OR
<i>LPM_XOR</i>	-- parametřitelný XOR
<i>LPM_BUSTRI</i>	-- parametřitelný obousměrný budič sběrnice
<i>LPM_MUX</i>	-- parametřitelný multiplexer
<i>LPM_DECODE</i>	-- parametřitelný dekodér 1 z N
<i>LPM_CLSHIFT</i>	-- parametřitelný logický / aritmetický posuv nebo rotace

Kategorie aritmetika:

LPM_COUNTER	-- parametrizovatelný čítač
LPM_ADD_SUB	-- parametrizovatelná přepínatelná sčítáčka/odčítáčka
LPM_COMPARE	-- parametrizovatelné porovnání
LPM_ABS	-- parametrizovatelná absolutní hodnota
LPM_MULT	-- parametrizovatelná násobička
LPM_DIVIDE	-- parametrizovatelná dělička

Kategorie paměti:

LPM_LATCH	-- parametrizovatelný latch
LPM_FF	-- parametrizovatelný klopný obvod typu D nebo T
LPM_SHIFTREG	-- parametrizovatelný posuvný registr
LPM_ROM	-- parametrizovatelná paměť typu ROM
LPM_RAM_DQ	-- parametrizovatelná paměť typu RAM s oddělenými vstupy a výstupy
LPM_RAM_IO	-- parametrizovatelná paměť typu RAM s jedním vstupem/výstupem
LPM_RAM_DP	-- parametrizovatelná dvoubránová paměť typu RAM
LPM_FIFO	-- parametrizovatelná paměť typu FIFO s jedněmi hodinami
LPM_FIFO_DC	-- parametrizovatelná paměť typu FIFO s dvěma hodinami

Kategorie pravdivostní tabulky:

LPM_TTABLE	-- parametrizovatelná pravdivostní tabulka
LPM_FSM	-- parametrizovatelný stavový automat

Kategorie vstupní a výstupní piny:

LPM_INPAD	-- parametrizovatelný vstupní pin
LPM_OUTPAD	-- parametrizovatelný výstupní pin
LPM_BIPAD	-- parametrizovatelný obousměrný pin

Praktické použití knihovny LPM lze ukázat například na následujícím jednoduchém příkladu. Příklad realizuje parametrizovatelný generátor Fibonacciho posloupnosti (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...). Z důvodu zjednodušení příkladu je Fibonacciho posloupnost generována od jejího třetího člena. Generátor je tvořen dvěma registry s asynchronním nulováním a sčítáčkou. Všechny tři funkce jsou realizovány pomocí komponentů z knihovny LPM a mají nastavitelnou bitovou šířku pomocí generické konstanty *sirka*, která má nastavenou implicitní hodnotu na 16. První registr je signálem *reset* nastavován na hodnotu 0, druhý registr pak na hodnotu 1. Zápis do registrů se provádí vzestupnou hranou signálu *clk*. Generátor signalizuje pomocí signálu *preteceni* přetečení rozsahu výstupu.

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

library lpm;
use lpm.lpm_components.all;

entity fibex is
  generic (
    sirka : positive := 16
  );
  port (
    clk : in std_logic;
    reset : in std_logic;
    vystup : out std_logic_vector (sirka - 1 downto 0);
    preteceni : out std_logic
  );
end;
```

```

architecture struct of fibex is
    signal minula : std_logic_vector (sirka - 1 downto 0);
    signal soucasna : std_logic_vector (sirka - 1 downto 0);
    signal vysledek : std_logic_vector (sirka - 1 downto 0);
begin
    u1 : lpm_ff
        generic map (lpm_width => sirka, lpm_avalue => "0")
        port map (data => vysledek (sirka - 1 downto 0), clock => clk, aset =>
            reset, q => minula);

    u2 : lpm_ff
        generic map (lpm_width => sirka, lpm_avalue => "1")
        port map (data => minula, clock => clk, aset => reset, q => soucasna);

    u3 : lpm_add_sub
        generic map (lpm_width => sirka)
        port map (dataa => soucasna, datab => minula, result => vysledek,
            cout => preteci);

    vystup <= vysledek;
end;

```

Výpis 3.6 Paramatrizovatelný generátor Fibonacciho posloupnosti pomocí knihovny LPM

Další informace o knihovně LPM lze nalézt na internetu, viz seznam použité literatury.

3.12.4 Uživatelské knihovny a knihovní balíky

Jak bylo zmíněno již v kapitole o knihovnách, lze v jazyce VHDL definovat uživatelské knihovní balíky a ty umístit do uživatelských knihoven. Knihovní balík lze definovat pomocí příkazu **package** a **package body**. Příkaz **package** definuje deklarace a hlavičky procedur a funkcí obsažených v knihovním balíku. Příkaz **package body** pak definuje vlastní tělo procedur a funkcí obsažených v knihovním balíku. Jméno knihovního balíku v příkazu **package** i **package body** musí být stejné a stejně by se měl jmenovat i soubor, v němž je zdrojový text knihovního balíku uložen.

Syntaxe příkazu package:

```

package jmeno_baliku is
    {deklarace_hlavicek_funkci_a_procedur}
    {deklarace_typu_a_podtypu}
    {deklarace_konstant_a_signalu}
    {deklarace_sdilenych_promennych}
    {deklarace_souboru}
    {deklarace_aliases}
    {deklarace_komponentu}
    {deklarace_atributu}
    {definice_atributu}
end [package] [jmeno_baliku];

```

Ze syntaxe příkazu **package** je zřejmé, že v jeho těle nelze uvést deklaraci entity ani architektury. V příkazu **package** lze uvést pouze deklaraci komponentu, vlastní entita

a architektura nebo architektury (pokud jich je více) již musí být v samostatném zdrojovém souboru nebo mimo sekce **package** a **package body** ve zdrojovém textu knihovního balíku.

Syntaxe příkazu package body:

```
package body jméno_balíku is
    {deklarace_a_tela_hlavicek_funkci_a_procedur}
    {deklarace_typu_a_podtypu}
    {deklarace_konstant}
    {deklarace_sdilencích_promennych}
    {deklarace_souboru}
    {deklarace_aliasu}
end [package body] [jméno_balíku];
```

Příklad deklarace uživatelského knihovního balíku:

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

package seg_lib is
    component segment is
        port (
            data : in std_logic_vector (3 downto 0);
            seg : out std_logic_vector (6 downto 0)
        );
    end component;

    function slv2str (data : std_logic_vector (3 downto 0)) return string;
end;

package body seg_lib is
    function slv2str (data : std_logic_vector (3 downto 0)) return string is
    begin
        case data is
            when x"0" => return "0";
            when x"1" => return "1";
            when x"2" => return "2";
            when x"3" => return "3";
            when x"4" => return "4";
            when x"5" => return "5";
            when x"6" => return "6";
            when x"7" => return "7";
            when x"8" => return "8";
            when x"9" => return "9";
            when others => return "X";
        end case;
    end;
end;
```

Výpis 3.7 Příklad uživatelského knihovního balíku

Ve výše uvedeném knihovním balíku je v sekci **package** definován komponent **segment** a hlavička funkce **slv2str**. V sekci **package body** je pak definováno vlastní tělo funkce **slv2str**. Funkce převádí čtyřbitový binární řetězec na odpovídající znak reprezentující odpovídající BCD hodnotu. Deklarace entity a architektury komponentu **segment** musí být uvedena v souboru s názvem **segment.vhd** nebo **segment.vhdl**. Výpis zdrojového kódu této entity a architektury lze nalézt v kapitole 8 jako výpis 8.15.

Připojení výše uvedeného uživatelského knihovního balíku **seg_lib** a použití v něm definovaného komponentu **segment** a funkce **slv2str** ukazuje následující výpis kódu.

```
library ieee;          -- pripojeni knihovny ieee
use ieee.std_logic_1164.all; -- zpristupneni celeho baliku std_logic_1164
use ieee.numeric_std.all; -- zpristupneni celeho baliku numeric_std
use work.seg_lib.all;   -- zpristupneni celeho baliku seg_lib

entity test is
end test;

architecture a_test of test is
    signal a : std_logic_vector (3 downto 0);
    signal seg : std_logic_vector (6 downto 0);
begin
    -- pouziti komponentu segment (balik seg_lib)
    dut : segment port map (a, seg);

    process
    begin
        for i in 0 to 15 loop
            -- pouziti pretypovani a konverzni funkce (balik numeric_std)
            a <= std_logic_vector (to_unsigned (i, 4));
            report slv2str(a); -- pouziti funkce slv2str (balik seg_lib)
            wait for 10 ns;
        end loop;
        wait;
    end process;
end;
```

Výpis 3.8 Použití uživatelského knihovního balíku

3.13 Testovací prostředí pro ověřování funkčnosti navrženého systému

Již v průběhu vytváření číslicového systému je velmi užitečné vytvářet souběžně také testovací prostředí (v literatuře se lze často setkat s pojmem **testbench**), jímž již v průběhu vývoje systém testujeme a s jehož pomocí v závěru návrhu dojde k ověření funkčnosti celého systému. Právě pro úkol testování číslicových systémů byl jazyk VHDL původně navržen, disponuje tedy pro tento úkol dostatečným množstvím vyjadřovacích prostředků.

Ověření funkčnosti (simulace) číslicového systému lze provádět v jazyce VHDL na dvou úrovních. První, nejjednodušší typ ověření je způsob, kdy testovací program v jazyce VHDL ověřuje pouze správnost hodnot výstupů. Této simulaci říkáme **funkční** simulace a lze ji provádět na dvou úrovních. Druhý způsob ověřuje kromě správnosti hodnot výstupů také dodržení časových parametrů výstupních signálů. Této simulaci říkáme **časová** simulace.

Funkční simulaci lze provádět buď na základě původních zdrojových textů napsaných návrháři číslicového systému (tzv. **simulace před syntézou** – *pre-synthesis simulation*), a nebo na základě zdrojového textu vzniklého po syntéze původních zdrojových textů do hardware (tzv. **simulace po syntéze** – *post-synthesis, post-fit* nebo *post-layout*). Při provádění funkční

simulace před syntézou tedy simulujeme původní zdrojové texty. Tuto simulaci je vhodné provádět v průběhu vytváření číslicového systému, až do doby než systém po funkční stránce vyhoví zadané specifikaci. Při syntéze navrženého číslicového systému do hardware lze u většiny návrhových systémů vygenerovat výsledek syntézy ve formě zdrojového souboru v jazyce VHDL. Výstupem je pak jeden jediný zdrojový soubor v jazyce VHDL, který má stejné jméno jako nejvyšší entita (top-level). Tento soubor má standardizovaný formát, který používá pouze určitou definovanou část jazyka VHDL a má zpravidla příponu *.vho* (angl. *VHDL Output File*). Tento zdrojový textový soubor obsahuje z původního zdrojového textu zpravidla jen deklaraci nejvyšší (top-level) entity. Text původních architektur je nahrazen výsledkem jejich syntézy a obsahuje zpravidla pouze velké množství navzájem propojených komponentů, které odpovídají architektuře cílového obvodu použitého pro syntézu. Připojením tohoto vygenerovaného zdrojového textu k testovacímu prostředí a následnou simulaci lze ověřit funkci navrženého systému, který se již skládá z komponentů cílové architektury. V tomto případě se tedy jedná o **funkční simulaci po syntéze**.

Pokud v návrhovém systému provedeme po syntéze ještě rozmístění a následně časovou analýzu vzniklého propojení, umožňuje většina návrhových systémů následně vygenerovat soubor popisující zpoždění jednotlivých použitých komponent (angl. *Standard Delay Format Output File – SDO*). Tento soubor má zpravidla stejné jméno jako nejvyšší (top-level) entita s příponou *.sdo*. Na základě souborů *.vho*, *.sdo* a původního zdrojového textu testovacího prostředí (testbenche) lze provést tzv. **časovou simulaci** celého systému. Tato simulace dává výsledky, které z výše popsaných simulací nejvíce odpovídají chování vytvořeného číslicového systému v reálném hardware. Tato simulace je ale také nejvíce časově náročná na běh v simulátoru. Pokud navržený číslicový systém dává i po časové simulaci správné výsledky, je velmi vysoká pravděpodobnost, že stejných výsledků bude navržený systém dosahovat i v reálném hardware (např. v CPLD nebo FPGA obvodu).

Následující výpis kódu ukazuje příklad testovacího prostředí (testbench) pro Fibonacci generátor, jehož zdrojový text je uveden v předchozí podkapitole jako výpis 3.6. Testbench generuje pro testovaný číslicový systém vstupy a kontroluje správnost výstupů. Očekávané hodnoty výstupů jsou čteny z binárního souboru. Výstupní hodnoty jsou zapisovány do textového a binárního souboru.

```

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

entity tb_fibex is
end tb_fibex;

architecture behav of tb_fibex is
-- deklarace testovane komponenty
component fibex
generic (
    sirka : positive := 16
);
port (
    clk : in std_logic;
    reset : in std_logic;
    vystup : out std_logic_vector (sirka - 1 downto 0);
    preteceni : out std_logic
);
end component;

```

```

-- deklarace pomocnych signalu a konstanty
constant sirka : positive := 16;
signal clk, clr, overflow : std_logic;
signal result : std_logic_vector (sirka - 1 downto 0);
begin
    -- pouziti a prijmeni testovane komponenty
    dut : fibex port map (clk, clr, result, overflow);

    -- proces vytvarejici testovaci prostredi
process
    type tf_integer is file of integer; -- definice typu soubor (integer)
    type tf_text is file of string; -- definice typu soubor (string)
    -- deklarace vstupniho souboru
    file f_vstup : tf_integer open read_mode is "vstup.bin";
    -- deklarace vystupnich souboru
    file f_vyst_bin : tf_integer open write_mode is "vystup.bin";
    file f_vyst_txt : tf_text open write_mode is "vystup.txt";
    variable test_vysl : integer;
begin
    -- generovani resetu pro testovany komponent
    clk <= '0';
    clr <= '1';
    wait for 100 ns;
    clr <= '0'; -- konec resetu
    while true loop -- testovaci smycka
        -- generovani jedne periody hodin
        clk <= '0';
        wait for 100 ns;
        clk <= '1';
        wait for 100 ns;

    -- test zda komponenta hlasí pretečení, pokud ano - konec testu
    if (overflow = '1') then
        report "Nastalo pretečení, konec simulace." severity note;
        exit;
    end if;

    -- zapis vysledku, ktery vratila komponenta do souboru
    write (f_vyst_bin, to_integer(unsigned(result)));
    write (f_vyst_txt, integer'image(to_integer(unsigned(result))) & " ");

    -- cteni ocekavaneho vysledku ze vstupniho binarniho souboru
    if not endfile (f_vstup) then
        read (f_vstup, test_vysl);
    else
        report "Dosla data ve vstupnim souboru." severity error;
        exit;
    end if;

    -- vysledku z komponentu na ocekavanou hodnotu
    assert test_vysl = to_integer(unsigned(result))
    -- pokud ma vysledek jinou hodnotu, vypis ocekavane hodnoty
    -- a zastaveni simulace
    report "Vysledek nema ocekavanou hodnotu. Ocekavano: " &
        integer'image(test_vysl)

```

```

        severity failure;
end loop;
-- zavreni otevrenych souboru
file_close (f_vstup);
file_close (f_vyst_bin);
file_close (f_vyst_txt);
-- vypis hlaseni, konec simulace
report "Konec simulace" severity note;
wait;
end process;
end behav;

```

Výpis 3.9 Příklad testovacího prostředí (testbenche)

3.14 Seznam klíčových slov jazyka VHDL

V této podkapitole je uveden výčet všech klíčových slov (příkazů) jazyka VHDL. Výčet je pro přehlednost uveden v abecedním pořadí, přičemž na každé řádce jsou klíčová slova začínající na stejné písmeno. Uvedená klíčová slova nelze použít, podobně jako v ostatních jazycích, jako identifikátor, apod.

- abs, access, after, alias, all, and, architecture, array, assert, attribute**
- begin, block, body, buffer, bus**
- case, component, configuration, constant**
- disconnect, downto**
- else, elsif, end, entity, exit**
- file, for, function**
- generate, generic, group, guarded**
- if, impure, in, inertial, inout, is**
- label, library, linkage, literal, loop**
- map, mod**
- nand, new, next, nor, not, null**
- of, on, open, or, others, out**
- package, port, postponed, procedure, process, pure**
- range, record, register, reject, rem, report, return, rol, ror**
- select, severity, shared, signal, sla, sll, sra, srl, subtype**
- then, to, transport, type**
- unaffected, units, until, use**
- variable**
- wait, when, while, with**
- xnor, xor**

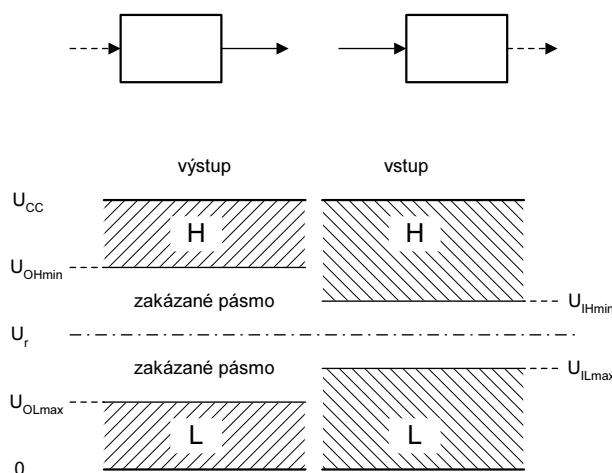
Syntaxe většiny výše uvedených příkazů byla uvedena a vysvětlena v jednotlivých předchozích podkapitolách. Zjednodušená syntaxe většiny příkazů je dále přehledně shrnuta v příloze na konci této knihy.



ČÍSLICOVÉ SOUČÁSTKY A TECHNOLOGIE

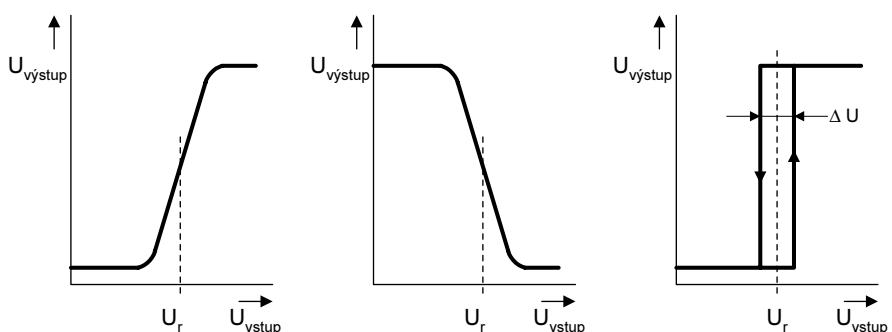
4.1 Vlastnosti číslicových součástek

Číslicové součástky zpracovávají číslicové signály. Pro snadné spojování součástek do sítě bez nutnosti výpočtu a kontroly vstupních a výstupních napětí je zaveden princip **rezervy v úrovňích** H a L mezi výstupem jedné a vstupem následující součástky. Na výstupu je tak definováno napětí $U_{OHmin} > U_{IHmin}$, obdobně $U_{OLmax} < U_{ILmax}$. Při průchodu signálu součástkou ze vstupu na výstup se zakázané pásmo **rozšiřuje**. Docílí se toho jedině využitím aktivního (zesilujícího) prvku uvnitř součástky. Nejdříva se přitom ale o zesilovač v běžném smyslu – k tomu slouží lineární obvody, zatím co číslicové obvody jsou velmi nelineární. Napěťové úrovně ukazuje obr. 4.1.



Obr. 4.1 Vstupní a výstupní napěťové úrovně

Napěťové rezervy garantují činnost obvodů i při krajních a nejpříznivějších hodnotách teplot, napájecího napětí, zátěže výstupů a superponovaného rušení.

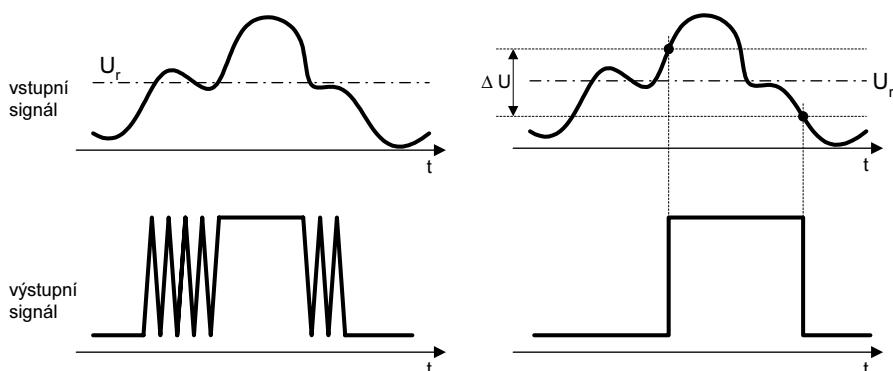


Obr. 4.2 Typické převodní charakteristiky logických členů

Závislost výstupního napětí na napětí vstupním u jednoduché součástky – logického členu – udává **převodní charakteristika**. Na obr. 4.2 vlevo je ukázána charakteristika součástky,

která neneguje vstupní signál, uprostřed charakteristika součástky negující, vpravo pak charakteristika s **hysterezí** (nenegující).

V úzké oblasti těsně kolem U_r se člen chová jako zesilovač se zesílením několika set. Pokud se vstupní signál **mění pomalu** a přes tuto oblast přechází dlouho, člen se může **rozkmitat**. Přitom slovo „dlouho“ zde znamená řádově deset nanosekund. Proto je tak důležitá „kvalita“ signálů, tj. jejich dostatečně strmý průběh. Pokud to nelze zaručit, je třeba použít člen s hysterezní převodní charakteristikou. Ten reaguje tak, že při zvyšování vstupního napětí se při $U_{vst} > U_r + \frac{1}{2}\Delta U$ překlopí do jednoho stavu, ale zpět se překlopí až při $U_{vst} < U_r - \frac{1}{2}\Delta U$, a to vždy velmi rychle. Hystereze je způsobena vnitřní **kladnou zpětnou vazbou**. Vliv pomalu proměnného signálu na člen bez hystereze je vidět na obr. 4.3 vlevo, na člen s hysterezí na obrázku vpravo. **Pomalý průchod** přes rozhodovací úroveň a dokonce ani **překmity** v rozsahu ΔU nevadí.

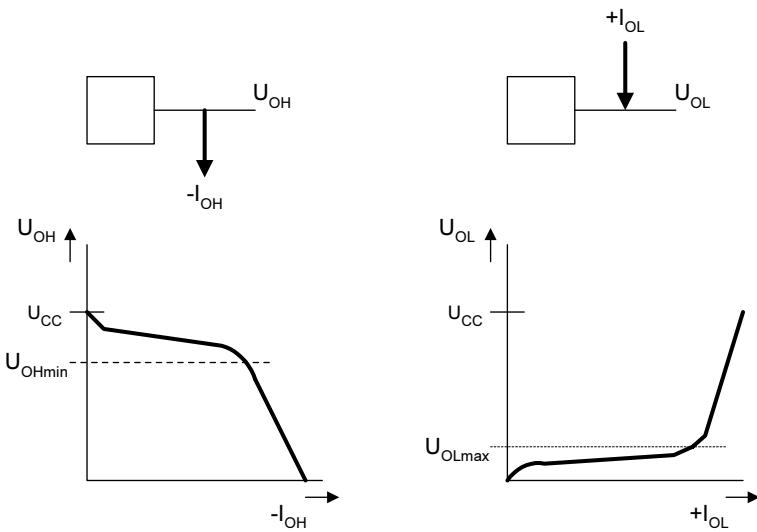


Obr. 4.3 Chování člena bez hystereze a s hysterezí při pomalém signálu

Výstup číslicové součástky je vždy zatížen buď vstupy následujících číslicových součástek, nebo zátěží jiného charakteru. Druhý případ budeme nazývat „**nestandardní zátěž**“. Nestandardní zátěž může být např. rezistor a LED dioda. Počet vstupů číslicových součástek připojených na výstup dané součástky se nazývá „**výstupní rozvětvení**“.

Zátěží součástky se změní napěťové úrovni a též dynamické parametry signálu. V současné době se převážná většina číslicových součástek vyrábí v unipolární technologii (CMOS), a proto se jeví ve většině případů jako zátěž ryze kapacitní. K tomu přispívají ještě parazitní kapacity spojů. **Kapacitní zátěž** neovlivní napětí v ustáleném stavu, zhorší ale **časový průběh** signálu, zvláště pak doby t_{pLH} a t_{pHL} . V tomto případě je výstupní rozvětvení omezeno tím, že systém musí splňovat požadavky na rychlosť. V případě bipolární technologie nelze zanedbat vstupní proudy součástek a proto je výstupní rozvětvení omezeno i zhoršením **napěťových úrovní**, což je dáno vlastnostmi výstupních obvodů zatížené součástky. Je proto nutné znát i průběh výstupních (zatěžovacích) charakteristik číslicových součástek. Obr. 4.4 vlevo ukazuje obecnou závislost výstupního napětí na dodávaném proudu ve stavu H (proud je označován jako záporný) a obrázek vpravo ukazuje závislost výstupního napětí na pohlceném proudu ve stavu L (proud je označován jako kladný).

Pokles výstupního napětí ve stavu H pod U_{OHmin} by znemožnil využití tohoto signálu pro další číslicové členy, i když zátěž nemusí být pro součástku nebezpečná (přehřátím).



Obr. 4.4 Závislost výstupního napětí na výstupním proudu

Obdobně ve stavu L pohlcení proudu může zvýšit výstupní napětí nad U_{OLmax} a tím signál znehodnotit.

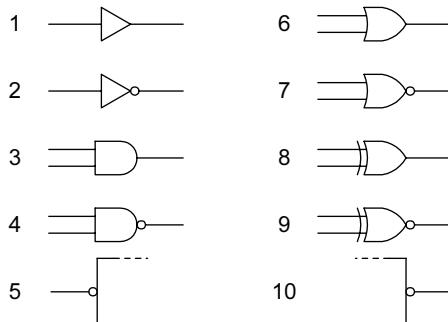
Znalost výstupních charakteristik je nutná při nestandardních zátěžích. Obecně platí, že takovéto zátěž znehodnotí výstupní signál tak, že by neměl být již zaváděn do **dalších vstupů** číslicových součástek. U součástek v bipolární technologii výrobce dodržuje specifikované vstupní proudy součástek a tak postačí dodržet jeho informaci o maximálním výstupním rozvětvení. Týká se vždy součástek **stejné technologie**. Při kombinování součástek různé technologie však je třeba si již všimat vstupních proudů.

U každé číslicové součástky lze rozlišit **tři vnitřní stupně**. Vstupní stupeň obsahuje **ochranné obvody** zabraňující poškození součástky statickými náboji a zajišťuje rozlišení úrovní L a H, druhý stupeň slouží ke **zpracování** informace, a třetí stupeň obsahuje **výstupní obvody**, poskytující správné výstupní úrovně L a H a též patřičné výstupní proudy. Střední stupeň může být velmi jednoduchý (např. u základních logických členů), ale i velmi složitý (např. u mikroprocesoru). První a třetí stupeň je pro danou technologii v podstatě nezávislý na složitosti součástky. U integrovaných obvodů LSI a VLSI v unipolární technologii je zcela běžné, že jednotlivé vnitřní stupně mají **různá napájecí napětí**. Vyšší je pak vždy u vstupního a zvláště u výstupního stupně, nižší je u středního stupně. Tam je totiž soustředěna většina obvodů, u kterých je podstatná rychlosť a úspora příkonu. Moderní technologie toho dosahují při stálém snižování napájecího napětí. Naopak výstupní obvody musí dodávat potřebný proud pro rychlé nabíjení a vybíjení zatěžovacích kapacit a jejich napájecí napětí bývá vyšší. Oddělené přívody napájení (i nuly) jsou dále výhodné pro omezení **vzájemného rušení** vnitřních obvodů.

Výrobci zpravidla neuvádějí vnitřní schéma součástky (kromě elementárních členů to ani není možné), ale běžně uvádějí **náhradní schémata** vstupních a výstupních obvodů a též uvádějí výstupní charakteristiky.

4.2 Značení logických členů

Ve světě se vyskytuje několik způsobů schematického značení logických členů. V tomto textu bude dáná přednost značkám, které se vyskytují nejčastěji, a které dávají nejpřehlednější schémata. Přehled značek základních členů dává obr. 4.5.



Obr. 4.5 Schematické značky elementárních logických členů

Význam jednotlivých symbolů udává tab. 4.1. Jako výstupní signál se předpokládá Y, jako vstupní signály A nebo A a B.

Tab. 4.1 Funkce elementárních logických členů

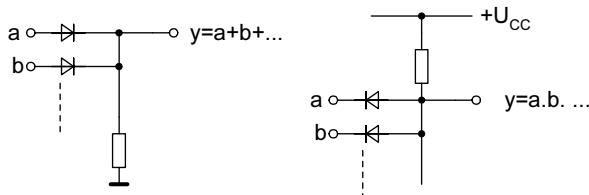
Symbol	Význam	Logická funkce
1	opakovač signálu	$Y = A$
2	invertor (obvod NOT)	$Y = \bar{A}$
3	logický součin (AND)	$Y = A \cdot B$
4	negovaný logický součin (NAND)	$Y = \overline{A \cdot B}$
5	negace na vstupu	
6	logický součet (OR)	$Y = A + B$
7	negovaný logický součet (NOR)	$Y = \overline{A + B}$
8	exkluzivní součet (EX-OR nebo též XOR)	$Y = A \oplus B$
9	negovaný exkluzivní součet (XNOR)	$Y = \overline{A \oplus B}$
10	negace na výstupu	

Opakovač signálu nerealizuje žádnou logickou funkci, ale zpravidla dodává vyšší **výstupní proudy** nebo jiné napěťové úrovně, než jaké jsou na vstupu.

4.3 Bipolární technologie

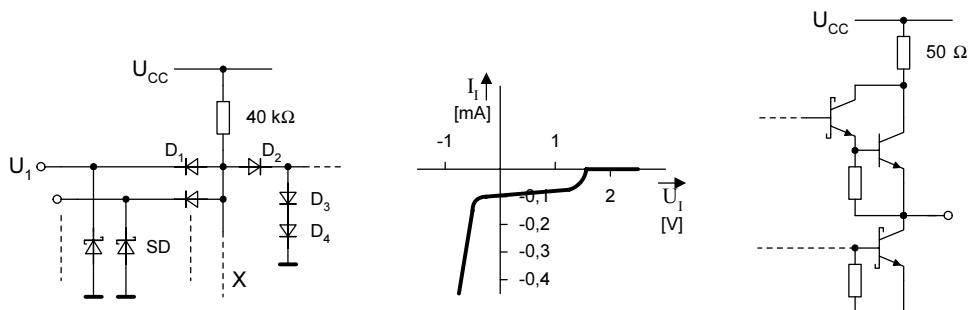
Nejjednodušší je **diodová logika**. Na obr. 4.6 je ukázán člen OR (vlevo) a AND (vpravo). Negace není možná, vyžaduje vždy aktivní prvek (tranzistor), obracející fazu. Jako **samostatné**

členy se v integrované verzi **nepožívají**, jsou však často vnitřní částí větších obvodů. Jako samotné nesplňují základní požadavek na napěťové úrovně číslicové součástky, tj. rozšíření zakázaného pásma při průchodu signálu. U členu OR se sníží U_{OL} o otevřicí napětí diod U_{AK} , u členu AND se naopak zvýší U_{OH} . Funkce je zřejmá. U členu OR postačí stav H na kterémkoliv vstupu, aby na výstupu byl též stav H. U členu AND musí být na všech vstupech stav H, aby na výstupu byl stav H.



Obr. 4.6 Diodový člen OR (vlevo) a AND (vpravo)

Moderní bipolární obvody používají mnohem složitější zapojení. Jedná se o **obvody typu TTL** (tranzistorově-tranzistorová logika), které prošly dlouhým vývojem od konce 60. let 20. století. Jejich zapojení bylo zpočátku jednoduché, časem se však zdokonalilo a zkomplikovalo. Z hlediska uživatele obvodů je jeho detailní znalost nepodstatná, důležitá je však znalost jeho vlastností. Výrobci zpravidla uvádějí jen náhradní schéma těch částí obvodů, které jsou podstatné pro jejich využití. Na obr. 4.7 je typické náhradní schéma vstupního (vlevo) a výstupního (vpravo) obvodu.

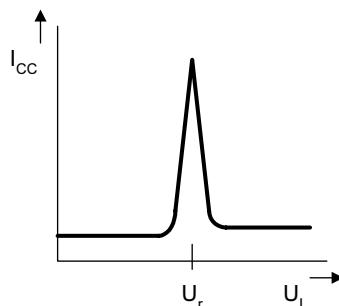


Obr. 4.7 Náhradní schéma vstupu a výstupu obvodu TTL

Náhradní schéma vstupního obvodu ukazuje, že vstupní proud ve stavu H je dán zpětným proudem diody D_1 a je tedy zanedbatelný. Naopak ve stavu L teče proud z napájecího zdroje napětí U_{CC} (typicky $+5\text{ V}$) přes rezistor $40\text{ k}\Omega$ a nyní otevřenou diodu D_1 ven ze vstupu. Jeho velikost je dána rezistorem. U moderních bipolárních technologií je to přibližně $0,1\text{ mA}$. Přesto se s tímto proudem musí počítat – ve výstupním rozvětvení se vstupní proudy sečítají a mohou nepříznivě ovlivnit U_{OL} výstupu předchozí součástky. Z tohoto důvodu je u bipolárních technologií výstupní **rozvětvení vždy omezeno**. Vstupní proud začne téci až při snížení vstupního napětí asi pod $1,4\text{ V}$. Při zavřených diodách D_1 je totiž v bodě X napětí $2,1\text{ V}$, dané třemi přechody P–N v sérii (diody D_2, D_3, D_4 nahrazují v náhradním schématu otevřené přechody tří tranzistorů, tak jak existují ve skutečném obvodu). Při poklesu vstupního

napětí U_I asi o 0,7 V pod napětí v bodě X se otevře dioda D1. To nastane při napětí U_I přibližně 1,4 V. Toto napětí je u bipolárních technologií považováno za **referenční**. Průběh vstupní charakteristiky ukazuje obr. 4.7 uprostřed. Vstupní proud je považován za **kladný**, teče-li **dovnitř** obvodu. Schottkyho diody SD zachycují případné záporné překmity na vstupech, ke kterým by mohlo dojít vlivem delších, impedančně nepřizpůsobených spojů.

Výstupní obvod na obr. 4.7 vpravo je složen ze **dvojí spínače**. V dolní věti je to jednoduchý spínač, v horní věti je typické Darlingtonovo zapojení. Plně otevřené tranzistory představují odpor kolem 10 až 30 Ω . Ve stavu H je však v sérii ještě přídavný odpor (na obrázku o hodnotě 50 Ω), takže vnitřní odpor ve stavu H je vyšší. Přídavný odpor má důležitou funkci, a to omezení špičkového proudu z napájecího zdroje při změně stavu z L do H a opačně. Budící obvody koncových spínačů nejsou zakresleny, u běžných obvodů zajišťují buzení koncových spínačů v protifázi. Žádoucí by bylo absolutně současné vybuzení jedné větve a vypnutí druhé větve. To v praxi není možné a zbývají tak další dvě možnosti – buď se dosud sepnutý spínač vypne dříve, než se zapne spínač druhý (spínače se „nedokrývají“), nebo se dosud sepnutý spínač vypne později, než se zapne spínač druhý (spínače se „překrývají“). První případ má za následek krátkodobě nedefinovaný stav na výstupu, kdy žádný spínač není sepnut a výstupní napětí bude náhodné. Druhý případ nevede k nedefinovanému napětí, ale znamená **krátkodobé zvýšení proudu** z napájecího zdroje při „překrytí“ obou spínačů. Nedefinované stavy nelze připustit a proto je nutné se smířit s menším zlem, tj. s proudovými špičkami během změny stavu. K jejich omezení slouží přídavný odpor. Závislost napájecího proudu I_{CC} na vstupním napětí U_I , tj. **odběrovou charakteristiku**, ukazuje obr. 4.8.



Obr. 4.8 Odběrová charakteristika členu TTL

Odběrové špičky částečně zvyšují celkový příkon obvodu. Zvláště výrazné je to při signálech s **pomalým přechodem** přes referenční úroveň. I další obvody v bipolárním členu však mají nezanedbatelný příkon. Jelikož bipolární tranzistor je řízen proudem, musí všechny stupně v součástce dodat patřičný proud pro vybuzení následujícího stupně. To samozřejmě znamená zvýšený odběr z napájecího zdroje.

Na začátku vývoje stálý „standardní TTL“ obvody, které byly postupně během 80. let vytlačeny obvody **TTL-LS** (Low-power Schottky). U obvodů LS byla snížena spotřeba i vstupní proudy a s využitím Schottkyho antisaturačních diod se podařilo poněkud zkrátit zpoždění. Dalším podstatným zlepšením technologických postupů se u obvodů **TTL-ALS** (Advanced Low-power Schottky) ještě dále snížila spotřeba a zmenšilo zpoždění. V kategorii

bipolárních obvodů jsou v současné době nejvíce používány pro nové konstrukce. Vedle snahy o co nejmenší spotřebu při relativně dobrých dynamických parametrech byla snaha i o vývoj obvodů s minimálním zpožděním bez ohledu na spotřebu. Vznikla tak řada **TTL-S** (Schottky) a později **TTL-AS** (Advanced Schottky). Jistým kompromisem je řada **TTL-F** (Fast), není však používána často. V každé řadě je vyráběno množství různých typů obvodů od jednoduchých členů až po obvody na mezi střední (MSI) a vysoké (LSI) integrace. Každá řada má variantu pro nenáročné podmínky (**řada 74** s rozsahem teplot okolí 0 až 70 °C) a pro náročné podmínky v průmyslu a armádě (**řada 54** s rozsahem teplot okolí -55 až +125 °C). Kromě klimatické odolnosti nejsou mezi variantou 74 a 54 významné rozdíly. Obvody jsou značeny jako 74ALS..., 54ALS..., 74AS..., 54AS..., 74F..., atd. Na místě teček je číslo, vyznačující typ obvodu – např. 74 ALS 00 značí čtverici dvojvstupových členů NAND.

Vlastnosti nejčastějších typů obvodů TTL jsou uvedeny v tab. 4.2. Příkon je přepočítán na jeden elementární člen (NAND, NOR). Rovněž zpoždění jsou udávána pro elementárních členy.

Tab. 4.2 Charakteristické vlastnosti vybraných řad obvodů TTL

Typ	Příkon na 1 člen [mW]	Zpoždění t_{pd} [ns]	Výstupní proud [mA]
74 LS	2	10	-15/24
74 ALS	1	4	-15/64
74 F	4	3	-15/64
74 AS	10	1,5	-15/64

Maximální výstupní proudy jsou uváděny pro členy s největším výstupním proudem, využívané pro připojení na sběrnice (budiče sběrnice). Běžné členy řady mají výstupní proudy několikanásobně nižší. Členy TTL dávají zásadně menší proudy ve stavu H, než jaké mohou pohltit ve stavu L.

U technologie TTL je **standardní napájecí napětí 5 V**. Napětí pro otevření přechodu B-E bipolárního tranzistoru je vždy kolem 0,7 V a nelze je technologickými postupy významně změnit. Od tohoto napětí a základního zapojení TTL členu se dojde k napájecímu napětí, které také nelze významně změnit.

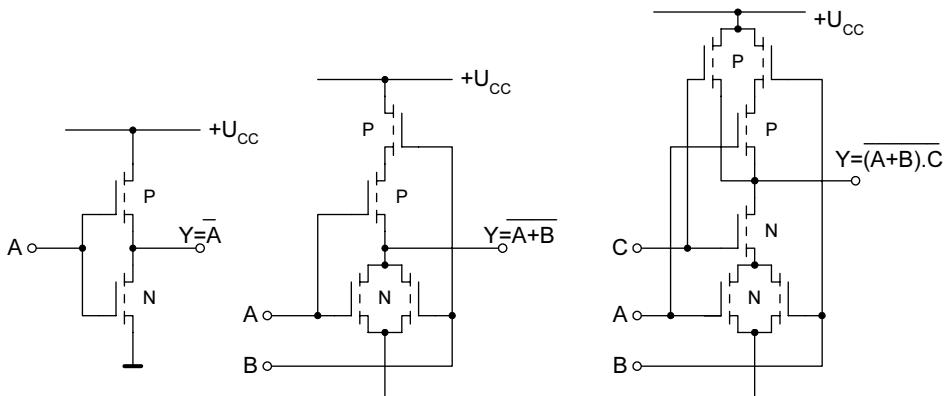
Úrovně U_L a U_H jsou u TTL obvodů také standardní, a to:

$$\begin{aligned} U_{OL\max} & \dots & 0,4 \text{ V} \\ U_{OH\min} & \dots & 2,4 \text{ V} \\ U_{IL\max} & \dots & 0,8 \text{ V} \\ U_{IH\min} & \dots & 2,0 \text{ V} \\ U_r & \dots & 1,4 \text{ V} \end{aligned}$$

Napěťová rezerva ve stavu L i H je u TTL technologie 0,4 V.

4.4 Unipolární technologie – CMOS

Unipolární obvody jsou v současné době používány jedině ve variantě **CMOS** (Complementary MOS). Jedná se o kombinaci tranzistorů MOS s kanály N a P. Základní zapojení invertoru je na obr. 4.9 vlevo, členu NOR na obrázku uprostřed a členu realizujícího funkci $(A+B) \cdot \bar{C}$ na obrázku vpravo.



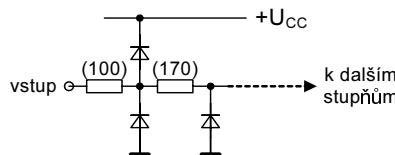
Obr. 4.9 Základní zapojení v technologii CMOS

U invertoru je úrovní H (tj. kladným napětím) na vstupu A otevřen kanál dolního tranzistoru NMOS, ale současně je zavřen kanál horního tranzistoru PMOS (je jen malé nebo nulové napětí mezi jeho hradlem a kanálem P). Opačně je tomu při úrovni L na vstupu – spodní tranzistor NMOS nevede (je jen malé nebo nulové napětí mezi jeho hradlem a kanálem N) a naopak horní tranzistor PMOS vede (je velké záporné napětí na jeho hradle vzhledem ke kanálu P). Oba tranzistory se tak střídají v otevírání kanálů a v ustáleném stavu je vždy jeden z nich nevodivý. Proto z napájecího zdroje teče jen **velmi nepatrný proud** (rádu nA).

Obdobný princip střídání vodivosti horní větve a dolní větve je využit i u složitějších obvodů. Obrázek uprostřed ukazuje člen NOR. Paralelní spojení tranzistorů NMOS v dolní věti realizuje funkci NOR – při stavu H na kterémkoliv vstupu nebo vstupech skupina tranzistorů vede proud alespoň jedním tranzistorem a výstup je tak spojen se zemí, tedy ve stavu L. Naopak sériové spojení tranzistorů PMOS v horní věti se chová opačně – při stavu H na kterémkoliv vstupu nebo vstupech je alespoň jeden tranzistor zavřen (je to PMOS) a celá skupina nevede proud z napájecího zdroje. Při všech vstupech ve stavu L naopak nevede dolní větev a horní vede, tedy na výstupu je stav H. V ustáleném stavu opět nikdy neneče proud z napájecího zdroje (s výhradou nepatrých zbytkových proudů).

Složitější obvody lze sestavit s využitím skutečnosti, že zapojení horní větve je vždy **duální k zapojení spodní větve**, tj. součet je zaměněn za součin a opačně. Na obrázku vpravo je obvod realizující funkci $Y = (A + B) \cdot \bar{C}$. To odpovídá spodní věti, kde součet $A + B$ je realizován skupinou dvou paralelně zapojených tranzistorů a k tomu součin s proměnnou C jako tranzistor do série s touto skupinou. Horní větev je zapojena přesně opačně, tj. jako $(A \cdot B) + C$. Tím je zaručeno vzájemné střídání vodivosti obou větví.

Zpracování logických proměnných je záležitostí středního stupně součástky. První stupeň CMOS vždy obsahuje **omezovače přepětí** na signálových vstupech podle obr. 4.10.



Obr. 4.10 Omezovač přepětí na vstupu obvodů CMOS

Vstupní obvod chrání další vnitřní obvody především před následky **statického náboje** přivedeného na vstup. Vzhledem k velmi vysokému vstupnímu odporu tranzistorů MOS (rádově $10^{12} \Omega$) může i velmi malý náboj způsobit průraz oxidu pod hradlem a tím zničení obvodu. Záporné náboje jsou svedeny diodou do rozvodu země integrovaného obvodu, kladné do rozvodu napájecího napětí integrovaného obvodu a odtud do napájecího zdroje. Tato cesta ale existuje jen u obvodu zamontovaného do systému – u nezapojené součástky je tudíž daleko větší nebezpečí poškození.

Výstupní stupeň součástky CMOS je v podstatě invertorem, ovšem s tranzistory s malým odporem kanálu, schopnými dodat velký výstupní proud.

Obdobně jako u technologie TTL, i u technologie CMOS je během přechodného děje při změně stavu krátkodobě vodivá horní i dolní větev. Důsledkem je **proudová špička** z napájecího zdroje. Střední hodnota takto odebraného proudu je samozřejmě přímo úměrná kmitočtu přepínání. Je ale závislá i na strmosti vstupních impulzů – při pomalém průchodu přes referenční úroveň trvá proudová špička déle. Dalším důvodem pro zvyšování odběru s kmitočtem jsou **parazitní kapacity**, existující uvnitř integrovaného obvodu i na jeho výstupech. Při změně stavu z L na H se kondenzátor o kapacitě C_p nabije napěťovým skokem ΔU . Následně se při změně z H na L stejným skokem vybije. Tím se z napájecího zdroje odebral náboj Q_p . Stejný náboj by odpovídal konstantnímu proudu I_{CC} tekoucímu po dobu Δt . Lze tak psát

$$Q_p = C_p \cdot \Delta U = I_{CC} \cdot \Delta t, \text{ a tedy}$$

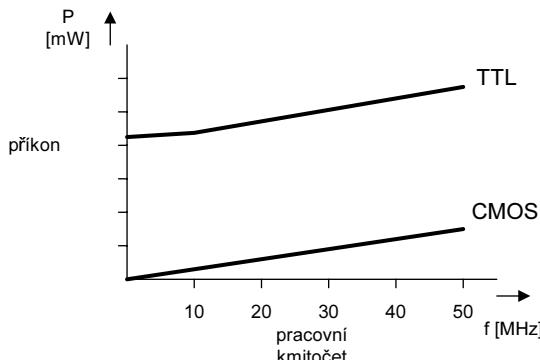
$$I_{CC} = C_p \cdot \frac{\Delta U}{\Delta t} = C_p \cdot \Delta U \cdot f,$$

kde f je kmitočet přepínání mezi stavami L a H, neboli **pracovní kmitočet**. Pro výpočet příkonu z napájecího zdroje je nutné znát velikost napěťového skoku ΔU . Předpokládejme, že napěťové úbytky na otevřených tranzistorech jsou zanedbatelné, a tudíž $\Delta U = U_{CC}$. Pak příkon vlivem parazitních kapacit je

$$P_p = U_{CC} \cdot I_{CC} = C_p \cdot U_{CC}^2 \cdot f$$

Důsledkem je přibližně **lineárně rostoucí příkon** členů CMOS s pracovním kmitočtem. To je i důvodem, proč se příkon CMOS členů často uvádí v $\mu\text{W}/\text{kHz}$. Jindy se uvádí příkon ve statickém stavu (tj. s nulovým pracovním kmitočtem) a příkon při definovaném pracovním kmitočtu. Příkon ve statickém stavu je u obvodů CMOS velmi nízký, nesrovnatelný

s příkonem obvodů TTL. Z výše uvedeného vztahu je rovněž zřejmý význam **snižování napájecího napětí**, které zde vystupuje dokonce v kvadrátu. Rovněž snižování parazitních kapacit je podstatné. Zvyšování hustoty integrace a s tím související zmenšování rozměrů tranzistorů má za následek snižování kapacit i napájecího napětí. Porovnání příkonu obvodů TTL a CMOS jako funkce kmitočtu je na obr. 4.11. Stupnice příkonu není cejchována – záleží na složitosti obvodu. Je třeba vždy srovnávat obvody o stejně funkci.



Obr. 4.11 Závislost příkonu číslicových členů na pracovním kmitočtu

Vývoj obvodů CMOS šel v podstatě dvěma směry. Jedním směrem byl vývoj obvodů, u kterých byl kladen hlavní důraz na rychlosť. Toho se dosahuje **zmenšováním rozměru** tranzistorů, zkracováním délky spojů, apod. Současně se zmenšováním rozměrů bylo nutné **snižovat i napájecí napětí**, aby se zamezilo průrazům. Zmenšení rozměrů znamená menší parazitní kapacity a menší zpoždění ve spojích, snížené napájecí napětí znamená menší napěťové skoky – to vše přispívá ke zrychlení obvodů. Obojí má příznivý vliv i na snížení příkonu. V rámci stavebnice logických členů malé a střední integrace tak vznikly obvody vývodové (a mnohdy i napěťové) ekvivalentní s obvody TTL, ovšem proudově mnohem úspornější. Prvotní řada 74C... byla během 80. let nahrazena řadou **74 HC...** Z důvodu vzájemné slučitelnosti – kompatibility – s řadou TTL vznikly současně obvody **74 HCT...** Písmeno T signalizuje slučitelnost s obvody TTL, tj. shodné napěťové úrovni, samozřejmě při napájecím napětí rovněž shodném s TTL, tedy 5 V. Řady bez písmena T mají jiné (CMOS) napěťové úrovni a jejich slučitelnost s TTL není zaručena. S pokroky v technologii byly později zavedeny řady **AHC/AHCT** s důrazem na minimální spotřebu a **AC/ACT** s důrazem na rychlosť. Řady AC a AHC mohou pracovat i při sníženém napájecím napětí 3,3 V, kdy mají menší příkon, ale také horší statické i dynamické vlastnosti. Zmenšováním rozměrů při dokonalejší technologii se došlo k řadám na snížené napájecí napětí. Vznikly tak řady s napájením 3,3 V, 2,5 V a nižším – některé od 1,2 V. Některé z nich umožňují při napájení 3,3 V spolupráci se součástkami TTL. Typické jsou řady **74 LV, LVC, ALVC**.

Při porovnávání vlastností je třeba rozlišit vlastnosti obvodů vyvinutých na vyšší napájecí napětí, ale napájených sníženým napětím, s vlastnostmi obvodů vyvinutých zásadně pro nižší napětí. V prvém případě znamená snížení napětí zhoršení dynamických parametrů i výstupních proudů, ale současně též snížení příkonu. Případ od případu tak lze volit kompromis. Ve druhém případě se však jedná o všeobecně zlepšení vlastností obvodů.

Doménou obvodů CMOS jsou ovšem obvody nejvyšší integrace, jako jsou paměti, mikroprocesory a jednočipové mikropočítače. I zde se uplatňuje trend zmenšování rozměrů při současném snižování napájecího napětí.

Druhým směrem vývoje šly řady obvodů malé a střední integrace pro použití v náročných podmínkách **zvýšeného rušení**. Typické jsou **řady 4000 a 4500**, dovolující napájecí napětí od 3 do 15 V. Při vyšším napájecím napětí má obvod větší rezervy mezi vstupními a výstupními úrovněmi L a H a tím i větší odolnost proti rušení. Obvody jsou **pomalé** (t_{pd} kolem 100 ns), což je opět příznivé pro potlačení rušení. Celá řada obsahuje velmi zajímavé a užitečné kombinace obvodů malé a střední integrace. **Nejsou** funkčně ani vývodově ekvivalentní s řadami 74...

Vlastnosti nejčastějších typů obvodů CMOS jsou uvedeny v tab. 4.3. Příkon je přepočítán na jeden elementární člen (NAND, NOR). Rovněž zpoždění jsou udávána pro elementární členy.

Tab. 4.3 Charakteristické vlastnosti vybraných řad obvodů CMOS

Typ	Příkon na 1 člen [$\mu\text{W}\cdot\text{kHz}^{-1}$]	Zpoždění t_{pd} [ns]	Výstupní proud [mA]
74 HC, $U_{CC} = 5 \text{ V}$	0,5	10	-8/8
74 AC, $U_{CC} = 5 \text{ V}$	0,8	3	-12/12
4000 , $U_{CC} = 15 \text{ V}$	0,3	100	-5/13
74 LVC, $U_{CC} = 3,3 \text{ V}$	0,2	4	-24/24

Obdobně jako u TTL členů, i zde jsou maximální výstupní proudy uváděny pro členy s největším výstupním proudem, využívané pro připojení na sběrnice (budiče sběrnice). Běžné členy řady mají výstupní proudy několikanásobně nižší. Na rozdíl od TTL jsou maximální **proud shodné pro stav L i H** (s výjimkou netypické řady 4000). Obvody HC a AC mohou pracovat i při napájecím napětí 3,3 V, ale s horšími vlastnostmi.

Napěťové úrovně obvodů CMOS mohou a nemusí být kompatibilní (= slučitelné) s úrovněmi TTL. Kompatibilita je zaručena u obvodů s napájením 5 V a označením „T“, např. HCT, ACT, AHCT, atd. I některé řady s napájecím napětím 3,3 jsou TTL kompatibilní – je to např. LV, LVC, ALVC, atd. Při vzájemném propojování součástek z různých řad je vždy nutné prostudovat firemní dokumentaci k jednotlivým obvodům. Pokud si napěťové úrovně neodpovídají, lze pro jejich převod použít speciální součástky – převodníky úrovní. Existují však jen pro některé kombinace řad. U programovatelných logických obvodů (viz kapitolu 13) lze naprogramovat, s jakými vstupními či výstupními napěťovými úrovněmi bude obvod pracovat. To řeší otázku propojení obvodů s různým napájením bez nutnosti užití převodníků úrovní.

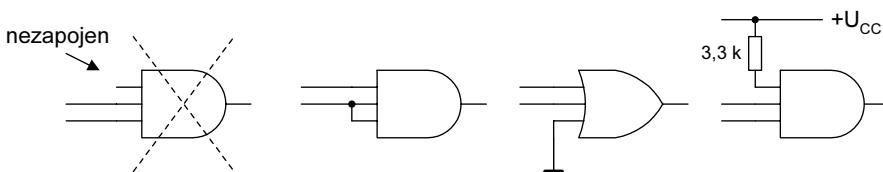
4.5 Technologie BiCMOS

Tato technologie kombinuje výhodné vlastnosti technologie CMOS i bipolární. Bipolární tranzistory s jejich schopností pracovat s většími proudy jsou využity ve výstupních obvodech, naopak tranzistory NMOS a PMOS jsou využity na všech ostatních místech a tak jsou

obvody BiCMOS podstatně úspornější, než obvody TTL. Jsou určeny pro připojení na sběrnice s velkou kapacitní zátěží. Mezi nejrychlejší patří řada **ALB** (3,3 V, 25 mA, 2 ns), velké výstupní proudy zvládá řada **LVT** (3,3 V, 64 mA, 4,5 ns) i řada **ABT** (5 V, 64 mA, 3,5 ns, TTL kompatibilní).

4.6 Nevyužité vstupy číslicových součástek

Nevyužité vstupy jsou časté. Technologická řada logických členů obsahuje vždy jen omezený počet členů, takže při konkrétním návrhu se využívají součástky, pokrývající potřebné funkce, ale často nadbytečně. U **technologie TTL** se nezapojený vstup chová, jako by v obvodu vůbec nebyl – neprotéká jím proud. Obvod je pak ale náchylný na rušení, na nezapojený vstup mohou kapacitními vazbami pronikat signály od okolních obvodů. Proto se nezapojené vstupy nedoporučují. Druhá možnost je spojit nevyužitý vstup s vedlejším využitým vstupem. Při pohledu na náhradní schéma vstupního obvodu TTL je zřejmé, že tímto spojením do dvojice se nezmění vstupní proud dvojice ve stavu L, ale zdvojnásobí se vstupní proud ve stavu H – ten je ovšem i tak zanedbatelný. Je však třeba počítat se vztuřstem vstupní kapacity. Třetí možností je spojení nevyužitého vstupu se zemí – to je vhodné u členů OR a NOR. Poslední možností, vhodnou např. pro členy AND a NAND, je spojení nevyužitého vstupu přes rezistor s napájecím napětím. Tím je vstupní přechod závěrně polarizován a vstup je vyřazen. Rezistor o hodnotě několika $k\Omega$ brání případnému průrazu vstupu při přepětích na napájecím vedení. *Obr. 4.12* ukazuje všechny možnosti.



Obr. 4.12 Ošetření nevyužitých vstupů

U obvodů **CMOS a BiCMOS** je ponechání nezapojeného vstupu zcela vyloučené. Vstupní impedance je u těchto obvodů velmi vysoká a nezapojený vstup je tak podstatně více rušen vazbami nejen na vnější, ale i na vnitřní obvody. Dále, pokud se napětí na nezapojeném vstupu přiblíží referenční úrovni, zvýší se podstatně **napájecí proud**. Ostatní opatření dle *obr. 4.12* jsou však vhodná i pro tyto technologie. Vzhledem k nulovému vstupnímu proudu je rezistor při připojení na napájecí napětí zbytečný.

Na obrázcích v dalším textu bude trvalé uvedení kteréhokoliv vstupu do stavu H znázorňeno jako přivedení napětí U_H bez ohledu na způsob jeho získání (přímo z $+U_{CC}$, přes rezistor, z pomocného zdroje, atd.), a uvedení vstupu do stavu L jako uzemnění vstupu.

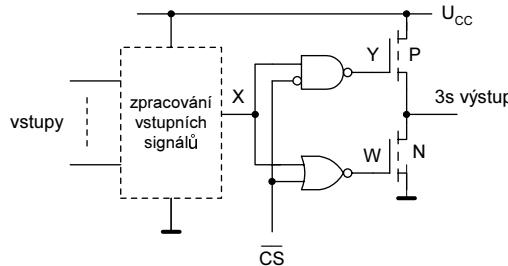
426410

5

TŘÍSTAVOVÉ VÝSTUPY A OTEVŘENÉ KOLEKTORY

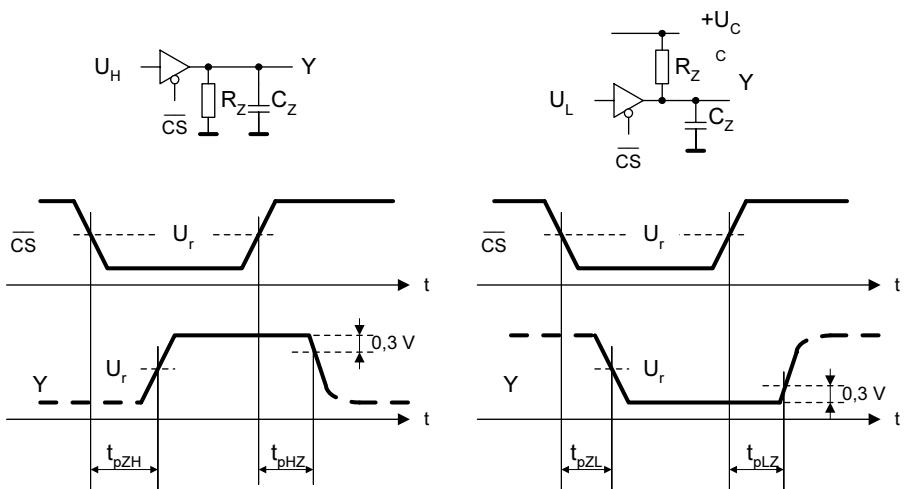
5.1 Obvody s třístavovými výstupy

Třístavový výstupní obvod umožňuje vypnutí obou tranzistorů. Výstup je pak ve **vysokoimpedančním stavu** (značí se „Z“) a projevuje se jen jako **kondenzátor** o hodnotě několika pF. Proudové zavřených tranzistorů lze zanedbat. Signál, generovaný takovýmto obvodem, tedy může nabývat tří hodnot – „0“, „1“, „Z“. Princip řízení koncového stupně u technologie CMOS ukazuje obr. 5.1.



Obr. 5.1 Výstupní obvod třístavového členu

Vstup **výběrového signálu \overline{CS}** (angl. *Chip Select*) řídí zablokování nebo odblokování výstupu. Zřejmě při $\overline{CS} = 1$ je $W=0$ a $Y=1$, tedy oba tranzistory vypnuty. Naopak při $\overline{CS} = 0$ je $W=Y=X$, tedy na výstupu bude signál X (samotný koncový stupeň s tranzistory NMOS a PMOS neguje). Z hlediska dynamických parametrů se u třístavových členů rozlišují dvě cesty signálů – jednak cesta zpracovávaných signálů ze vstupu na výstup při odblokovaném výstupu ($\overline{CS} = 0$), jednak cesta ovládání vysokoimpedančního stavu ze vstupu \overline{CS} na výstup. V prvém případě se dynamické parametry neliší od parametrů běžných (tj. dvojstavových členů), tedy se bude uvádět t_{pLH} , t_{pHL} . Ve druhém případě se jedná o **doby odblokování** (t_{pZH} nebo t_{pZL}) a **doby zablokování** (t_{pHZ} a t_{pLZ}). Jejich definici a způsob měření ukazuje obr. 5.2.



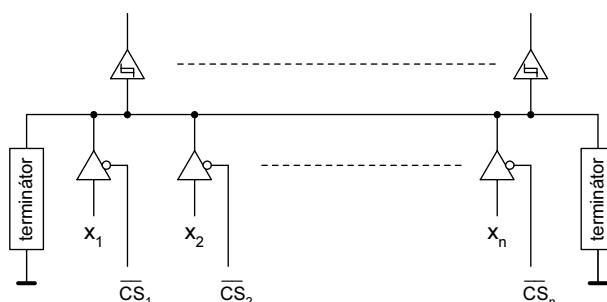
Obr. 5.2 Dynamické parametry třístavového členu

Hodnotu zatěžovacího rezistoru a kondenzátoru, které jsou připojeny k výstupu při měření dynamických parametrů, určují technické podmínky výrobce (např. $R_Z = 1 \text{ k}\Omega$, $C_Z = 50 \text{ pF}$). Vysokoimpedanční stav je v časovém diagramu vyznačen čárkovanou čarou. Bez zatěžovacího rezistoru by napětí v tomto stavu nebylo definováno. Doby blokování a odblokování se pohybují v rozsahu několika ns.

V obrázku 5.2 je vyznačen nejjednodušší třístavový člen – **budič sběrnice** (angl. *Bus Driver*). Signál v něm není nijak logicky zpracován, jediným účelem je řídit odpojení a připojení výstupu. Vedle nenegujících budičů existují i budiče negující. Budiče sběrnice jsou vždy těmi nejvýkonnějšími obvody dané technologické řady.

Třístavové výstupní obvody umožňují spojovat výstupy číslicových součástek **paralelně**. Z elektrického hlediska se jedná o uzel, z konstrukčního hlediska se zpravidla jedná o sběrnici. Sběrnice představuje vodič nebo skupinu vodičů stejněho charakteru (např. adresová nebo datová sběrnice), procházející celým systémem, na kterou se napojují jednotlivé obvody. U třístavových výstupů platí přísné pravidlo, že v jednom okamžiku může být **nejvíše jeden** z nich připojen prostřednictvím výběrového signálu \overline{CS} . U všech ostatních musí být výstup blokován. Pokud by byly odblokovány dva výstupní obvody a byly by přitom v opačných stavech, tekl by mezi nimi nepřijatelně velký proud (u výkonových budičů sběrnice běžně přes 100 mA), který by velmi rychle vyvolal místní přehřátí v integrovaném obvodu a tím jeho poškození. Doba současného připojení dvou nebo více obvodů bez rizika poškození je velmi krátká a pohybuje se v řádu mikrosekund. Význam připojení signálů na sběrnice přes třístavové obvody je ten, že výběrovými signály \overline{CS} lze **volit zdroj signálu** x_1 , x_2 , ..., x_n , který bude přiveden na sběrnici. Na ní je pak k dispozici pro obvody, které vibraný signál přijímají.

Sběrnice není zdaleka ideálním spojem a projevuje se na ní **rušení** od jiných obvodů v systému i efekty, způsobené jejím charakterem **dlouhého vedení**. Proto přijímací členy mají **hysterezní** převodní charakteristiku, která způsobuje jejich sníženou citlivost na rušení, překmity a pomalé přechody přes referenční úroveň. Vzhledem k rychlosti moderních obvodů je třeba počítat s efekty dlouhého vedení již u sběrnice o délce kolem 10 cm. Pro zamezení odrazů je třeba vedení **zakončit** vhodným rezistorem, ideálně o velikosti charakteristické impedance vedení (Z_0). Vedení, realizované jako úsek plošného spoje, má typicky Z_0 blízkou hodnotě 100 Ω . K zakončení se používají jednak prosté rezistory, jednak kombinace rezistorů a kondenzátorů, souhrnně označované jako „**terminátory**“. U sběrnice nelze stanovit „začátek“ a „konec“ vedení – budiče a přijímače mohou být napojeny kdekoli. Proto je



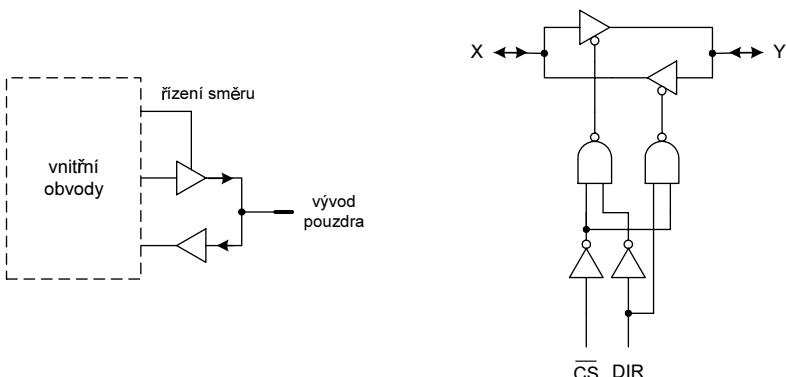
Obr. 5.3 Sběrnice s třístavovými členy

nutné použít terminátory na **obou koncích**. Každý budič je tak zatížen paralelní kombinací dvou terminátorů, což vyžaduje jeho zvýšený výkon – viz obr. 5.3.

Efekty dlouhého vedení a zatížení budičů jsou zvláště patrné u rozsáhlých decentralizovaných systémů, ve kterých jsou data přenášena sériově po společné sběrnici na vzdálenost jednotek až stovek metrů. Rychlé sériové sběrnice – tzv. „průmyslové sběrnice“ – s třístavovými budiči podle normy RS 485 jsou velmi rozšířené.

Třístavové výstupy mohou být integrovány ve složitějších obvodech střední a velké integrace. Velmi často mají obvody dvojsměrné vývody – mohou mít funkci vstupů nebo výstupů. To se týká zvláště pamětí, mikroprocesorů a dalších obvodů mikropočítačů, pracujících s daty. Dále pak programovatelných logických obvodů PLD a FPGA. Princip ukazuje obr. 5.4 vlevo. Na straně vývodu pouzdra jsou signály dvojsměrné, uvnitř jsou rozděleny. U obvodů velké integrace však nejsou výstupní obvody konstruovány na velké proudy (jen na několik mA). V případě jejich velké zátěže rozsáhlou sběrnici s velkými parazitními kapacitami (desítky a stovky pF) by tak došlo k výraznému zpomalení systému. Proto se obvody velké integrace často navíc oddělují samostatnými budiči sběrnice s velkými výstupními proudy. U některých obvodů nejsou třístavové výstupy přítomny vůbec, takže pro jejich připojení na sběrnici je třeba rovněž použít samostatné budiče sběrnice.

Pro připojení takovýchto obvodů na sběrnici slouží **dvojsměrné budiče sběrnice**. Jejich zapojení ukazuje obr. 5.4 vpravo. Signálem DIR se řídí směr přenosu, signálem CS pak vyvolání vysokoimpedančního stavu. Při $DIR=0$ je směr přenosu $x \rightarrow y$, při $DIR=1$ je $y \rightarrow x$. Při $CS=1$ jsou ve vysokoimpedančním stavu oba výstupy.

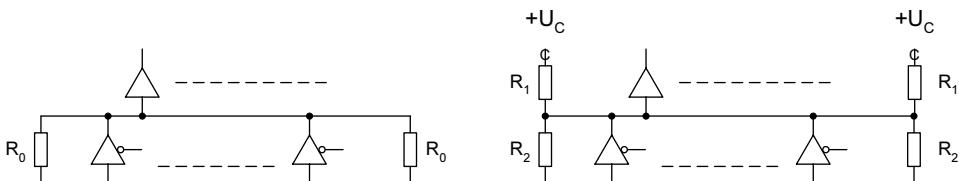


Obr. 5.4 Dvojsměrné budiče sběrnice

5.2 Terminátory sběrnice

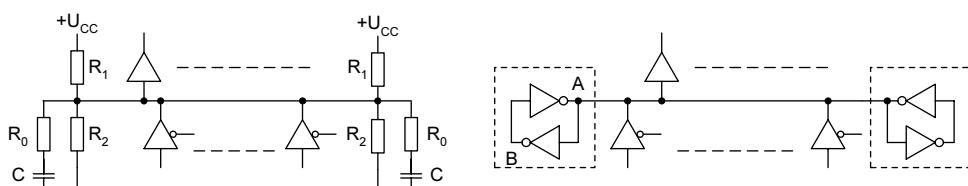
Na obr. 5.5 vlevo jsou znázorněny nejjednodušší terminátory. Pro správné impedanční zakončení sběrnice musí platit $R_0 = Z_0$. Výstupní proud budičů je přitom největší ve stavu H a nulový ve stavu L. To není nevhodnější situace pro bipolární technologii, ve které větší proud členy pohltí ve stavu L a naopak menší proud generují ve stavu H. Terminátory na obrázku vpravo jsou pro tyto účely vhodnější. Paralelní kombinace R_1 s R_2 je rovna R_0 a poměr obou rezistorů je takový, aby při odpojených všech budičích bylo na sběrnici napětí

ležící v pásmu U_H . Tak např. pro $Z_0 = 100 \Omega$ je vhodná kombinace $R_1 = 180 \Omega$ a $R_2 = 220 \Omega$ (hodnoty řady E12). Pak napětí na sběrnici při odpojených budičích a $U_{CC} = 5$ V bude 2,75 V. Při provozu je budič ve stavu L zatížen proudem dodávaným terminátory a budič ve stavu H naopak zatížen není. V obou případech podle obr. 5.5 je tedy napětí při odpojených budičích definováno. Nedefinované napětí nelze dlouhodobě připustit, neboť jeho případná hodnota blízká referenčnímu napětí by způsobila zvýšený příkon členů připojených na sběrnici svými vstupy.



Obr. 5.5 Terminátory s rezistory

Nevýhodou terminátoru podle obr. 5.5 vpravo je velká spotřeba proudu ze zdroje U_{CC} . Ta může být podstatná u datových a adresových sběrnic s velkým počtem signálů, zvláště u systémů s bateriovým napájením. Pak lze použít řešení podle obr. 5.6. Vlevo jsou terminátory s kombinací RC. Rezistor R_0 je dolním koncem uzemněn pro střídavé signály. Kondenzátor brání trvalému průtoku proudu přes rezistor a budiče tak nemusí dlouhodobě dodávat proud ani ve stavu H, ani ve stavu L. Časová konstanta R_0C se volí jako nejméně čtyřnásobek doby zpoždění signálu ve vedení. Rezistory R_1 a R_2 mají v tomto případě velkou hodnotu, tak aby nezatěžovaly zdroj napětí U_{CC} . Jejich účelem není impedanční zakončení (to obstarává R_0), ale zajištění vhodného napětí na sběrnici při odpojených všech budičích. Mohou být vynechány, pokud nenastává situace, kdy všechny budiče jsou odpojeny.



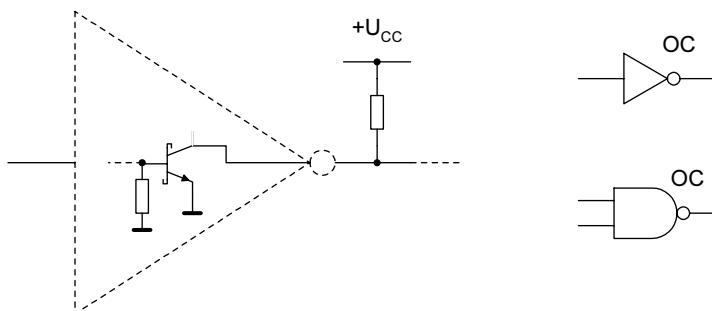
Obr. 5.6 Terminátory se sníženým příkonem

Na obr. 5.6 vpravo je ukázán **aktivní terminátor** (angl. *bus hold*). Jedná se o nejjednodušší verzi bistabilního klopného obvodu, ve kterém působí kladná zpětná vazba. Je-li $A=1$, je $B=0$, a proto $A=1$, což je potvrzením původně předpokládaného stavu. Stejně je tomu při $A=0$ a $B=1$. K překlopení v bodu A z 1 do 0 je třeba krátkodobě vnitit do bodu A stav 0, což vyžaduje odvedení proudu některým z budičů sběrnice. Pak $B=1$ a tudíž $A=0$, a proud již dále neteče. Klopný obvod se překlopil a dále v tomto stavu setrvává. Obdobně je tomu při překlopení A z 0 do 1, kdy je zapotřebí dodat proud do bodu A . Proudový impulz, potřebný pro překlopení terminátoru, odčerpá energii přicházející vlně, takže již nedojde k odrazu. V ustáleném stavu je aktivní terminátor (CMOS) velmi úsporný. Zvláštní vlastností aktivních terminátorů je to, že při odpojení všech budičů zůstává na sběrnici poslední stav před odpojením.

Jistý, i když ne stoprocentní účinek mají omezovače přepětí integrované na vstupech číslicových obvodů. Zabraňují překmitu napětí nad $+U_{CC}$ a pod 0 a tím potlačují odrazy.

5.3 Výstupy s otevřenými kolektory

U tohoto výstupního obvodu existuje jen tranzistor, který uzemňuje výstup a zajišťuje tak stav L – viz obr. 5.7. Na místě bipolárního tranzistoru může být též MOSFET.



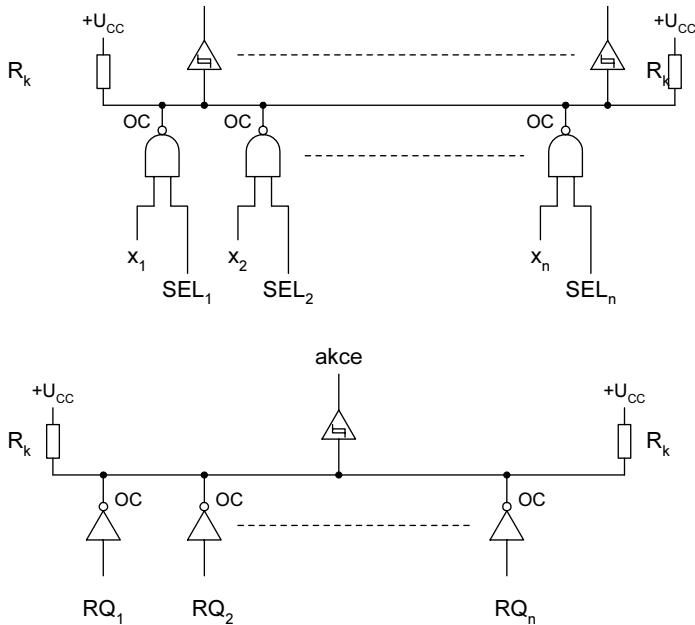
Obr. 5.7 Výstup s otevřeným kolektorem

Stav H obvod nedokáže sám vytvořit a musí být zajištěn vnějším rezistorem. Logické členy s otevřenými kolektory existují jednak jako prosté invertory, jednak jako členy realizující jednoduché logické funkce (NAND). Jejich použití vyplývá z možnosti jejich **paralelního spojování**. Situace je ale jiná, než u třístavových výstupů. U otevřených kolektorů není na závadu, pokud více než jeden z paralelně spojených členů má na výstupu stav L. Proud je totiž dán velikostí jediného společného kolektorového rezistoru, a to bez ohledu na počet otevřených tranzistorů. Stav H celé skupiny nastane, jsou-li všechny tranzistory vypnuty. Ani zde nehrozí kolize.

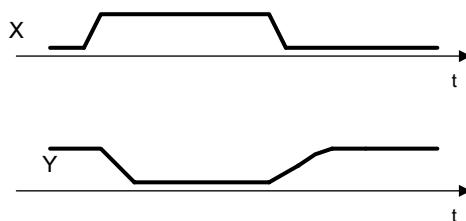
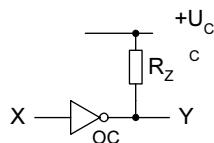
Členy s otevřenými kolektory mají široké využití pro připojování na **společnou sběrnici**. Obr. 5.8 ukazuje připojení členů NAND (nahoře) a invertorů (dole). O zakončení sběrnice platí stejně zásady jako v případě třístavových členů. Rezistory R_k zde plní funkci terminátorů a jsou proto na obou koncích sběrnice.

Obě uspořádání mají svoje pole aplikací. Členy NAND na horním obrázku lze vstupy SEL blokovat (při $SEL=0$ je vypnut výstupní tranzistor) nebo připojit. Vybraný signál x se tak dostane na sběrnici (podle obrázku v negaci). Je-li vybrán více než jeden člen, nedojde k ohrožení těchto členů, nýbrž jen ke zkreslení signálů na sběrnici. Tato situace se nazývá „**nedestruktivní konflikt**“. Členy s otevřenými kolektory existují i ve výkonové verzi pro připojení na rychlé sériové sběrnice, jako je např. CANbus, s arbitráží založenou právě na nedestruktivních konfliktech.

Druhá, zcela odlišná aplikace je ukázána na dolní části obr. 5.8. Invertory připojené na sběrnici nemají žádný vstup pro řízení a proto jejich jedinou úlohou je vnitrit stav L na sběrnici při kterémkoliv aktivním (tj. ve stavu H) signálu RQ , případně i při větším počtu takto aktivních signálů. Celá soustava nemá význam přenosu signálů, nýbrž **žádosti o vyvolání**



Obr. 5.8 Sběrnice s obvody s otevřenými kolektory



Obr. 5.9 Zpomalení náběhu impulzu u obvodů s otevřeným kolektorem

nejaké akce. Tak je tomu např. v počítačích, kde jsou takto vyvolávána přerušení programu či vynulování („reset“). Dalším příkladem jsou jednotlivé diagnostické obvody, kontrolující jednotlivé části systému, které v případě poruchy vyvolají akci pro zamezení nebezpečného stavu.

Sběrnice, na které jsou připojené obvody s otevřenými kolektory, přechází ze stavu H do stavu L stejně rychle, jako v případě třístavových členů – v obou případech je v cestě ze

společné sběrnice do země otevřený tranzistor, přes který se rychle vybijí parazitní kapacity. Přechod z L do H je ale zajištěn jen vnějším rezistorem, jehož hodnota nemůže být příliš nízká. Kdyby tomu tak bylo, pak by proud přiváděný z rozvodu napájecího napětí do kolektoru sepnutého tranzistoru příliš zvyšoval napětí U_{OL} . Proto přechod ze stavu L do H je **vždy pomalejší** – viz obr. 5.9.



KOMBINAČNÍ OBVODY

Pro kombinační obvody je charakteristické to, že na jistou kombinaci hodnot vstupních signálů reagují vždy stejnou kombinací hodnot výstupních signálů, a to bez ohledu na minulé hodnoty signálů. Kombinace hodnot signálů se nazývá stav.

Činnost kombinačních obvodů lze popsat logickými funkciemi. **V ustáleném stavu** může logická funkce nabývat jen hodnot 0 a 1. Jako proměnné v ní mohou vystupovat pouze logické proměnné, které též mohou nabývat jen hodnot 0 nebo 1.

Logické funkce se vyjadřují (kromě jiného) prostřednictvím logických výrazů. **Logický výraz** je tvořen logickými **proměnnými** a **operátory**. Soustava operátorů musí být volena tak, aby umožnila vyjádřit libovolnou logickou funkci. Existuje několik takovýchto soustav, nejčastěji se ale používá soustava operátorů Booleovy algebry, tj. logický součet, logický součin a negace.

Jelikož logické proměnné nabývají jen dvou možných hodnot, nemohou v logických funkciích existovat žádné mezistavy a přechodné děje, na rozdíl od reálných signálů. Popis systému logickými funkciemi je tedy platný jen **v ustálených stavech** – pro přechodné děje jej nelze použít. Dalším nutným předpokladem použitelnosti logických funkcí jsou správné napěťové úrovně v obvodu.

6.1 Základní pravidla Booleovy algebry

Vliv konstant 0 a 1:

$$\begin{aligned}0 + a &= a + 0 = a \\1 + a &= a + 1 = 1 \\1 \cdot a &= a \cdot 1 = a \\0 \cdot a &= a \cdot 0 = 0 \\a + a &= a \\a \cdot a &= a\end{aligned}$$

Vlastnosti negace:

$$\begin{aligned}\bar{0} &= 1 \\ \bar{1} &= 0 \\ a + \bar{a} &= 1 \\ a \cdot \bar{a} &= 0 \\ &= \\ a &= a\end{aligned}$$

Označování negace v literatuře není jednotné. Nejlépe čitelný je pruh nahoře, ovšem komplikuje psaní textu – musí se vkládat pomocí editoru rovnice. Proto se často vidí zápis negace jako $/a$ nebo a' . Obojí usnadňuje psaní textu běžným editorem, ale komplikuje naopak zápis složitějších výrazů, neboť vyžaduje použití závorek. Např. dobré čitelné $a + b$ by se muselo zapsat jako $/(a+b)$ nebo jako $(a+b)'$, což zřetelně zhoršuje čitelnost.

Konstanty „0“ a „1“ nemají význam čísel 0 a 1 typu **integer** a nemohou se s nimi volně zaměňovat.

Operátor „+“ označuje logický součet, neboli funkci OR. Operátor „·“ označuje logický součin, neboli funkci AND. V psaném textu lze operátor logického součinu „·“ případně vynechat, podobně jako v aritmetice, takže lze psát ab namísto $a \cdot b$. Symbol „~“ označuje negaci, neboli funkci NOT, čte se jako „non“.

6.2 Hlavní pravidla pro tvorbu a úpravy logických výrazů

Úpravy výrazů sledují dva cíle. Buď se jedná o zjednodušení ve smyslu snížení počtu písmen ve výsledném výrazu, nebo se jedná o úpravu do takového tvaru, který vyhovuje číslicovému obvodu (součástce) obvodu, který je již k dispozici.

Obdobně jako v běžné algebře, i zde platí komutativní, asociativní a distributivní zákon. Používání závorek je stejné jako v běžné algebře. V dále uvedených vztazích mohou symboly a, b, \dots označovat jak jednotlivé proměnné, tak i celé výrazy. Platí následující zákony:

Komutativní:

$$a + b = b + a,$$

$$a \cdot b = b \cdot a$$

Asociativní:

$$a + b + c = a + (b + c),$$

$$a \cdot b \cdot c = a \cdot (b \cdot c)$$

Distributivní:

$$a \cdot (b + c) = a \cdot b + a \cdot c,$$

$$a + b \cdot c = (a + b) \cdot (a + c)$$

Princip duality:

Každá identita platí i při vzájemné záměně operátorů „+“ a „·“, a též konstant „0“ a „1“. Zřetelně je to vidět na všech dalších pravidlech.

Pohlcení (absorpce):

$$a + a \cdot b = a$$

Neboť $a + a \cdot b = a \cdot (1 + b) = a \cdot 1 = a$

$$a \cdot (a + b) = a$$

Neboť $a \cdot (a + b) = a \cdot a + a \cdot b = a + a \cdot b = a$

Spojování:

$$a \cdot \bar{b} + a \cdot b = a$$

Neboť $a \cdot \bar{b} + a \cdot b = a \cdot (\bar{b} + b) = a \cdot 1 = a$

$$(a + \bar{b}) \cdot (a + b) = a$$

Neboť ... aplikuj princip duality.

Zjednodušení:

$$a \cdot (\bar{a} + b) = a \cdot b$$

Nebot' $a \cdot (\bar{a} + b) = a \cdot \bar{a} + a \cdot b = 0 + a \cdot b = a \cdot b$

$$a + \bar{a} \cdot b = a + b$$

Nebot' ... aplikuj princip duality.

Konsensus:

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

Nebot' $a \cdot b + \bar{a} \cdot c + b \cdot c = a \cdot b + \bar{a} \cdot c + b \cdot c \cdot (a + \bar{a}) = a \cdot b + \bar{a} \cdot c + a \cdot b \cdot c + \bar{a} \cdot b \cdot c = a \cdot b \cdot (1+c) + \bar{a} \cdot c \cdot (1+b) = a \cdot b \cdot 1 + \bar{a} \cdot c \cdot 1 = a \cdot b + \bar{a} \cdot c$

$$(a+b) \cdot (\bar{a}+c) \cdot (b+c) = (a+b) \cdot (\bar{a}+c)$$

Nebot' ... aplikuj princip duality.

DeMorganovy zákony:

$$\begin{aligned}\overline{a+b} &= \bar{a} \cdot \bar{b} \\ \overline{a \cdot b} &= \bar{a} + \bar{b}\end{aligned}$$

6.3 Pravdivostní tabulka

Jeden z možných popisů činnosti kombinačního obvodu je pravdivostní tabulka. Pravdivostní tabulka přiřazuje všem možným kombinacím hodnot na vstupech obvodu hodnoty na výstupu (výstupech) obvodu. Její příklad pro funkci tří proměnných $f(x,y,z)$ a její negaci $\bar{f}(x,y,z)$ je uveden v tab. 6.1.

Tab. 6.1 Příklad pravdivostní tabulky

DEK	x	y	z	f(x,y,z)	$\bar{f}(x,y,z)$
0	0	0	0	1	0
1	0	0	1	1	0
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	1	0
7	1	1	1	0	1

Jednotlivé kombinace je vhodné uspořádat v rostoucím smyslu tak, jak roste binární číslo xyz (x je nejvyšší řad, z nejnižší). Řádky pravdivostní tabulky pak můžeme označit odpovídajícími dekadickými čísly – viz sloupec DEK ($DEK = x \cdot 2^2 + y \cdot 2^1 + z \cdot 2^0$). Pak ovšem nemu-

síme kreslit celou tabulku a stačí vyjmenovat ty řádky, kde $f = 1$. Dostáváme tak zhuštěný zápis pravdivostní tabulky. Pro tab. 6.1 by tento zápis měl tvar:

$$f(x, y, z) = \sum m(0, 1, 3, 4, 6)$$

Symbol Σ naznačuje, že funkce f je vyjádřena v **součtovém** (disjunktním) tvaru. Bude se samozřejmě jednat o **logický** součet. Vychází se ze skutečnosti, že libovolnou logickou funkci lze vyjádřit jako součet elementárních logických funkcí m_i , z nichž každá má hodnotu 1 jen na jediném řádku tabulky. Takovéto elementární funkce musí obsahovat všechny proměnné funkce f , ať nenegované nebo negované. Tak např. funkce $m_0 = x \cdot y \cdot z$ má hodnotu 1 jen v řádku DEK=0, funkce $m_1 = x \cdot y \cdot z$ má hodnotu 1 jen v řádku DEK=1, funkce $m_2 = x \cdot y \cdot z$ má hodnotu 1 jen v řádku DEK=2, atd. Takto definované elementární funkce se nazývají **mintermy**. Vybereme jen ty mintermy, které odpovídají řádkům s funkční hodnotou 1. Výsledný součet mintermů se nazývá **úplná součtová normální forma** (či úplná disjunktní normální forma) funkce, zkratkou **ÚDNF**. Každý minterm **ÚDNF pokrývá** právě jeden jednotkový bod funkce (tj. řádek pravdivostní tabulky s funkční hodnotou 1). Vyjádření funkce jako **ÚDNF** je sice ekvivalentní k vyjádření funkce pravdivostní tabulkou, pro implementaci obvodu je však příliš složité. Pro její zjednodušování (minimalizaci) existují dobře popsané systematické postupy.

Stejně dobře, jako vyjmenovat řádky s funkční hodnotou 1, by bylo možné vyjmenovat řádky s funkční hodnotou 0. Pak dostaneme funkci z výše uvedené tabulky 6.1. ve tvaru

$$f(x, y, z) = \prod M(2, 5, 7)$$

Symbol Π naznačuje, že funkce f je vyjádřena v **součinovém** (konjunktním) tvaru. Libovolnou logickou funkci lze vyjádřit jako součin elementárních logických funkcí M_i , z nichž každá má hodnotu 0 jen na jediném řádku tabulky. Takto definované elementární funkce se nazývají **maxtermy**. Obdobně jako minterm, i maxterm obsahuje všechny proměnné, ale v logickém součtu. Tak např. k řádku s DEK=2, kdy f má hodnotu 0, by existoval maxterm $M_2 = x + y + z$. Ten má hodnotu 0 jedině při $x = 0, y = 1, z = 0$. Povšimneme si, že proti mintermu m_2 jsou zde proměnné negovány opačně. Výsledný součin maxtermů se nazývá **úplná součinová normální forma** (či úplná konjunktní normální forma) funkce, zkratkou **ÚKNF**. Každý maxterm **ÚKNF pokrývá** právě jeden nulový bod funkce (tj. řádek pravdivostní tabulky). Stejně jako **ÚDNF**, i **KDNF** je pro implementaci obvodu příliš složitá. I pro její zjednodušování existují systematické postupy.

Pro výše uvedený příklad z tab. 6.1 dostaneme **ÚDNF** jako:

$$f(x, y, z) = \bar{x}\bar{y}z + \bar{x}yz + \bar{xy}z + xy\bar{z} + xyz$$

Stejně lze sestavit **ÚKNF** jako:

$$f(x, y, z) = (x + \bar{y} + z)(\bar{x} + y + z)(\bar{x} + \bar{y} + z)$$

Pokud je pravdivostní tabulka zaplněna jedničkami příliš hustě (je jich více než nul), bude ve výsledné **ÚDNF** velký počet členů. Pak stojí za úvahu řešit funkci jako negaci a tím dostat řidčeji zaplněnou tabulku – pro výše uvedený příklad byl v tab. 6.1 doplněn sloupec $\bar{f}(x, y, z)$:

$$\bar{f}(x, y, z) = \sum m(2, 5, 7) = \bar{x}\bar{y}z + x\bar{y}z + xyz$$

Nenegovaná podoba funkce:

$$f(x, y, z) = \overline{\overline{xyz} + \overline{x}yz + xy\overline{z}}$$

Snadno se lze přesvědčit, že po aplikaci DeMorganových zákonů dostaneme výraz, který se shoduje s výše nalezenou ÚKNF:

$$\overline{\overline{xyz} + \overline{x}yz + xy\overline{z}} = \overline{\overline{xyz} \cdot \overline{x}yz \cdot xy\overline{z}} = (x + \overline{y} + z)(\overline{x} + y + \overline{z})(\overline{x} + \overline{y} + \overline{z})$$

6.4 Neurčené stavy

Ve většině případů technické praxe se stává, že v důsledku zvláštního uspořádání zařízení **nemůže dojít** k některým kombinacím vstupních signálů. Tak např. kontrolní kontakty stykače nemohou ukazovat současně stav „zapnuto“ a „vypnuto“, automatika výtahu nemůže dostat hlášení o průjezdu kabiny více patry současně, apod. Jelikož nemůže k jisté kombinaci signálů vůbec dojít, je funkční hodnota bezvýznamná a může být libovolně doplněna na 1 či 0. V pravdivostní tabulce pak tento neurčený stav patřičně vyznačíme (např. jako „“) a mluvíme o **funkci s neurčenými stavy**. Neurčené stavy mohou mít i jiný původ. Např. jistá kombinace se vyskytne při nepřípustném stavu soustavy, který je chápán jako havárie a soustava je jinými obvody blokována. Funkční hodnota je pak rovněž bezvýznamná.

Tab. 6.2 ukazuje případ funkce s neurčenými stavy, ve které kombinace a = 1, b = 1 nemůže nastat:

Tab. 6.2 Pravdivostní tabulka funkce s neurčenými stavy

a	b	c	f(a,b,c)
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	-
1	1	1	-

Ve zhuštěném zápisu se dají neurčené stavy do závorky:

$$f(a, b, c) = \sum m(0, 2, 3, 5(6, 7))$$

Do ÚDNF se mintermy odpovídající neurčeným stavům mohou, ale **nemusí** zahrnout. Můžeme je v prvém kroku zahrnout všechny. V některé fázi dalšího zjednodušování funkce se některé mohou ukázat jako užitečné a tedy se ponechají. To znamená, že původní neurčený stav v tabulce byl nakonec doplněn na jedničku. Jiné mintermy se mohou ukázat jako nadbytočná komplikace a pak se mohou zrušit. To znamená, že původní neurčený stav v tabulce byl nakonec doplněn na nulu.

Je třeba si uvědomit, že v **reálném** obvodu žádné neurčité stavy neexistují. Obvod vždy dává nějakou **konkrétní** kombinaci hodnot výstupních signálů – v neurčených stavech však nezáleží na tom, jaké signály vydává.

6.5 Minimalizace logické funkce

Úplnou disjunktní normální formu (ÚDNF), získanou zpravidla z pravdivostní tabulky, lze dále upravovat, a to zcela systematicky. Mintermy označíme čísly a zkoušíme provést operaci spojování mezi každou dvojicí. Operace spojování je definována jako

$$a \cdot \bar{b} + a \cdot b = a,$$

kde a může být jak samostatná proměnná, tak i logický výraz a b je vždy samostatná proměnná. Operací spojování se zřejmě vyloučila proměnná b . Všimneme si, že podmínkou ke spojování mintermů je to, že musí být sousední tj. liší se jen v jedné proměnné – u jednoho je přímá (nenegovaná), u druhého je negovaná. Každý minterm, který byl spojen s jiným, vhodně označíme. Výsledky spojování dvojic spojíme log. součtem. Mintermy, které nebylo možné spojit (jsou neoznačené), opíšeme beze změny.

Jako příklad mějme funkci s ÚDNF nalezenou předem z pravdivostní tabulky:

	1✓	2✓	3✓	4✓	5✓	6
$f(a,b,c,d) =$	$a\bar{b}c\bar{d}$ +	$\bar{a}bc\bar{d}$ +	$a\bar{b}cd$ +	$a\bar{b}\bar{c}d$ +	$abc\bar{d}$ +	$\bar{a}bc\bar{d}$

Značkou ✓ je označen minterm, který byl spojován. Výsledky spojování jsou následující:

1-2 ... $a\bar{b}\bar{d}$

1-3 ... nelze spojovat – liší se ve dvou proměnných (c, d)

1-4 ... $a\bar{b}\bar{c}$

1-5 ... nelze spojovat

1-6 ... nelze spojovat

2-3 ... $a\bar{b}c$

2-4 ... nelze spojovat

2-5 ... $a\bar{c}\bar{d}$

2-6 ... nelze spojovat

3-4 ... $a\bar{b}d$

3-5 ... nelze spojovat

3-6 ... nelze spojovat

4-5 ... nelze spojovat

4-6 ... nelze spojovat

5-6 ... nelze spojovat

Člen 6 nebyl spojen se žádným jiným a zůstane beze změny. Výsledek prvého kroku je:

	1✓	2✓	3✓	4	5✓	6
$f(a,b,c,d) =$	$a\bar{b}\bar{d}$ +	$\bar{a}b\bar{c}$ +	$a\bar{b}c$ +	$a\bar{c}\bar{d}$ +	$a\bar{b}d$ +	$\bar{a}bc\bar{d}$

Jednotlivé členy opět očíslovujeme a opakujeme stejný postup. Výsledky spojování jsou tyto:

1-2 ... nelze spojovat

1-3 ... nelze spojovat

1-4 ... nelze spojovat

1-5 ... \bar{ab}

2-3 ... \bar{ab}

2-4 ... nelze spojovat

2-5 ... nelze spojovat

3-4 ... nelze spojovat

3-5 ... nelze spojovat

4-5 ... nelze spojovat

Členy 4 a 6 nebyly spojeny se žádnými jinými a zůstanou beze změny. Výsledek druhého kroku je:

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

Další spojování již zřejmě není možné.

Pokud již nelze původní ÚDNF uvedeným postupem dále zjednodušovat, dostáváme **koncovou** disjunktní normální formu (KDNF). Její členy se nazývají **implikanty**. Na rozdíl od mintermu, který pokrývá jen jeden bod funkce, každý implikant pokrývá **více než jeden** bod funkce. Jestliže při spojování byla vyloučena jedna proměnná, bude implikant pokrývat dva body. Tak např. implikant xz funkce $f(x,y,z)$ pokrývá mintermy xyz i xzy . Jestliže byly vyloučeny dvě proměnné, bude implikant pokrývat čtyři body, atd. Implikant považujeme za **větší**, když pokrývá větší počet bodů, tj. když je tvořen **menším** počtem proměnných.

V případě, že logická funkce má **neurčené stavы**, je nejlépe všechny nejprve doplnit na 1. Je tak více možností spojování. Ty z neurčených stavů, které nebylo možné spojit s jiným členem, mohou být pak vypuštěny (tj. doplněny na 0), aby zbytečně nekomplikovaly výsledek.

K plnému pokrytí všech mintermů ÚDNF často nepotřebujeme vždy všechny implikanty z KDNF. Některé se mohou ukázat jako **nadbytečné**, mohou být vyloučeny, a tím je dosaženo dalšího zjednodušení funkce. Přehlednou informaci dává tzv. **tabulka implikantů**. Příklad následuje:

Je dána ÚNDF

$$f(x,y,z) = xyz + x\bar{y}z + \bar{x}\bar{y}z + \bar{x}y\bar{z}$$

Byla nalezena KDNF

$$f(x,y,z) = xz + \bar{y}z + \bar{x}y$$

Tab. 6.3 Příklad tabulky implikantů

Implikanty	Mintermy			
	xyz	$x\bar{y}z$	$\bar{x}\bar{y}z$	$\bar{x}y\bar{z}$
xz	✓	✓		
$\bar{y}z$		✓	✓	
$\bar{x}y$			✓	✓

Značka \checkmark znamená „pokrývá“. Zřejmě mohl být vypuštěn implikant $\bar{y}z$, neboť všechny mintermy zůstávají pokryty i potom.

Přesněji lze postup minimálního pokrytí popsat takto: implikant xz pokrývá minterm xyz jako jediný a proto nemůže být vypuštěn – je **podstatný**. Takže je jím pokryt i minterm xyz . Obdobně i implikant $\bar{x}y\bar{z}$ který jako jediný pokrývá minterm $x\bar{y}z$ (a tedy nemůže být vypuštěn), pokrývá i minterm $x\bar{y}z$. Tím jsou ale v tomto případě pokryty všechny mintermy a implikant $y\bar{z}$ je již zbytečný. Jeho zahrnutí do výsledného výrazu by sice nebylo na závadu, ale bez něj je výraz kratší.

Z příkladu plyne **formální postup**:

S pomocí tabulky implikantů se vyloučí nadbytečné implikanty tak, aby všechny mintermy zůstaly pokryty a výsledný výraz aby obsahoval:

- co nejmenší počet implikantů,
- každý z nich co největší (tj. s nejmenším počtem písmen).

Výsledkem je **minimální disjunktní forma** (MDNF). Někdy nastane situace, že jedna a tatáž funkce může mít i více různých MDNF. Proces hledání MDNF se nazývá **minimalizace**.

V případě funkcí s **neurčenými stavami** se v tabulce implikantů uvádějí jen ty mintermy, které patří plně určeným stavům (v pravdivostní tabulce je 1). Tím se pokrytí zjednoduší.

Vzhledem ke způsobu hledání MDNF je zřejmé, že kriteriem je minimální počet písmen. Za písmeno považujeme každou proměnnou v MDNF. Pokud se písmeno vyskytuje několikrát (v různých implikantech), počítáme je pokaždé.

Toto kriterium minimálního počtu písmen nevystihuje přesně složitost obvodů, kterými má být daná logická funkce realizována. Přesto však je alespoň částečná minimalizace dobrým východiskem pro návrh kombinačního obvodu.

6.6 Skupinová minimalizace

Dosud se jednalo o minimalizaci jedné funkce několika proměnných. Běžné kombinační obvody ovšem mají zpravidla několik výstupních proměnných. Je samozřejmě možné navrhnut kombinační obvod tak, že se na jednotlivé výstupy díváme jako na **zcela izolované** logické funkce, které mají společné jen vstupní proměnné. Minimalizují se každá zvlášť. Realizovaný kombinační obvod pak analogicky sestává z jednotlivých izolovaných částí. To sice garantuje požadovanou funkci, mnohdy ale značně **neúsporně**. Často se totiž stane, že implikanty jedné funkce by mohly být **vícenásobně** použity v jiných funkciích, což zjednoduší celkový kombinační obvod. Vícenásobné využití implikantů se může ukázat jako tak výhodné, že stojí za to je podporovat i za cenu složitějšího (neminimálního) tvaru jednotlivých funkcí. Pro skupinovou minimalizaci sice existují systematické metody, jsou však dosti pracné. Částečným řešením je minimalizace každé funkce zvlášť a následně sestavení **seznamu** všech **implikantů**. Z tohoto seznamu jsou pak implikanty dosazovány do jednotlivých funkcí, při čemž v obvodu jsou realizovány jen jedenkrát.

Například kombinační obvod se čtyřmi vstupy (a, b, c, d) a dvěma výstupy (y, z) je popsán těmito funkcemi v minimální formě:

$$y(a, b, c, d) = \bar{b}\bar{d} + b\bar{c}\bar{d} + abd$$

$$z(a, b, c, d) = b\bar{d} + bc + \bar{b}\bar{c}\bar{d} + ab$$

Neexistují žádné společné implikanty. Kromě negací bude k realizaci celého kombinačního obvodu zapotřebí celkem 7 členů AND a 2 členy OR. Kdyby se funkce z ponechala v ekvivalentním, ale jen neúplně minimalizovaném tvaru (o ekvivalence obou funkcí se lze přesvědčit vyplněním pravdivostní tabulky nebo z map), mohly obě funkce vypadat takto:

$$y(a, b, c, d) = \bar{b} \bar{d} + b \bar{c} \bar{d} + abd$$

$$z(a, b, c, d) = b \bar{c} \bar{d} + bc + \bar{b} \bar{c} d + abd \quad (\text{tvar sám o sobě složitější, ale...})$$

V seznamu implikantů by bylo jen 5 různých implikantů. Zde jsou implikanty $b \bar{c} \bar{d}$ a abd využity dvakrát, ovšem v obvodu budou vytvářeny jen jednou. Proto bude zapotřebí celkem jen 5 členů AND a 2 členy OR.

6.7 Mapy a jejich použití

Jednou z možností zápisu funkce je mapa. Nejpoužívanější je mapa **Karnaughova** (čti „karnau“). Je uspořádána do čtverce nebo obdélníka a to tak, že **sousední** pole se liší vždy jen v **jedné proměnné**. Tab. 6.4 ukazuje mapu pro 3 proměnné, tab. 6.5 mapu pro 4 proměnné.

Tab. 6.4 Mapa Karnaugh pro 3 proměnné

		b	
		a	
c		X	

Tab. 6.5 Mapa Karnaugh pro 4 proměnné

		b			
		a			
c		1	0	0	-
		1	1	1	1
		1	0	0	-
d		1	0	0	-

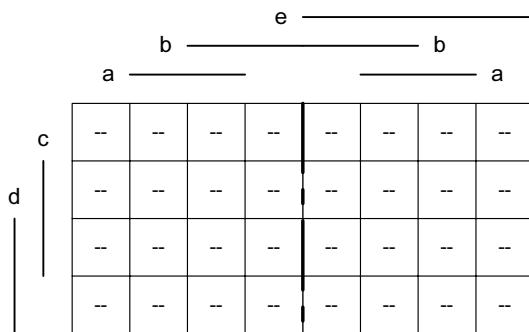
Každému jednotlivému **poli** mapy jednoznačně odpovídá **kombinace hodnot všech proměnných**. Nalézá-li se pole pod pruhem označeným u proměnné, bude tato proměnná negovaná. Nalézá-li se mimo pruh, bude proměnná negovaná. Tak např. pole označené jako X v mapě v tab. 6.4 bude odpovídat kombinaci $\bar{a} \bar{b} c$. Z pravdivostní tabulky funkce lze snadno sestavit její mapu a naopak. Řádkům pravdivostní tabulky, ve kterých je funkční hodnota 1, odpovídají pole mapy s vepsanou jedničkou; obdobně to platí i pro nuly. V mapě lze znázornit i **neurčené stavy** (pomlčkou) podobně jako v pravdivostní tabulce.

V tab. 6.5 je znázorněna logická funkce

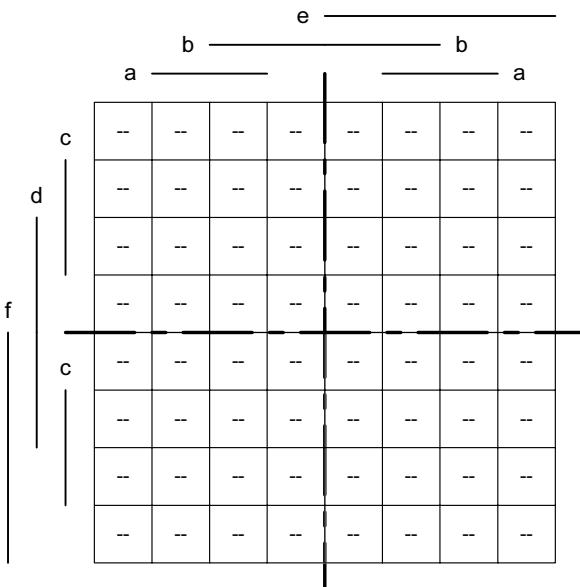
$$f(a,b,c,d) = \sum m(0,1,2,3,6,10,14,(4,5,7))$$

Mapy lze rozšířit i na větší počet proměnných. Tab. 6.6 ukazuje mapu pro 5 proměnných a tab. 6.7 pro 6 proměnných. Mapa pro 5 proměnných se dostane překlopením mapy pro 4 proměnné podle svislé osy symetrie a přidáním páté proměnné, mapa pro 6 proměnných se dostane překlopením mapy pro 5 proměnných podle vodorovné osy symetrie a přidáním šesté proměnné.

Tab. 6.6 Mapa Karnaugh pro 5 proměnných



Tab. 6.7 Mapa Karnaugh pro 6 proměnných



Konstrukce Karnaughovy mapy je významná tím, že souřadnice polí jsou uspořádány tak, že u sousedních polí se liší jen v jedné proměnné. Tudíž geometricky sousední pole jsou sousední i v algebraickém smyslu (liší se v jediné proměnné). Každé pole s hodnotou 1

odpovídá **mintermu** z pravdivostní tabulky. Sousední pole tedy odpovídají mintermům lišícím se jen jednou proměnnou, a ty lze **spojoval do implikantu**. Sousední jsou i pole na okrajích mapy, neboť i ta se liší jen v jedné proměnné (konce řádek, konce sloupců a rohy mapy). U mapy pro 5 a 6 proměnných jsou sousední i pole symetrická podle osy (os) souměrnosti. Spojování polí se vyznačí **smyčkou**. Pole po dvojcích sousední lze spojovat do větších smyček, ty opět do větších atd. Každá smyčka tedy musí mít stranu dlouhou právě 2^k polí, kde k je celé kladné číslo. Smyčky zahrnují 2 pole, 4 pole, 8 polí, atd.

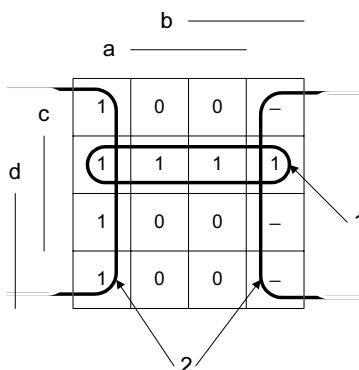
Každá **smyčka** v mapě odpovídá **implikantu** funkce. Princip **minimalizace** spočívá v pokrytí všech jedniček (a libovolných neurčených stavů) soustavou smyček, přičemž:

- smyčky musí být co možná největší
- smyček musí být co nejmenší počet

Tento princip je ilustrován na funkci čtyř proměnných v mapě podle *tab. 6.8*.

$$f(a,b,c,d) = \sum m(0,1,2,3,6,10,14, (4,5,7,15))$$

Tab. 6.8 Příklad minimalizace funkce 4 proměnných



Smyčky byly vytvořeny tak, aby každá zahrnovala co největší počet polí s vepsanou jedničkou nebo neurčenou hodnotou. Smyčka 1 odpovídá implikantu cd' (na proměnných a a b nezáleží), smyčka 2 odpovídá implikantu a (na proměnných b, c, d) nezáleží. Smyčka 2 je sice na obrázku složená ze dvou polovin, splňuje ale podmínky pro tvorbu smyček. Mapu si můžeme představit tak, že byla původně nakreslena na povrchu koule, takže pole na koncích řádků a sloupců byla geometricky sousední. Pak byla rozříznuta a vyrovnána do roviny. Původně sousední pole se tak ocitla na koncích řádek a sloupců, ale v algebraickém smyslu (liší se jen v jedné proměnné) sousedními zůstala.

Podmínka vytvoření co největších smyček je splněna zahrnutím neurčených stavů do smyčky. Byly tedy výhodně doplněny na 1. Zbývající neurčený stav by vyžadoval vytvoření nové smyčky, což by funkci zbytečně komplikovalo. Byl tedy doplněn na 0. Kdyby se smyčka 2 vytvořila jen přes levý sloupec (neurčené stavy v pravém sloupci doplněny na 0), odpovídala by implikantu ab . To by sice nebylo nesprávné, ale výsledný výraz pro funkci by byl složitější. Obdobně i smyčka 1 mohla zahrnovat jen dvě prostřední pole řádku, ale implikant by měl o písmeno více ($ac\bar{d}$ místo $c\bar{d}$). Dvě krajní pole jsou tak pokryta dvakrát, to však

nemá na funkční hodnotu vliv. Odpovídá to **vícenásobnému pokrytí** některých bodů funkce v pravdivostní tabulce. Výsledná funkce v minimálním tvaru je:

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

Minimalizace funkcí v mapách odpovídá minimalizaci ÚDNF takto:

minterm

... pole mapy

implikant

... smyčka

minimální počet písmen v implikantu ... maximální velikost smyčky

minimální počet implikantů ... minimální počet smyček

Využívání nadbytečných implikantů přitom v mapách odpadá, neboť nadbytečné smyčky se vzhledem k přehlednosti mapy vůbec nezavádějí. Je třeba počítat s tím, že funkce může mít několik MDNF, což v mapách znamená několik možných variant výběru smyček. To ilustrují mapy v *tab. 6.9* a *tab. 6.10*, kde jsou smyčky sice vytvořeny dvěma způsoby, ale oba dřívají stejně složité výrazy:

Tab. 6.9 První varianta minimalizace funkce

$$f(a,b,c) = \bar{b}\bar{c} + ac + \bar{a}\bar{b}$$

		b	
		a	
c		1	1
	0	0	1

Tab. 6.10 Druhá varianta minimalizace funkce

$$f(a,b,c) = \bar{a}\bar{c} + ab + bc$$

		b	
		a	
c		1	1
	0	1	1

Obě funkce jsou ekvivalentní – jejich pravdivostní tabulky by byly shodné, lze jenahradiť jedinou *tab. 6.11*.

Pro jednoduché případy není třeba sestavovat tabulku jako mezistupeň, ale z požadavků na chování obvodu lze přímo sestavit mapu. Mapy dřívají velmi názorný pohled na logické funkce a jejich minimalizaci. To je jejich hlavní význam. **Praktické využití** při návrhu číslcových systémů je však velmi **malé**. To souvisí s pojmem sousednosti. Až do čtyř proměnných je vytváření optimální soustavy smyček jednoduché. Pro 5 proměnných je již třeba větší představivosti ve využití osové symetrie, pro 6 proměnných je to ještě obtížnější. Pro větší počet proměnných již mapy zcela ztrácejí výhodnost. Lze tedy konstatovat, že jejich použitelnost končí u pěti, maximálně šesti proměnných. Tento rozsah je však pro praktické návrhy příliš malý.

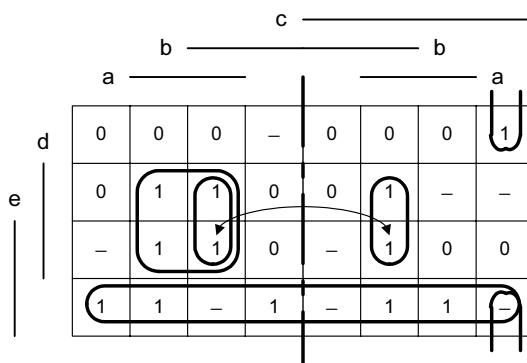
Tab. 6.11 Pravdivostní tabulka funkcí podle tab. 6.9. a tab. 6.10

a	b	c	f(a,b,c)
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Následuje příklad použití mapy pro minimalizaci funkce 5 proměnných – viz tab. 6.12. Využitím osové symetrie vzniká smyčka abd , která je složena ze dvou symetricky umístěných složek. Jejich souvislost je vhodné vyznačit šípkami. Výsledný výraz pro funkci byl:

$$f(a,b,c,d,e) = \bar{acd} + abd + \bar{de} + \bar{a}\bar{bcd}\bar{d}$$

Tab. 6.12 Příklad minimalizace funkce 5 proměnných



6.8 Návrh kombinačních obvodů

Při návrhu kombinačního obvodu se vychází z jeho popisu buď pravdivostní tabulkou, nebo výrazem, nebo slovním popisem. Z tabulky lze standardním postupem získat výraz. Ze slovního popisu lze rovnou získat výraz, nebo pravdivostní tabulku a z té teprve výraz. Výraz se pak dále upravuje do tvaru, který vede na nejhodnější realizaci danými součástkami.

Někdy je k dispozici schéma obvodu, které je třeba analyzovat. Buď potřebujeme získat popis obvodu, nebo je třeba obvod přepracovat do vhodnější podoby. Každopádně se bude jednat o úlohu **analýzy**. Při analýze daného obvodu je nejlépe vyznačit **vnitřní signály**.

Začne se od konce a postupně se dosazuje. Například pro obvod na obr. 6.1 tak lze dospět k výsledku:

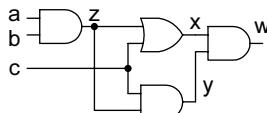
$$w = x \cdot y$$

$$x = c + z$$

$$y = c \cdot z$$

$$z = a \cdot b$$

$$w = (c + z) \cdot cz = (c + ab) \cdot cab = abc + abc = abc$$



Obr. 6.1 Označení vnitřních signálů pro analýzu obvodu

Původní obvod byl zřejmě zbytečně složitý, což ovšem nebylo na první pohled patrné. V případě práce s obvody NAND a NOR bývá nutné mnohonásobně aplikovat **DeMorganova** pravidla. Výsledek analýzy obvodu podle obr. 6.2 opět svědčí o zbytečně složitém obvodu.

$$w = \overline{b + d}$$

$$a = \overline{xyz}$$

$$b = \overline{y + z}$$

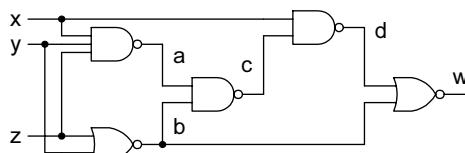
$$c = \overline{ab}$$

$$d = \overline{xc}$$

$$w = \overline{\overline{y + z} + \overline{xc}} = (y + z) \cdot xc = (y + z) \cdot x \cdot \overline{ab} = (y + z) \cdot x \cdot \overline{xyz} \cdot \overline{(y + z)}$$

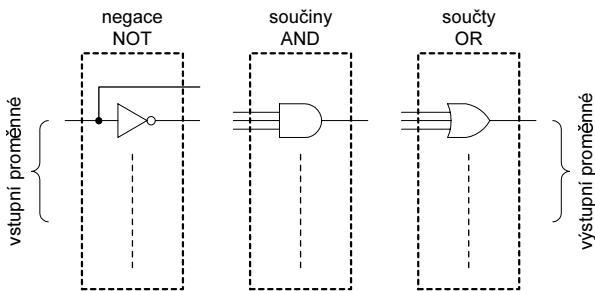
$$w = (y + z) \cdot x \cdot (xyz + y + z) = (y + z) \cdot x \cdot (y + z) \dots \text{operace pohlcení } xyz$$

$$w = x \cdot (y + z)$$



Obr. 6.2 Analýza obvodu s použitím DeMorganových pravidel

Při návrhu obvodu lze vycházet z disjunktní formy (t.j. součtu součinů). Obvod je pak vždy **nejvýše třístupňový**. Prvý stupeň realizuje negace proměnných, druhý stupeň součiny (implikanty), třetí stupeň jejich součet. Jsou-li dodávány vstupní signály z předchozích obvodů v přímém i negovaném tvaru (např. z klopných obvodů), postačí obvod **dvojstupňový**. Předběžná znalost počtu stupňů obvodu, přes které prochází signál, umožňuje předvídat jeho dobu zpoždění, což je evidentně velmi podstatná informace. Jednotlivé stupně ukazuje obr. 6.3.



Obr. 6.3 Obvod, odpovídající disjunktní formě funkce

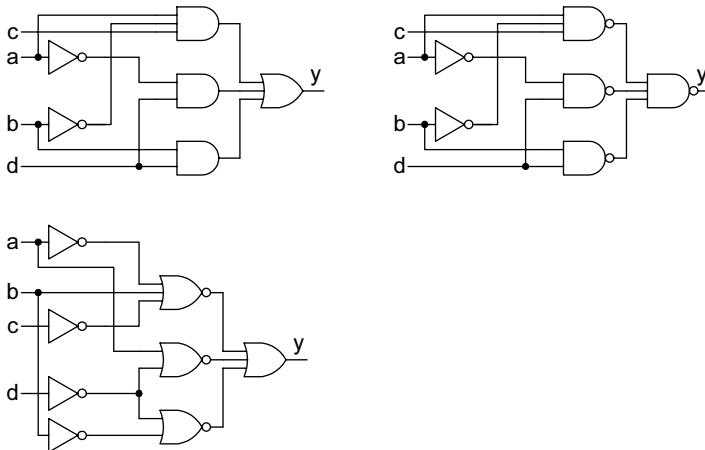
Při návrhu obvodů s logickými členy s negací (NOR, NAND) je postup stejný jako u obvodů s členy AND-OR, pouze se aplikují DeMorganovy vztahy. Na začátku úprav je zavedení dvojí negace výrazu (sudý počet negací nemění výsledek). Například výraz $y = abc + ad + bd$ odpovídá obvodu na obr. 6.4 vlevo nahoře. Výraz upravíme:

$$y = \bar{a}\bar{b}c + \bar{a}\bar{d} + bd = \overline{\overline{\bar{a}\bar{b}c} + \bar{a}\bar{d} + bd} = \overline{\overline{\bar{a}}\cdot\overline{\bar{b}}\cdot\overline{c} + \overline{\bar{a}}\cdot\overline{d} + \overline{b}\cdot\overline{d}}$$

Obvod (na obrázku vpravo) bude opět třístupňový, tvořený stupni NOT-NAND-NAND. Pokud chceme dát přednost obvodům NOR před NAND, naznačíme dvojí negaci jinak:

$$y = \bar{a}\bar{b}c + \bar{a}\bar{d} + bd = \overline{\overline{\bar{a}\bar{b}c} + \overline{\bar{a}\bar{d}} + bd} = \overline{\overline{\bar{a} + b + \bar{c}} + \overline{\bar{a} + d} + \overline{b + d}}$$

Obvod (na obrázku vlevo dole) bude opět třístupňový se stupni NOT-NOR-OR.



Obr. 6.4 Různá zapojení obvodů se stejnou funkcí

Někdy je možné funkci tak upravit, že postačí **nenegované** vstupní proměnné. Úspěch však nelze zaručit. Proto dostupnost vstupních proměnných i v negované podobě je podstatným přínosem. Odpadá pak první stupeň, což zkrátí délku obvodu o třetinu – to je podstatné pro zrychlení činnosti systému. Negace jsou automaticky vytvářeny např. na výstupech klopných obvodů (Q a \bar{Q}) v předřazených registrech.

Dalšího zjednodušení obvodu lze někdy dosáhnout úpravou MDNF pomocí **závorek**. Pro tyto úpravy však neexistuje žádná systematická procedura a nelze zaručit, že výsledný obvod bude jednodušší. Přesto je užitečné je zkusit. Například pro realizaci funkce

$$y = ab + cd + ad + ac$$

je potřeba celkem 5 členů ($4 \times$ součin, $1 \times$ součet). Při vytknutí a před závorky dostaneme

$$y = a \cdot (b + c + d) + cd$$

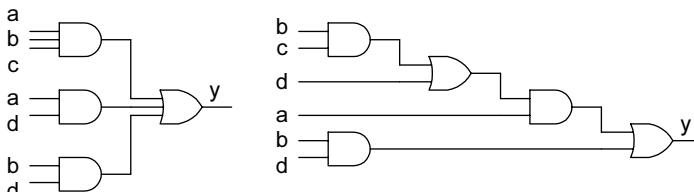
Výsledný obvod lze realizovat jen 4 členy. Často se však obvod naopak zkomplikuje, jak ukazuje příklad funkce

$$y = abc + ad + bd$$

K realizaci je potřeba celkem 4 členů ($3 \times$ součin, $1 \times$ součet). Při vytknutí a před závorky:

$$y = a \cdot (bc + d) + bd$$

Výsledný obvod vyžaduje 5 členů, jak ukazuje obr. 6.5 – vlevo před úpravou, vpravo po ní.



Obr. 6.5 Vliv úprav pomocí závorek na tvar obvodu

Tato úprava tedy rozhodně nevede k lepší realizaci. Navíc prodlužuje obvod, který má teď 4 stupně. Vytýkáním před závorky roste délka obvodu (tj. počet členů zapojených za sebou) a tím i jeho zpoždění.

Velmi často vedou různé úpravy na zcela rozdílné struktury obvodu, přesto však může být počet logických členů stejný. Je zřejmé, že MDNF **není zárukou** nalezení nejjednoduššího obvodu, i když může být dobrým východiskem. Při návrhu kombinačních obvodů je třeba respektovat ještě **další požadavky**. Těmi jsou zatížení jednotlivých členů, rozdělení rozsáhlého obvodu na vhodné konstrukční celky, odstranění nežádoucích přechodných dějů, zpoždění obvodu, využití standardních funkčních celků.

Zpoždění logického členu je závislé na zátěži (zvláště na její kapacitní složce) a na počtu vstupů – členy s větším počtem vstupů (OR, NOR, AND, NAND) mají větší zpoždění. Obecně lze zpoždění průchodu signálu členem t_{pd} vypočítat takto (viz [Erc99]):

$$t_{pd} = K_k + K_i \cdot N_i + K_o \cdot N_o,$$

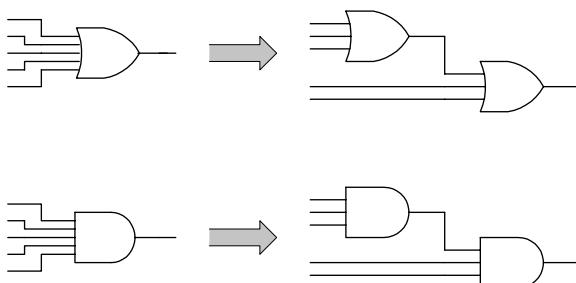
- kde K_k je konstantní složka rovná zpoždění členu s 1 vstupem a bez zátěže
 K_i je přírůstek zpoždění na každý další přidaný vstup členu
 N_i je počet vstupů (tzv. vstupní rozvětvení, angl. *fan-in*)
 K_o je přírůstek zpoždění na zátěž výstupu vstupem dalšího obvodu téže technologie
 N_o je počet takových zátěží (tzv. výstupní rozvětvení, angl. *fan-out*)

Důsledkem může být to, že člen s velkým vstupním i výstupním rozvětvením může být dokonce pomalejší než dva méně zatížené a méně rozvětvené členy v sérii. To naznačuje možné úpravy. Ty jsou založeny na **asociativním** zákonu:

$$a + b + c + d + \dots = a + b + c + (d + \dots)$$

$$a \cdot b \cdot c \cdot d \cdot \dots = a \cdot b \cdot c \cdot (d \cdot \dots)$$

Obr. 6.6 ukazuje obvodové řešení pro případ snížení počtu vstupů z 5 na 3 u členů OR a AND.



Obr. 6.6 Snížení počtu vstupů u členů OR a AND

Pro zvláště velké zátěže jsou v každé technologické řadě k dispozici **výkonové členy** nebo **budiče** (angl. *buffer*) se zvýšeným výstupním proudem (kolem 50 mA) a výstupem buď negovaným nebo nenegovaným.

6.9 Využití multiplexorů

V případě funkce velkého počtu proměnných je možné počet proměnných snížit použitím **Shannonovy věty o rozkladu**:

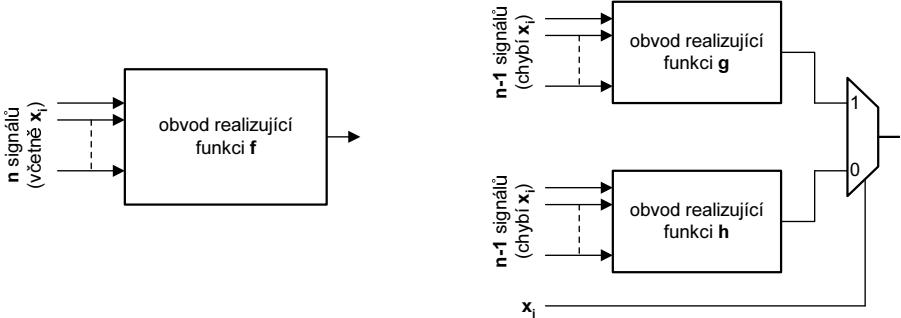
$$f(x_n, x_{n-1}, \dots, x_i, \dots, x_1) = x_i \cdot f(x_n, x_{n-1}, \dots, 1, \dots, x_1) + \bar{x}_i \cdot f(x_n, x_{n-1}, \dots, 0, \dots, x_1)$$

Symbolicky lze napsat:

$$f(\mathbf{n} \text{ proměnných}) = x_i \cdot g(\mathbf{n-1} \text{ proměnných}) + \bar{x}_i \cdot h(\mathbf{n-1} \text{ proměnných})$$

Vztah lze snadno odvodit: je-li proměnná $x_i = 1$ (a tudíž $\bar{x}_i = 0$), bude platný první člen na pravé straně a místo proměnné x_i v něm bude jednička. Obdobně při $x_i = 0$ bude platný druhý člen a místo proměnné x_i v něm bude nula. Funkce na pravé straně budou mít jen $n-1$ proměnných. Obr. 6.7 ukazuje, jak se tato úprava projeví v obvodu.

Za výstupy bloků g a h následuje **multiplexor**, což je číslicový přepínač. Symboly „0“ a „1“ u jeho vstupů vyznačují, při které hodnotě signálu x_i je daný vstupní signál převeden na výstup. Zde se jedná o dvojkanálový multiplexor – existují i multiplexory vícekanálové (viz další text).



Obr. 6.7 Obvodová implementace Shannonovy věty o rozkladu

Ve snižování počtu proměnných lze dále pokračovat a provést rozklady funkcí g a h podle další proměnné. Zřejmě se tak dojde ke kaskádě multiplexorů. Multiplexor je velmi často využíván k realizaci kombinačních funkcí zvláště u obvodů vyšší integrace.

Jako příklad rozložíme funkci

$$z(x_3, x_2, x_1, x_0) = x_3(x_1 + x_2 x_0)$$

podle proměnné x_0 . Dostaneme:

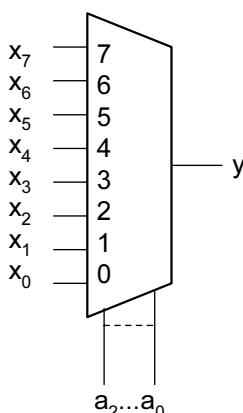
$$z(x_3, x_2, x_1, 0) = x_3 x_1$$

$$z(x_3, x_2, x_1, 1) = x_3(x_1 + x_2)$$

Takže

$$z = (x_3 x_1) \bar{x}_0 + x_3(x_1 + x_2)x_0$$

Zřejmě proměnná x_0 bude řídit multiplexor, který bude přepínat mezi funkcemi $x_3 x_1$ a $x_3(x_1 + x_2)$. Obě funkce jsou jednodušší než funkce původní. Mohou být ovšem opět rozloženy, tentokrát podle proměnné x_1 . Nakonec se celá původní funkce může rozložit tak, že je realizována výhradně multiplexory. V praxi se s rozkladem skončí při dosažení přijatelného počtu proměnných v dílčích funkčích, např. 4 nebo 5 – to záleží na způsobu realizace obvodu.



Obr. 6.8 Osmikanálový multiplexor

Užitečnost multiplexoru je zřejmá z výrazu popisujícího jeho funkci. Předpokládejme multiplexor se skupinou k signálů pro volbu kanálu. Jedná se o vektor s významem **adresy**. Při k složkách lze rozlišit 2^k kombinací a tím tedy vybrat jeden z 2^k možných vstupních kanálů – viz obr. 6.8 pro případ multiplexoru s osmi kanály.

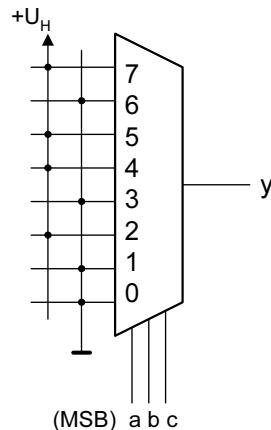
Při adrese 000 bude vybrán vstup x_0 , při adrese 001 ($a_2=\text{MSB}$) vstup x_1 , atd. Obecně bude pro výstup y platit:

$$y = \bar{a}_2 \bar{a}_1 \bar{a}_0 \cdot x_0 + \bar{a}_2 \bar{a}_1 a_0 \cdot x_1 + \dots + a_2 a_1 a_0 \cdot x_7$$

Jestliže na vstupy x_7 až x_0 přivedeme konstanty 0 nebo 1, budou ve výrazu pro y ponechány jen některé součiny $\tilde{a}_2 \tilde{a}_1 \tilde{a}_0$ (vlnovka znamená možnou, ale ne nutnou negaci). Zcela zřejmě dostaneme výraz pro y jako **součet mintermů** z proměnných a_2, a_1, a_0 . Osmikanálovým multiplexorem tak lze realizovat libovolnou funkci 3 proměnných. Obdobnou úvahu lze aplikovat i na další rozměry multiplexorů a počty proměnných.

V následujícím příkladu je na obr. 6.9 multiplexorem realizována funkce, zadaná pravdivostní tabulkou.

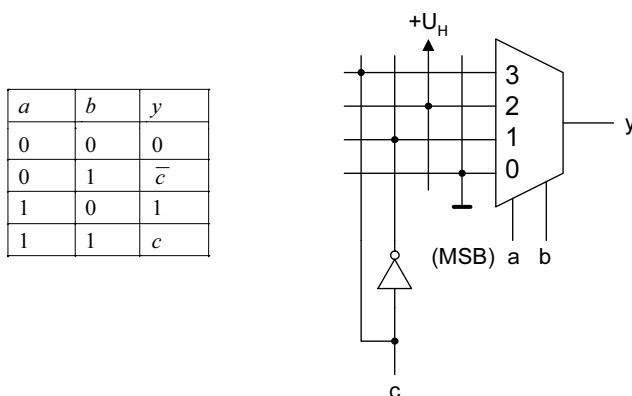
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



Obr. 6.9 Tabulka funkce a její realizace multiplexorem

Je zřejmé, že tato realizace nepotřebuje žádnou minimalizaci, neboť složitost obvodu je pro všechny funkce 3 proměnných stejná. Mění se jen propojení vstupních kanálů se zemí nebo s U_H . Toto propojení lze vhodně konstrukčně usporádat tak, že je programovatelné – prokovenými dírami na plošném spoji, přerušením cest na plošném spoji, v integrované verzi programovatelnými spínači, atd.

Počet adresových vstupů multiplexoru (a tím i počet jeho kanálů) lze ještě dále snížit. Všimneme si, že v pravdivostní tabulce příkladu nahoře je v prvních dvou řádkách výstup konstantní 0, nezávisle na c . Ve druhých dvou řádkách je výstup negací c . Ve třetích dvou řádkách je výstup konstantní 1, nezávisle na c . A konečně v posledních dvou řádkách je roven c . Dostaneme tak novou pravdivostní tabulkou a zapojení s multiplexorem jen o čtyřech kanálech (viz obr. 6.10).



Obr. 6.10 Snížení počtu adresových vstupů multiplexoru

6.10 Využití členů EX-OR

V některých případech vede použití členů EX-OR (exclusive OR) nebo zkráceně XOR na jednodušší realizaci obvodu. Funkce XOR je znázorněna v pravdivostní tabulce v porovnání s funkcí OR.

Tab. 6.13 Porovnání funkce OR a XOR

<i>a</i>	<i>b</i>	XOR <i>a</i>⊕<i>b</i>	OR <i>a</i>+<i>b</i>
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

Rozdíl je jen ve čtvrté řadce. Funkce XOR vylučuje případ obou jedniček na vstupech, proto název „vylučující“ (angl. exclusive) OR. Jako operátor je zaveden symbol \oplus . Ten naznačuje součet a má smysl **aritmetického** součtu, ale v aritmetice **mod 2**. Skutečně, při zanedbání přenosu do vyššího řádu je hodnota XOR rovna aritmetickému součtu dvou jednobitových čísel *a* a *b*. Proto se funkce XOR též někdy nazývá „součet modulo 2“.

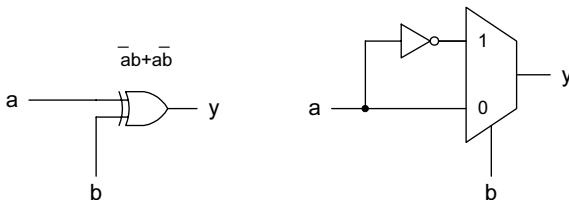
Kromě operátoru \oplus lze funkci rozepsat i jako součet (logický) součinů:

$$y = a \oplus b = \bar{a}b + a\bar{b}$$

Z druhého výrazu je vidět zajímavou obvodovou vlastnost členu XOR. Díváme-li se na jeden jeho vstup (např. *a*) jako na vstup signálu procházejícího na výstup *y*, a na druhý vstup (např. *b*) jako na vstup řídící, pak

$$\begin{array}{lll} \text{při } b = 0 & \text{je} & y = a \\ \text{při } b = 1 & \text{je} & y = \bar{a} \end{array}$$

Je tak k dispozici řízená nebo **programovatelná negace**. Stejnou funkci lze realizovat i invertorem a multiplexorem (viz obr. 6.11).

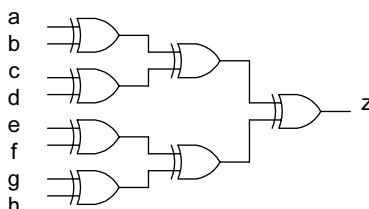


Obr. 6.11 Realizace řízené negace

Jiné výhodné využití členu XOR je v generátorech paritního bitu. Parita se zjišťuje na základě momentálního počtu vstupních signálů s hodnotou 1. Generátor **liché parity** dává na výstupu hodnotu 1, je-li na vstupech **lichý** počet jedniček. Obdobně generátor **sudé parity** dává na výstupu hodnotu 1, je-li na vstupech **sudý** počet jedniček. Při dvou vstupech by postačil člen XOR – z pravdivostní tabulky nahoře je zřejmé, že realizuje lichou paritu. Pro více vstupů lze využít asociativní vlastnosti funkce XOR, podobně jako u funkcí OR a AND:

$$a \oplus b \oplus c \oplus \dots = (a \oplus b) \oplus (c \oplus d) \oplus \dots$$

Například osmivstupový generátor liché parity je na obr. 6.12. Generátor sudé parity by měl negovaný výstup.



Obr. 6.12 Generátor liché parity

Funkce XOR dává hodnotu 1, jsou-li hodnoty na obou vstupech opačné. Proto se též nazývá „**neekvivalence**“. Naopak negaci funkce XOR je funkce „**ekvivalence**“ se symbolem „ \equiv “. Platí:

$$a \equiv b = \overline{a \oplus b}$$

Tak jako funkce XOR mohla být vyjádřena součtem součinů, může i ekvivalence být rozepsána jako:

$$a \equiv b = \overline{ab} + ab$$

Členy XOR mohou být užitečné i při úpravách výrazů, kdy se někdy dojde k potřebě realizace výrazu $\overline{ab} + a\overline{b}$. Například funkce:

$$\begin{aligned} \overline{abc} + a\overline{bc} + \overline{ab}c + abc &= \overline{a}(b\overline{c} + \overline{bc}) + a(\overline{bc} + bc) = \overline{a}(b \oplus c) + a(b \equiv c) = \\ &= \overline{a}(b \oplus c) + a\overline{(b \oplus c)} \end{aligned}$$

Dosadíme-li za $b \oplus c = x$, pak můžeme pokračovat v úpravách až k jednoduchému výsledku:

$$\bar{a}(b \oplus c) + a\overline{(b \oplus c)} = \bar{a}x + a\bar{x} = a \oplus x = a \oplus b \oplus c$$

Členem ekvivalence lze realizovat jednobitový **komparátor**. Číslicový komparátor je obvod, který dává na výstupu jedničku při shodě dvou n -bitových čísel (A, B) přivedených na vstupy. Shoda dvou čísel samozřejmě znamená shodu ve všech dvojicích bitů stejných řádů – tedy musí platit současně $a_{n-1}=b_{n-1}$, $a_{n-2}=b_{n-2}$, ..., $a_0=b_0$. Realizace je zřejmá – každá dvojice bitů bude vyhodnocována jedním členem ekvivalence. Jejich výstupy musí mít všechny současně ukazovat shodu, tedy musí být spojeny logickým součinem. Obecně:

$$K = (a_{n-1} \equiv b_{n-1}) \cdot (a_{n-2} \equiv b_{n-2}) \cdot \dots \cdot (a_0 \equiv b_0)$$

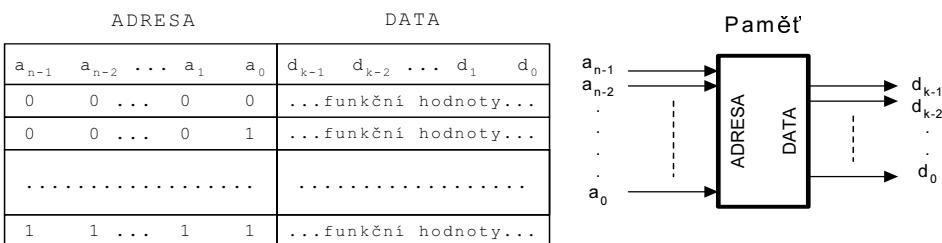
Pro použití členů XOR lze zavést dvojí negaci a následně úpravu podle DeMorgana:

$$\begin{aligned} K &= \overline{\overline{(a_{n-1} \equiv b_{n-1}) \cdot (a_{n-2} \equiv b_{n-2}) \cdot \dots \cdot (a_0 \equiv b_0)}} \\ &= \overline{\overline{(a_{n-1} \equiv b_{n-1})}} + \overline{\overline{(a_{n-2} \equiv b_{n-2})}} + \dots + \overline{\overline{(a_0 \equiv b_0)}} \\ &= (a_{n-1} \oplus b_{n-1}) + (a_{n-2} \oplus b_{n-2}) + \dots + (a_0 \oplus b_0) \end{aligned}$$

Komparátor bude ted' realizován členy XOR a následujícím NOR.

6.11 Realizace kombinačních obvodů paměti

U paměti jsou v tomto případě využity jen adresové vstupy a datové výstupy. Obsah paměti je předpokládán konstantní a definovaný ještě před začátkem její činnosti jako kombinační obvod. Paměť má n adresových vstupů a k datových výstupů. Paměť takového rozsahu lze realizovat skupinou k libovolných funkcí o n proměnných. Důkaz je jednoduchý – tabulka obsahu paměti zahrnuje všechny adresy a na nich uložená data. Tím je formálně zcela shodná s pravdivostní tabulkou funkce. Vstupní proměnné funkcí odpovídají jednotlivým bitům adresy a výstupní proměnné odpovídají jednotlivým bitům dat.



Obr. 6.13 Paměť jako kombinační obvod

Realizace paměti je vhodná u různých převodníků kódů, tabulek hodnot funkcí, apod. Lze tak jednoduše převést lineárně rostoucí číslo na vstupu na sinusovku nebo jinou křivku a generovat tak periodické kmity. Výhodnost se ztrácí u velmi jednoduchých funkcí, ale

velkého počtu proměnných. Uvažme např. funkci AND o 16 proměnných. Na to by postačil jeden logický člen, ale paměť bude mít 2^{16} adres! Pro realizaci funkcí malého počtu proměnných (4 až 5) se však paměť používá velmi často, zvláště v **součástkách PLD** (angl. Programmable Logic Device – programovatelný logický obvod). Využívají se permanentní paměti (ROM), jednou programovatelné paměti (PROM), elektricky vymazatelné paměti (EEPROM, FLASH), a konečně i paměti RAM (ve verzi statické, SRAM). U poslední je nutné vždy po náběhu napájení zajistit její naplnění potřebným obsahem. Je ale proudově nejúspornější a umožňuje neomezený počet přaprogramování a tedy i změn funkce.

7

PŘECHODNÉ DĚJE V KOMBINAČNÍCH OBVODECH

Uvažujeme kombinační obvod s m vstupy (x_1, \dots, x_m) a p výstupy (y_1, \dots, y_p). Při analýze přechodných dějů je třeba rozlišit dva případy:

- Při změně vstupního stavu dochází na sledovaném výstupu ke změně $0 \rightarrow 1$ nebo $1 \rightarrow 0$. Vyšetřují se krajní hodnoty **zpoždění** změny signálu na sledovaném výstupu.
- Při změně vstupního stavu má být na sledovaném výstupu stále stav 1, případně stále stav 0, tj. nemění se. Při některých změnách vstupních stavů však dochází k vytvoření falešných krátkých výstupních impulzů. Vstupní stavy, při kterých se tyto **falešné impulzy** projeví, se nazývají **hazardní stavy** nebo krátce „hazardy“.

Hlavním nebezpečím hazardů je to, že se falešné impulzy mohou, ale nemusí vyskytnout, takže nelze spoléhat na to, že se projeví při ověřovacích zkouškách. Na krátké falešné impulzy mohou některé navazující obvody zareagovat tak, že se dostanou do zcela nepředvidaného stavu. Je proto nezbytné konstruovat číslicové systémy již v počátečních fázích návrhu tak, aby nebyly na případné falešné impulzy citlivé.

V dalších úvahách se bude často jednat o změnách signálu na vstupech, výstupech i vnitřních bodech kombinačního obvodu. Pro stručný zápis zavedeme následující symboliku:

- \tilde{a} proměnná a může mít hodnotu 0 nebo 1 bez vlivu na výstup
 $a\uparrow$ proměnná a se mění střídavě mezi 0 a 1
 $a\uparrow$ proměnná a se změní z 0 do 1
 $a\downarrow$ proměnná a se změní z 1 do 0

K vyjádření kombinace hodnot proměnných (stavu) se jednotlivé proměnné spojí součinem a jsou uvedeny nenegované, mají-li hodnotu 1, a negované, mají-li hodnotu 0. Tak např. zápis $ab\bar{c}d$ znamená kombinaci hodnot $a = 1, b = 0, c = 0, d = 1$. Zápis $\bar{a}b\bar{d}, c\uparrow$ znamená, že $a = 0, b = 1, d = 0, c$ kmitá mezi 0 a 1.

7.1 Zpoždění signálů ze vstupů na výstupy

Zpoždění kombinačního obvodu je dáno dobou od změny vstupního stavu do ustálení výstupního stavu, tj. **všech** výstupních signálů. V důsledku zpoždění jednotlivých logických členů (t_{pd}) nesleduje výstup kombinačního obvodu změny na vstupech okamžitě, ale vždy s jistým zpožděním. Toto zpoždění může v mnoha případech být kritické pro správnou činnost dalších obvodů a měla by tedy být známa jak jeho maximální, tak i minimální hodnota. Zatím co **maximální hodnoty** t_{pd} lze u všech typů logických členů nalézt v katalogu, minimální hodnoty se neudávají. Na základě praktických zkušeností lze počítat asi s třetinou maximální hodnoty. **Skutečná** hodnota zpoždění je tedy **neznámá** a záleží na výběru součástek, napájecím napětí, teplotě, délce spojů a záťaze jednotlivých členů. Měření na reálném obvodu má význam jen pro ověření a kontrolu výpočtů, nemůže však na jeho základě být stanovena zaručovaná hodnota.

Skutečné zpoždění logických členů se jen velmi zřídka bude rovnat některé z krajních hodnot (max. nebo min.). Spíše se bude pohybovat kolem průměrné hodnoty. Tato úvaha stojí na začátku statistických metod výpočtu zpoždění obvodu. Tyto metody vedou k reálnějším odhadům zpoždění, nedávají však **zádnou záruku**, že obvod (výjimečně) nebude mít zpož-

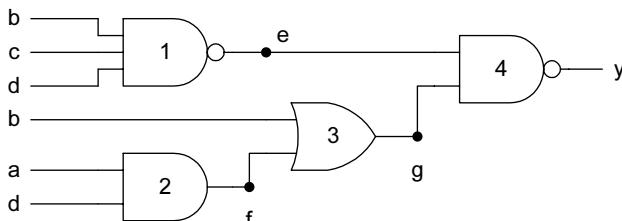
dění zcela jiné (na horní nebo dolní mezi). Mají proto v praxi omezený význam. Naopak rozbor obvodu z hlediska krajních hodnot zpoždění dává realističtější údaje, ze kterých je možno vycházet.

Úkolem je zjistit zpoždění z kteréhokoliv vstupu na výstup – označení $t_p(x_i, y_j)$. K tomu účelu je nutné zjistit **délku cesty** ze vstupu x_i na výstup y_j , tj. počet logických členů v cestě. Při známých krajních hodnotách t_{pd} jednotlivých členů v cestě pak již lze spočítat celkové zpoždění jako sumu jednotlivých zpoždění podél celé cesty. Při velmi přesném výpočtu je třeba brát zřetel na skutečnost, že u většiny členů $t_{pHL} \neq t_{pLH}$ a že tedy bude záležet i na **smystu změny** na vstupu x_i ($0 \rightarrow 1$ nebo $1 \rightarrow 0$).

Zpoždění kombinačního obvodu je dáno dobou od změny vstupního stavu do ustálení výstupního stavu, tj. všech výstupních signálů. V mnoha obvodech však jsou cesty z různých vstupů na různé výstupy **různě dlouhé**. Zpoždění pak je závislé na tom, na kterém vstupu dochází ke změně. Někdy se setkáme i s případem, kdy ze vstupu na výstup existuje více než jedna cesta. Výběr cesty, kterou se signál šíří, může záviset na některých **dalších vstupech**. V každém případě však lze určit nejdelší cestu a tím i **maximum zpoždění**.

Jako příklad vyšetříme zpoždění obvodu na obr. 7.1. Jeho funkci lze po aplikaci DeMorganových zákonů popsat jako:

$$y = \overline{bcd} \cdot (b + ad) = bcd + \overline{ab} + \overline{bd}$$



Obr. 7.1 Obvod s různými dobami zpoždění

Předpokládejme pro jednoduchost u všech členů maximální hodnoty $t_{pLH} = 5$ ns, $t_{pHL} = 3$ ns. Při vstupním stavu $b = 0, c = 0$ nebo $1, d = 1$ a měnícím se a (zkráceně zapsáno $\bar{b}cd, a\uparrow$) je v cestě od a do y přes hradla 2, 3, 4 zpoždění bud'

$$t_p(a, y) = 2 \cdot t_{pLH} + t_{pHL} = 13 \text{ ns}$$

při $a\uparrow$, nebo naopak

$$t_p(a\downarrow, y) = 2 \cdot t_{pHL} + t_{pLH} = 11 \text{ ns}$$

při $a\downarrow$. Změny $b\uparrow$ se projeví na výstupu při $a\bar{c}d$, se zpožděním

$$t_p(b, y) = t_{pLH} + t_{pHL} = 8 \text{ ns}$$

podél cesty 3, 4. Signál d bude postupovat cestou 1, 4 při $\bar{a}bc$ se zpožděním

$$t_p(d, y) = t_{pLH} + t_{pHL} = 8 \text{ ns},$$

nebo cestou 2, 3, 4 při $a\bar{b}\bar{c}$ se zpožděním

$$t_p(d, y) = 2 \cdot t_{pLH} + t_{pHL} = 13 \text{ ns}$$

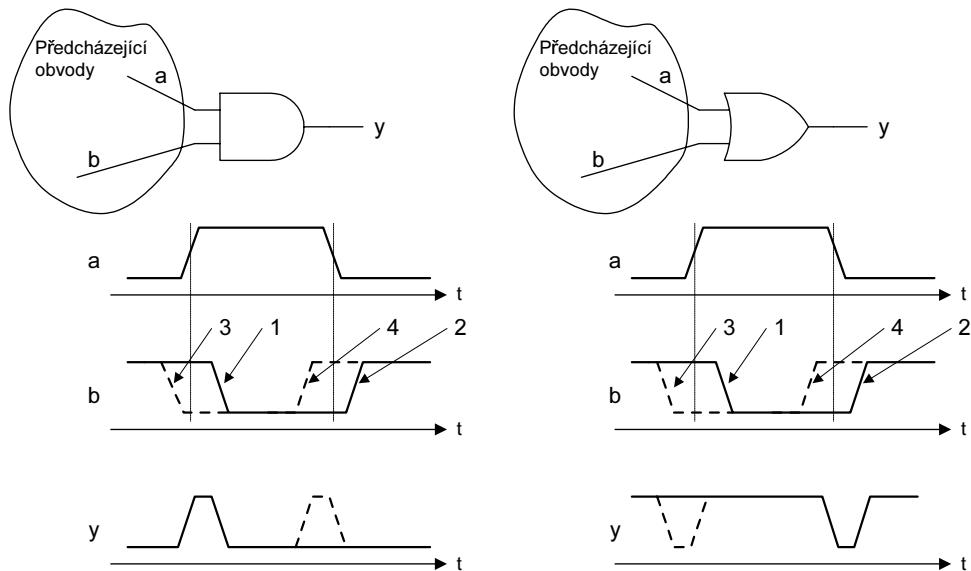
při $d \uparrow$, nebo

$$t_p(d, y) = 2 \cdot t_{pHL} + t_{pLH} = 11 \text{ ns}$$

při $d \downarrow$. Vidíme, že i u takto jednoduchého obvodu se může zpoždění pohybovat v širokých mezích.

7.2 Hazardní stavy v kombinačních obvodech

Uvažujme situaci, kdy na vstupy logického členu AND nebo OR přivedeme z předcházejících obvodů signály, z nichž jeden se mění z 0 na 1 a druhý opačně, ovšem změny obou signálů **nejsou absolutně současné**. Na výstupu vznikne **impulz** o délce dané zpožděním mezi oběma signály. Pro různé typy logických členů je situace shrnuta na obr. 7.2.



Obr. 7.2 Vznik falešného impulzu u obvodu AND a OR

Na levém obrázku s členem AND, při signálu b posunutém za signálem a (průběh s doběžnou hranou 1 a náběžnou hranou 2), je na okamžik $a=b=1$, a tedy $y=1$. Vznikne tak na y impulz kreslený plnou čarou. Při signálu b v předstihu před signálem a (doběžná hrana 3 a náběžná hrana 4) by naopak stejný impulz na y vznikl později, podle čárkovaného průběhu y . Zpoždění signálů by mohla být i jiná – pokud by signál b měl doběžnou hranu 3 a náběžnou 2, impulz na y by nevznikl nikdy. Pokud by ale signál b měl doběžnou hranu 1 a náběžnou 4, vzniknou na y dokonce 2 impulzy (plně a čárkovaně). O tom, **zda** falešný impulz

vznikne, **kdy** vznikne, či zda nevznikne vůbec, tedy rozhodují vzájemná zpoždění signálů. Je to do značné míry náhodná záležitost a proto se tomuto zjevu říká **hazard**. Falešný impulz může způsobit v dalších obvodech stejnou činnost, jako by se jednalo o impulz generovaný záměrně – to je samozřejmě chyba. Bude jen otázkou, zda následující obvody tak krátký falešný impulz zaregistrují. Může se jednat o obvody s dlouhou časovou konstantou (např. ovládání relé apod.), kde krátký impulz nevadí. Obecně ale **je na závadu** a musí se s ním počítat. Obdobná situace nastává i u členu OR podle obrázku vpravo, jen falešný impulz je z 1 do 0. Pro člen NAND a člen NOR samozřejmě platí průběhy *y* negované.

V reálném obvodu nikdy nemůžeme vzájemná zpoždění signálů vyloučit, už jenom proto, že jejich cesty jsou různě dlouhé (přes různý počet členů). Zpoždění předcházejících obvodů dokonce ani přesně neznáme, známe jen krajní hodnoty v rozmezí výrobních tolerancí.

Hazard se objeví, i kdyby se oba signály měnily naprostě současně (což je jen hypotetický předpoklad). Je to v důsledku elektrických vlastností logického členu při zpracování vstupních signálů lichoběžníkového tvaru.

V reálných obvodech tedy většina logických vztahů platí jen v ustáleném stavu. V **přechodném stavu** je musíme upřesnit. V předchozím *obr. 7.2* se signál *b* měnil (přibližně) v protifázi se signálem *a*, byl tedy jeho negací. Očekávali bychom, že

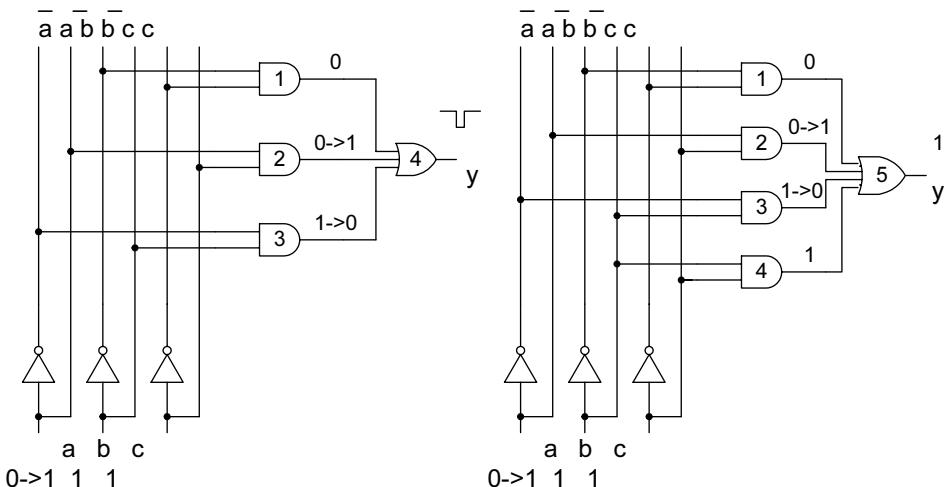
$$a \cdot \bar{a} = 0, \quad a + \bar{a} = 1,$$

což však platí jen v ustáleném stavu. Během přechodu vznikl falešný impulz. Pravidla Booleovy algebry tedy v přechodných dějích **neplatí**. To se týká i operace **spojování** (mintermů) podle vztahu $x\bar{y} + xy = x$. Ten platí jen v ustáleném stavu, neboť $x\bar{y} + xy = x \cdot (y + \bar{y})$, a již víme, že součet $y + \bar{y}$ není 1, ale svědčí o hazardu. Nebezpečí hazardu vzniká v každém obvodu, kde na vstupech jednoho jeho elementárního logického členu se mění **přibližně současně** alespoň dva signály v opačném smyslu.

Má-li člen alespoň jeden další vstup, může jím být vznik hazardu ovlivněn. Je-li totiž na něm taková hodnota, která zaručí **konstantní** hodnotu na výstupu bez ohledu na ostatní vstupy, pak hazard nevznikne. Uvažujme člen AND se třemi vstupy. Na prvním z nich je hodnota 0, na druhých dvou se střídá 0 a 1 v opačné fázi. Tím by samozřejmě vznikal falešný impulz na výstupu. Ale pokud po tu dobu existuje konstantní hodnota 0 na prvém vstupu, bude na výstupu vždy 0 a falešný impulz nevznikne – je blokován. Obdobně je tomu i u členu OR, když na **blokujícím** vstupu je hodnota 1 – ta sama garantuje hodnotu 1 na výstupu.

Blokující signály samozřejmě nemohou blokovat členy trvale – tím by byl celý obvod vyřazen z funkce. Blokování je nutné jen po dobu, kdy by se mohly projevit hazardy. Musí být tedy **součástí návrhu** logické funkce.

Obr. 7.3 ukazuje obvod realizující funkci $y = \bar{b}\bar{c} + ac + \bar{a}b$. Na levé polovině obrázku je na vstup obvodu přivedeno bc , $a\uparrow$. Na dvou vstupech členu 4 (OR) se signály mění ve vzájemně opačném smyslu a na třetím je trvale 0, což neblokuje hazard. Na výstupu bude krátký falešný impulz do 0. Na pravé polovině obrázku je obvod se stejnou funkcí, ale mírně upraven. Při stejném stavu na vstupech je na jednom vstupu členu 5 trvale stav 1, což postačí na trvalé vytvoření 1 na výstupu bez ohledu na změny signálů ze členů 2 a 3.



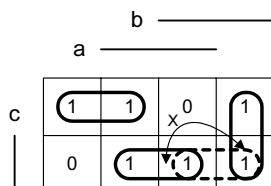
Obr. 7.3 Blokování hazardu signálem s hodnotou 1

Implikanty funkce y jsou $\bar{b}\bar{c}$, ac , $\bar{a}b$, realizované členy 1, 2, 3. Při změně vstupního stavu z $\bar{a}bc$ na abc je hodnota $y=1$ zajišťována nejprve implikantem $\bar{a}b$ a pak ac . Dochází tak k „přeskoku“ mezi dvěma implikanty, při němž ostatní implikanty (zde jen $\bar{b}\bar{c}$) tento přeskok nepřekrývají (nedají hodnotu 1). Na pravém obrázku je navíc vytvořen implikant bc , který při $b=1, c=1$ dává trvale jedničku bez ohledu na změny a , neboť proměnná a v něm vůbec není obsažena – nezáleží na ni. Tím jsou překryty změny na výstupech členů 2 a 3.

7.3 Hledání hazardu pomocí map

Pokrytí funkce implikanty a **přeskoky** mezi nimi jsou velmi názorně vidět v mapách. V tab. 7.1 je uvedena mapa funkce $y = \bar{b}\bar{c} + ac + \bar{a}b$ z obr. 7.3 vlevo. Smyčky v mapě odpovídají implikantům funkce.

Tab. 7.1 Přeskoky mezi implikanty v mapě



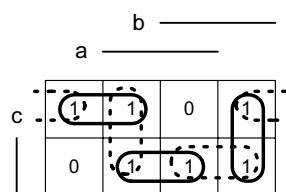
Šipky ukazují přeskoky mezi implikanty. Šipka X odpovídá vstupům bc , $a\downarrow$, kdy vznikal hazard. Na mapě se přeskakuje mezi dvěma smyčkami – ac a $\bar{a}b$. Čárkovaně je naznačena nadbytečná smyčka bc . Přeskakuje se tak uvnitř této smyčky.

Tím je naznačena jedna z metod pro odstranění hazardů. Všechny přeskoky mezi smyčkami je třeba překrýt **dodatečnými**, tj. nadbytečnými či **redundantními** smyčkami. Tím se

pak bude přeskakovat mezi body uvnitř smyčky a hazard je vyloučen. Obvod má redundantní části, je složitější. Tuto metodu však nelze všeobecně použít. Např. v mapě v tab. 7.1 nelze překrýt přeskoky mezi políčky $a\bar{b}c$ a $\bar{a}b\bar{c}$, a dalšími. Aby bylo možné přeskoky vždy překrýt, smí se měnit vždy **jen jedna** proměnná – pak jsou totiž políčka sousední a přes ty lze vždy uzavřít dodatečnou smyčku. Mapu s nadbytečnými smyčkami ukazuje tab. 7.2. Stejně však zřejmě nebylo možné překrýt všechny přeskoky – jen ty mezi sousedními poli. Funkce má nyní tvar $y = \bar{b}\bar{c} + ac + \bar{a}b + \bar{a}\bar{c} + ab + bc$ a je sice složitější, přesto však je ekvivalentní s původní funkcí. Úpravou nového výrazu a jeho zjednodušením pomocí tabulky implikantů se o tom lze přesvědčit, stejně tak vyplněním pravdivostní tabulky.

Zavedení redundancy je cílený postup, **opačný k minimalizaci**.

Tab. 7.2 Překrytí přeskoků nadbytečnými smyčkami



Systémy, kde se vyskytuje změna jen jedné proměnné, jsou vzácné. Patří mezi ně např. optické senzory pro odměřování polohy nebo úhlu natočení a následující obvody. Jako kód se změnou v jedné proměnné se používá **Grayův kód**. Převodní vztahy mezi Grayovým a binárním kódem ukazuje tab. 7.3. Ve třech sloupcích nejvíce vpravo je ilustrováno rozložení jedniček. Připomíná rozmístění proměnných v mapě Karnaugh, což není náhoda. Skutečně, tato mapa byla konstruována s využitím Grayova kódu.

Tab. 7.3 Vztah binárního a Grayova kódu

b_2	b_1	b_0	g_2	g_1	g_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

Výrazy pro převod jsou následující:

Binární \rightarrow Grayův

$$g_2 = b_2$$

$$g_1 = b_2 \oplus b_1$$

$$g_0 = b_1 \oplus b_0$$

Grayův \rightarrow Binární

$$b_2 = g_2$$

$$b_1 = g_2 \oplus g_1$$

$$b_0 = g_2 \oplus g_1 \oplus g_0$$

Rozšíření na větší počet proměnných je zřejmé.

7.4 Hledání hazardu z výrazu

Hazard může být nalezen i rozborem výrazu. Opět se předpokládá změna **jen v jedné** proměnné. Je třeba zjistit, zda při změnách některé proměnné a nepříznivých konstantních hodnotách ostatních proměnných je či není generován falešný výstupní impulz. Za tím účelem se funkce rozloží podle této proměnné (např. x). Tento rozklad je vždy možný.

$$y = x \cdot A + \bar{x} \cdot B + C$$

Členy A, B, C značí samostatné proměnné nebo logické výrazy. Ve všech se již nevyskytuje proměnná x , ale mohou se v nich vyskytovat kterékoliv ze zbylých proměnných. Při jejich konstantních hodnotách pak členy A nebo B nebo C nabývají hodnoty 0 nebo 1. O přítomnosti hazardu svědčí výraz $x + \bar{x}$, jak již bylo v předchozím textu ukázáno. Aby se původní výraz mohl převést na tento tvar, musí být $A = 1, B = 1, C = 0$. Vzniká tak **soustava tří rovnic**. Jestliže má řešení, hazard při změně proměnné x **vzniká**. Nemá-li řešení, hazard při změně proměnné x **nevzniká**. Řešení podmínek udává kombinaci hodnot ostatních proměnných, při které je možný hazard. Často je těchto řešení více. Celý postup lze opakovat pro všechny proměnné.

Soustava rovnic, kde A, B, C jsou logické výrazy:

$$A = 1$$

$$B = 1$$

$$C = 0$$

Všechny musí platit současně. Proto lze psát $A \cdot B \cdot \bar{C} = 1$. Tento součin se provede a případně zjednoduší jako každý jiný výraz. Má-li hodnotu 1 pro nějakou kombinaci hodnot proměnných, které se v něm nalézají, je tato kombinace hledaným **řešením soustavy** rovnic.

Následuje příklad výrazu, který by svědčil o hazardu:

$$z = \bar{b} \bar{d} + \bar{b} c + \bar{a} b$$

Hledáme možnost hazardu při $b \uparrow$. Rozložíme z podle této proměnné:

$$z = b(\bar{a}) + \bar{b}(c + \bar{d}) + 0$$

Zřejmě $A = \bar{a}, B = (c + \bar{d}), C = 0$. Proto

$$\bar{a} \cdot (c + \bar{d}) \cdot 1 = 1, \text{ nebo } \bar{a}c + \bar{a}\bar{d} = 1$$

To má dvě řešení:

$$a = 0, c = 1 \text{ nebo } a = 0, d = 0$$

Tedy v obvodu bude vznikat falešný impulz při $\bar{a}c\tilde{d}, b \uparrow$ (d libovolné) nebo při $\bar{a}\bar{c}\bar{d}, b \uparrow$.

Jiný příklad, ve kterém hazard při $a \uparrow$ nebude:

$$y = acd + bc + \bar{b}d + \bar{a}d$$

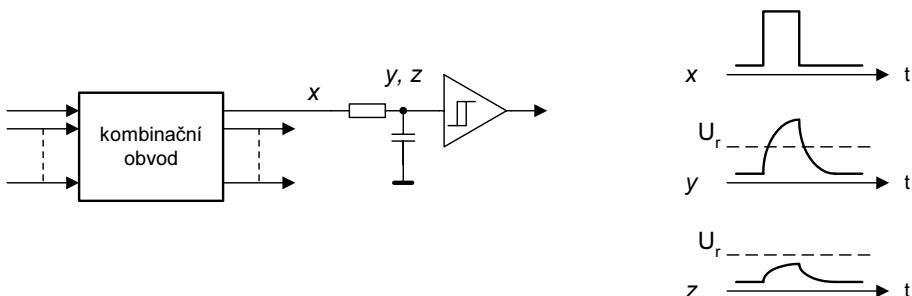
Stejný postup vede k podmínce $(cd)(d)(\overline{bc + \bar{bd}}) = 1$, neboli $cd \cdot \bar{b} \bar{d} + cd \cdot b\bar{c} + cd \cdot \bar{c}\bar{d} = 1$. Výraz na levé straně je roven nule pro jakoukoliv kombinaci hodnot proměnných b, c, d , a proto soustava nemá řešení. Hazard při $a\uparrow$ nehoří.

7.5 Hazard při změně více než jedné proměnné

Pokud se na vstupech kombinačního obvodu mění přibližně současně více proměnných a není známo přesné pořadí jejich změn, mohou mezi starým a novým ustáleným stavem existovat krátkodobé **mezistavy**, při kterých mohou na výstupech kombinačního obvodu být jiné hodnoty, než v obou ustálených stavech. Na př. pro dvě proměnné a a b funkce $y(a,b)$ se z kombinace 01 do kombinace 10 můžeme dostat takto: $01 \rightarrow 11 \rightarrow 10$ ($a\uparrow$ bylo dříve než $b\downarrow$) nebo $01 \rightarrow 00 \rightarrow 10$ ($b\downarrow$ bylo dříve než $a\uparrow$). Mohly se tedy vyskytnout všechny možné vstupní stavy. Jestliže funkční hodnota byla stejná (např. 1) v původním i novém stavu, pak alespoň v jednom mezistavu musí být opačná (tedy např. 0) – jinak by totiž nebyla závislá na žádné ze vstupních proměnných, což je spor. Krátkodobý falešný impulz lze tedy očekávat. V případě změny více než jedné vstupní proměnné nepomůže žádný redundantní obvod. Všeobecně u obvodů se změnou více než jedné vstupní proměnné je nutné falešný impulz předpokládat **vždy**. Je proto nutné využít jiné metody k jeho odstranění.

7.6 Potlačení falešného impulzu filtrem

Tato metoda je problematická. Spočívá ve vložení dolnofrekvenční propusti za kombinační obvod, takže krátké impulzy jsou potlačeny – nikoliv ale zcela odstraněny. Obr. 7.4 ukazuje princip.



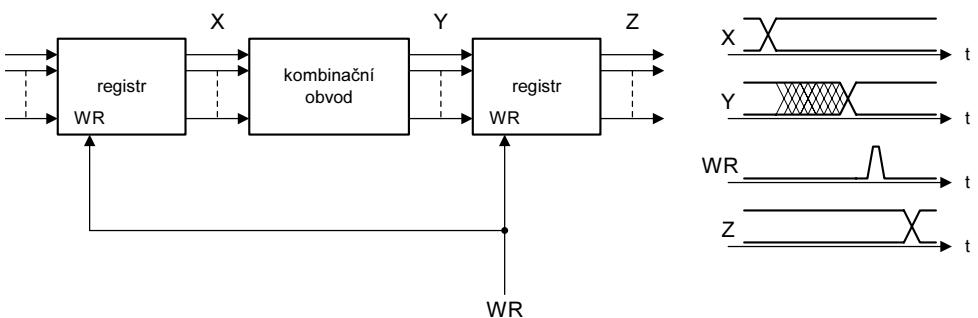
Obr. 7.4 Filtr za kombinačním obvodem

Falešný impulz na výstupu x kombinačního obvodu je jednoduchým filtrem vytvarován do průběhu y , který na obrázku zasahuje přes referenční úroveň následujícího člena. Pokud by časová konstanta RC filtru byla delší, dosáhlo by se ale průběhu z . Maximum napětí již nedosahuje referenční úrovně a proto následující člen již nebude na tento impulz reagovat. Zešikmené hrany signálu je vhodné vytvarovat členem s hysterezní charakteristikou. Dlouhé impulzy, odpovídající správným změnám signálu, však jsou filtrem rovněž zpožděny. Filtr proto **zpomaluje činnost** celého systému. Navíc nedává jistotu spolehlivého potlačení im-

pulzu za všech podmínek. Bude záležet na délce falešného impulzu vzhledem k časové konstantě, na úrovních U_L a U_H , na kolísání napájecího napětí, atd. Větší (ale ne úplnou) jistotu dá značné prodloužení časové konstanty, to ale neúměrně zpomalí celý systém. Tato metoda proto není vhodná.

7.7 Potlačení falešného impulzu registrum

Nejčastěji se falešné impulzy odstraňují pomocí **vyrovnávacích registrů**. Tomu ovšem musí být přizpůsobena struktura celého číslicového systému a způsob spolupráce jeho jednotlivých částí. Jednu z možností ukazuje obr. 7.5.



Obr. 7.5 Vložení registru na výstup kombinačního obvodu

Do registru lze zapsat vstupní signály působením zápisového impulzu WR. Zůstanou v něm zapamatovány až do dalšího zápisového impulzu. Registr je zařazen za kombinačním obvodem, takže jsou do něj zapsány výstupní signály Y. Jejich změny a případné falešné impulzy následují se zpožděním za změnami na vstupech kombinačního obvodu X (na obrázku znázorněno šrafováně). Pokud se zápisový impulz přiveze až po doznění přechodných dějů, budou se signály na vstupech registru Z měnit všechny současně a bez falešných impulzů. Nebezpečí hazardu je tak odstraněno.

Otzázkou je jen generace zápisových impulzů ve „správných“ okamžicích, tedy vždy po změně stavu na vstupech kombinačního obvodu. Nejsnadnější je zavést synchronizaci i do předcházejících a případně do následujících stupňů. Na obr. 7.5 je vyznačen kombinační obvod jako část systému vložená mezi registry. Synchronizační impulzy budou generovány centrálním blokem řízení a časování, a vložené kombinační obvody mezi registry budou zajišťovat zpracování dat. To je zcela standardní přístup k návrhu číslicových systémů.

7.8 Vliv úprav výrazu a obvodu na hazardy

V praxi je běžné, že výrazy se upravují pro získání vhodnějšího tvaru, který z nějakých důvodů lépe vyhovuje realizaci obvodu. Věnujme se otázce, zda po úpravách výrazu jsou ještě platné závěry o existenci hazardů, získané na základě původního výrazu. Jinými slovy – z výrazu byly vyvozeny závěry o existenci hazardu a podle výrazu byl sestaven obvod.

Závěry o hazardu pro něj platí. Pak se původní výraz upraví a podle něj se sestaví nový obvod. Ten bude mít sice jinou strukturu, ale v **ustálených** stavech se bude chovat stejně, jako obvod před úpravou (je s ním ekvivalentní, má i stejnou pravdivostní tabulkou). Otázkou ale je, zda v tomto obvodu budou existovat **stejné hazardy**, jako v obvodu **před úpravou**.

Skutečnost je taková, že některé úpravy skutečně znehodnocují původní závěry o hazardu. Takové úpravy budeme považovat za „hazard měnící“ na rozdíl od jiných úprav, které nevedou k jinému chování obvodu (úpravy „hazard neměnící“). Bez důkazu uvedeme, že úpravy pomocí **závorek** (vytýkáním před závorky či jejich roznásobením) a úpravy podle **DeMorgana** patří mezi hazard měnící. Naopak úpravy dosažené **spojováním** mintermů patří mezi hazard neměnící, jak již bylo v předchozích odstavcích ukázáno. V případě použití úprav, které mění chování obvodu v přechodných dějích, je nutné zapomenout na původní závěry a pro **upravený** obvod provést **nový rozbor**.

Na obr. 7.6 jsou dva ekvivalentní obvody. Obvod vlevo byl sestaven na základě funkce:

$$y(a, b, c, d) = \bar{b}cd + bcd + \bar{a}\bar{b}c$$

Obvod uprostřed vznikl po aplikaci De Morganových zákonů:

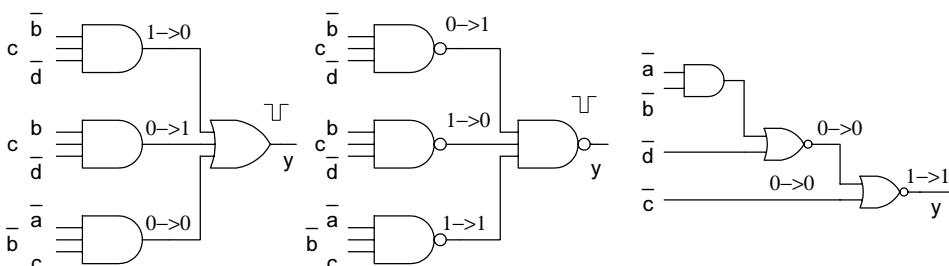
$$\bar{b}cd + bcd + \bar{a}\bar{b}c = \overline{\overline{\bar{b}cd}} \cdot \overline{\overline{bcd}} \cdot \overline{\overline{\bar{a}\bar{b}c}} \quad (\text{DeMorgan - neměnící hazard})$$

Obvod vpravo vznikl na základě upraveného prvního výrazu:

$$\bar{b}cd + bcd + \bar{a}\bar{b}c = \overline{cd} + \overline{\bar{a}\bar{b}c} \quad (!!! \text{ spojování - měnící hazard})$$

$$= c \cdot (\bar{d} + \bar{a}\bar{b}) = \overline{c} + \overline{\bar{d} + \bar{a}\bar{b}} \quad (\text{závorky a DeMorgan - neměnící hazard})$$

Na obrázcích je vyznačena situace při $a\bar{c}\bar{d}$, $b\uparrow$. V obvodu 7.6 vlevo hazard bude, v obvodu uprostřed bude, v obvodu napravo nebude, ačkoliv jsou všechny tři obvody ekvivalentní (ale jen v ustáleném stavu).

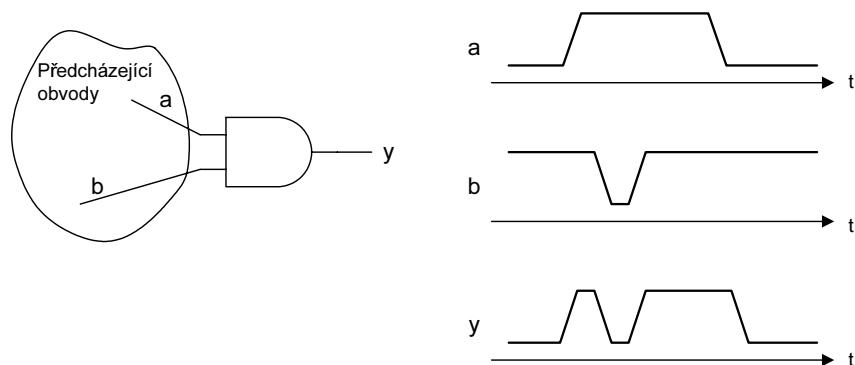


Obr. 7.6 Ekvivalentní úpravy výrazu a jejich vliv na hazard

7.9 Dynamický hazard

Dosud popisovaný hazard, projevující se jako falešný impulz namísto konstantní hodnoty, se nazývá **statický**. Vedle něj existuje ještě hazard **dynamický**. Ten se projevuje jako falešné zakmitnutí na náběžné nebo doběžné hraně signálu namísto jednoduché změny jeho

hodnoty. K tomuto zjevu dochází v obvodu, kde se v jednom místě spojují a zpracovávají signály, které mění stav a současně alespoň jeden další signál, zatížený statickým hazardem. Časový diagram ilustruje situaci.



Obr. 7.7 Dynamický hazard

Dynamický hazard je obdobně škodlivý, jako hazard statický. Může vést k omylům ve vyhodnocení počtu změn stavu (počítání hran) signálu. K jeho odstranění postačí odstranit hazard statický. Kdyby na obrázku průběh signálu b byl bez hazardu (ve stavu 1) během změny signálu a , pak součin $a \cdot b$ by nevykazoval zákmit.



ZÁKLADNÍ FUNKČNÍ BLOKY

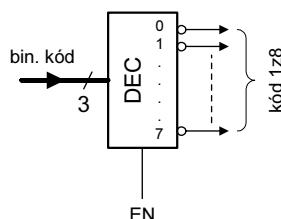
Při návrhu číslicových systémů se pracuje pokud možno s většími funkčními bloky. Mohou být implementovány jako integrované obvody, zpravidla střední integrace (MSI), nebo jako standardní bloky v počítačových návrhových programech. Zde budou postupně probrány ty základní a nejčastější.

8.1 Dekodér

Dekodér převádí binární kód na **kód IzN** . Při kódu IzN je vždy jen na jednom výstupu aktivní stav (předpokládejme, že aktivní stav je 0). Pozice tohoto aktivního výstupu odpovídá binárnímu číslu o k bitech, přiváděnému na vstup. Platí, že $N = 2^k$. V tab. 8.1 je pravdivostní tabulka dekodéru z tříbitového binárního kódu na kód $Iz8$ a na obr. 8.1 je jeho schematické znázornění.

Tab. 8.1 Pravdivostní tabulka dekodéru

Vstup			Výstup								
MSB	LSB		y ₇	y ₆	y ₅	y ₄	y ₃	y ₂	y ₁	y ₀	
x ₂	x ₁	x ₀	1	1	1	1	1	1	1	0	
0	0	0	1	1	1	1	1	1	1	0	
0	0	1	1	1	1	1	1	1	0	1	
0	1	0	1	1	1	1	1	0	1	1	
0	1	1	1	1	1	1	0	1	1	1	
1	0	0	1	1	1	0	1	1	1	1	
1	0	1	1	1	0	1	1	1	1	1	
1	1	0	1	0	1	1	1	1	1	1	
1	1	1	0	1	1	1	1	1	1	1	

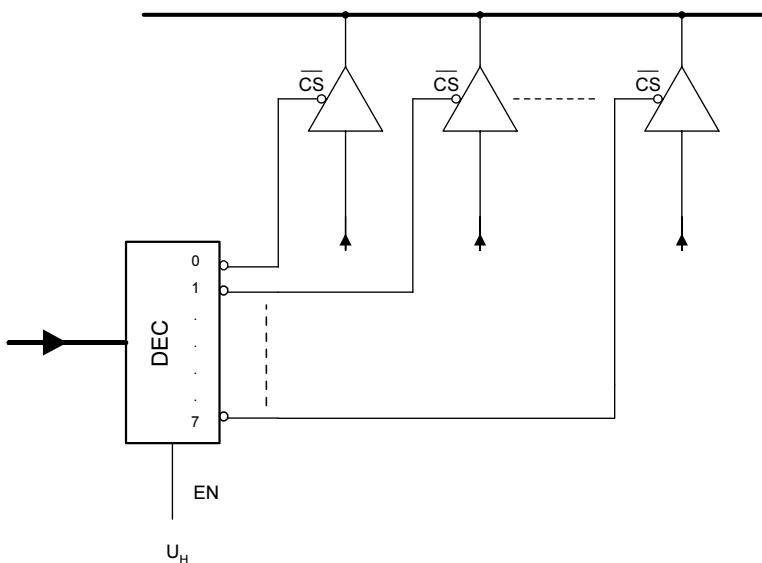


Obr. 8.1 Vstupy a výstupy dekodéru

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

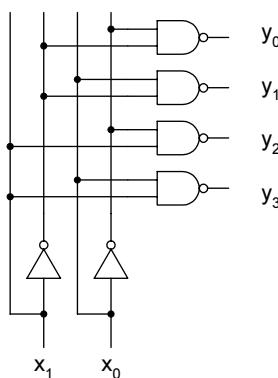
Dekodér se používá pro funkci výběru jednoho z několika obvodů. Velmi často se jedná o obvody třístavové, připojené na společnou sběrnici. Jejich výběrové vstupy jsou zpravidla

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



Obr. 8.2 Ovládání obvodů na sběrnici dekodérem

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



Obr. 8.3 Vnitřní zapojení dekodéru 1z4

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

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

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

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

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

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

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

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

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

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

```

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

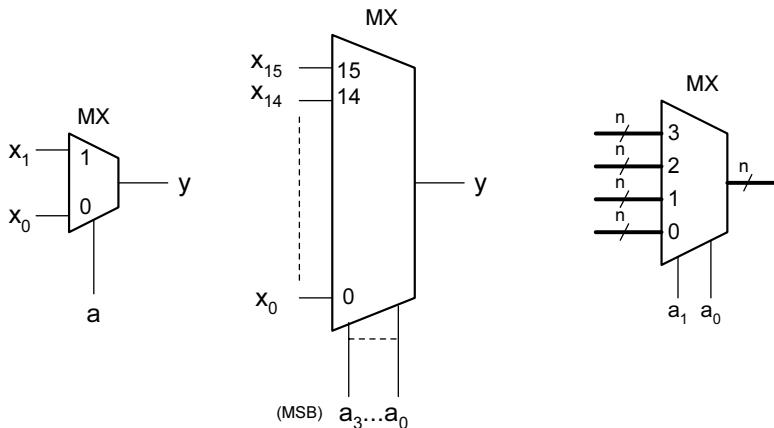
```

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

8.2 Multiplexor

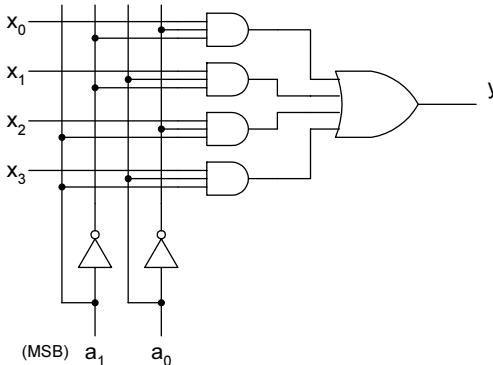
Multiplexor je vlastně číslicový **přepínač**. Má vstupy adresy (v binárním kódu) a vstupy přepínaných signálů. Ty jsou očíslované a čísla vyznačují, při které hodnotě adresy je daný vstupní signál převeden na výstup. Nejjednodušší je dvojkanálový multiplexor s pouze jedním adresovým signálem, který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

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



Obr. 8.4 Různé varianty multiplexorů

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



Obr. 8.5 Vnitřní zapojení čtyřkanálového multiplexoru

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

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

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

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

Výpis 8.3 Dvoukanálový multiplexor pomocí konstrukce **when-else**

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

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

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

```

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

```

Výpis 8.4 Šestnáctikanálový multiplexor pomocí procesu a indexace vektoru

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

```

library ieee;
use ieee.std_logic_1164.all;

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

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

```

Výpis 8.5 Skupinový čtyřkanálový multiplexor pomocí procesu a case

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

```

library ieee;
use ieee.std_logic_1164.all;

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

```

```

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

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

```

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

8.3 Demultiplexor

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

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

```

library ieee;
use ieee.std_logic_1164.all;

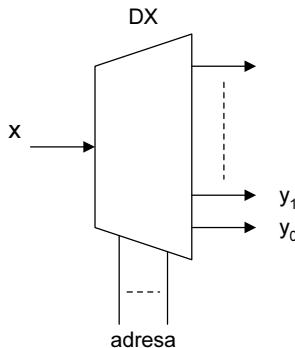
entity dmx1_2 is
  port (
    x : in std_logic;
    a : in std_logic;
    y0, y1 : out std_logic
  );
end dmx1_2;

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

```

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

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



Obr. 8.6 Vstupy a výstupy demultiplexoru

```

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

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

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

```

Výpis 8.8 Šestnáctikanálový demultiplexor pomocí procesu a indexace vektoru

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

```

library ieee;
use ieee.std_logic_1164.all;

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

```

```

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

```

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

8.4 Prioritní kodér

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

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

Tab. 8.2 a obr. 8.7 ukazuje příklad prioritního kodéru s 8 vstupy (x) v kódě $Iz8$ a třemi výstupy (y) v binárním kódu. Samostatný výstup z slouží k informování o tom, že je alespoň jeden vstupní signál aktivní – jsou-li totiž na všech vstupech hodnoty 0, nemá binární kód na výstupech $y_2y_1y_0$ smysl a hodnoty lze považovat za neurčené.

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

Pro jednotlivé výstupní proměnné platí vztahy:

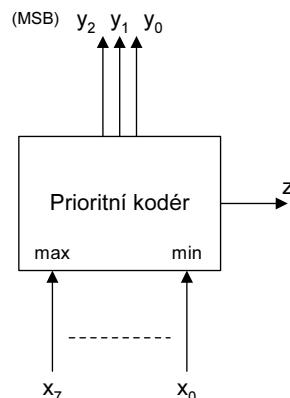
$$\begin{aligned}
 y_2 &= x_7 + \bar{x}_7x_6 + \bar{x}_7\bar{x}_6x_5 + \bar{x}_7\bar{x}_6\bar{x}_5x_4 \\
 y_1 &= x_7 + \bar{x}_7x_6 + \bar{x}_7\bar{x}_6\bar{x}_5\bar{x}_4x_3 + \bar{x}_7\bar{x}_6\bar{x}_5\bar{x}_4\bar{x}_3x_2 \\
 y_0 &= x_7 + \bar{x}_7\bar{x}_6x_5 + \bar{x}_7\bar{x}_6\bar{x}_5\bar{x}_4x_3 + \bar{x}_7\bar{x}_6\bar{x}_5\bar{x}_4\bar{x}_3\bar{x}_2x_1 \\
 z &= x_7 + x_6 + x_5 + x_4 + x_3 + x_2 + x_1 + x_0
 \end{aligned}$$

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

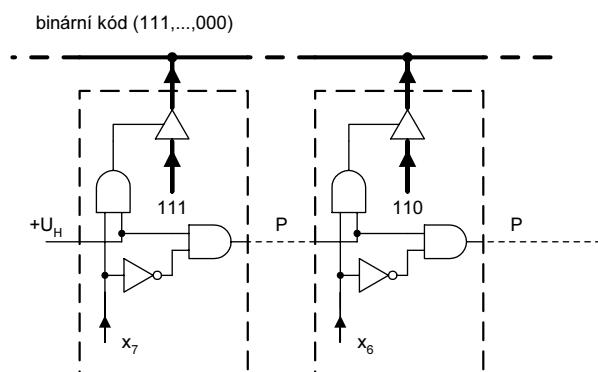
Výše uvedený prioritní kodér je „centralizovaný“ v tom smyslu, že všechny vstupní signály jsou svedeny na vstupy jednoho obvodu, a všechny výstupní signály vznikají v tomtéž obvodu. Prioritní kodér je ale možné sestavit ještě jinak, s využitím decentralizovaných obvodů – viz *obr. 8.8*.

Tab. 8.2 Pravdivostní tabulka prioritního kodéru

Vstupy								Výstupy (MSB)			
x ₇	x ₆	x ₅	x ₄	x ₃	x ₂	x ₁	x ₀	y ₂	y ₁	y ₀	z
1	-	-	-	-	-	-	-	1	1	1	1
0	1	-	-	-	-	-	-	1	1	0	1
0	0	1	-	-	-	-	-	1	0	1	1
0	0	0	1	-	-	-	-	1	0	0	1
0	0	0	0	1	-	-	-	0	1	1	1
0	0	0	0	0	1	-	-	0	1	0	1
0	0	0	0	0	0	1	-	0	0	1	1
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0	-	-	-	0



Obr. 8.7 Vstupy a výstupy prioritního kodéru



Obr. 8.8 Prioritní řetězec

Výběr vstupu podle priority zajišťuje **prioritní řetězec**. Jeho opakující se shodné části (orámované čárkovaně) jsou rozmištěny po jednotlivých modulech systému. Jejich spojení zprostředkují signály přenosu P. Je-li aktivní vstupní signál s vyšší prioritou (např. x_7), jsou všechny přenosové signály napravo od něho nulové a průchod všech signálů x_6, x_5 , atd. je blokován. Proto je jen u jednoho modulu s aktivním vstupním signálem vybuzen třístavový člen, přes který je na společnou sběrnici přivedena kódová kombinace příslušející danému modulu. Modul s nejvyšší prioritou má na vstupu přenosového signálu trvale stav 1. Funkce kodéru s prioritním řetězcem je tedy shodná s funkcí dekodéru centralizovaného. Rozdíl je v konstrukčním řešení – druhá varianta se hodí pro **stavebnicové uspořádání** systému. Priority jednotlivých modulů jsou dány pořadím jejich propojení v řetězci.

Následující výpis kódu ukazuje realizaci výše popsaného prioritního kodéru se vstupy v kódu 1 z 8 a výstupy v binárním kódu v jazyce VHDL pomocí přiřazení s konstrukcí **when-else**.

```

library ieee;
use ieee.std_logic_1164.all;

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

architecture a_prior_kod of prior_kod is
begin
  -- kod 000 dava když je x(0) = 1 nebo zadny vstup x není v log. 1
  y <= "111" when x(7) = '1' else
    "110" when x(6) = '1' else
    "101" when x(5) = '1' else
    "100" when x(4) = '1' else
    "011" when x(3) = '1' else
    "010" when x(2) = '1' else
    "001" when x(1) = '1' else
    "000";
  -- signalizace aktivního vstupu
  z <= '0' when x = "00000000" else '1';
end a_prior_kod;

```

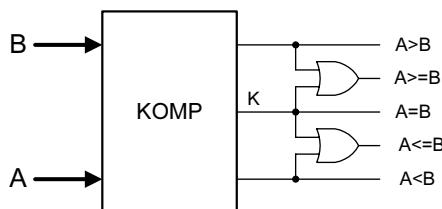
Výpis 8.10 Prioritní kodér pomocí přiřazení s konstrukcí **when-else**

8.5 Číslicový komparátor

Číslicový komparátor porovnává velikost dvou čísel (v binárním kódu). Nejčastěji je využívána informace o **shodě dvou čísel**. Ta se získá velmi jednoduše tak, že se porovnávají dvojice bitů na stejně pozici. Označíme-li jedno číslo A a jeho bity A_i , a druhé B a jeho bity B_i , pak pro výstup K platí:

$$K = (a_{n-1} \equiv b_{n-1})(a_{n-2} \equiv b_{n-2}) \dots (a_0 \equiv b_0)$$

Musí platit současně ekvivalence na pozici všech bitů čísel. Existují i komparátory, vyhodnocující na dalších dvou výstupech nerovnosti $A > B$ a $A < B$. Spolu s výstupem pro $A = B$ od nich lze odvodit i informaci o $A \geq B$ nebo $A \leq B$, jak je naznačeno na obr. 8.9.



Obr. 8.9 Číslicový komparátor

Následující výpis kódu ukazuje realizaci výše popsaného komparátoru pro 4bitové operandy bez znaménka v jazyce VHDL.

```

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

entity komparator is
    port (
        a : in unsigned (3 downto 0);
        b : in unsigned (3 downto 0);
        a_v_b : out std_logic;
        a_vr_b : out std_logic;
        a_r_b : out std_logic;
        a_mr_b : out std_logic;
        a_m_b : out std_logic
    );
end komparator;

architecture a_komp of komparator is
    signal v, r, m : std_logic;
begin
    v <= '1' when a > b else '0';
    r <= '1' when a = b else '0';
    m <= '1' when a < b else '0';
    a_v_b <= v; -- výstup a > b
    a_r_b <= r; -- výstup a = b
    a_m_b <= m; -- výstup a < b
    a_vr_b <= v or r; -- výstup a >= b
    a_mr_b <= m or r; -- výstup a <= b
end a_komp;

```

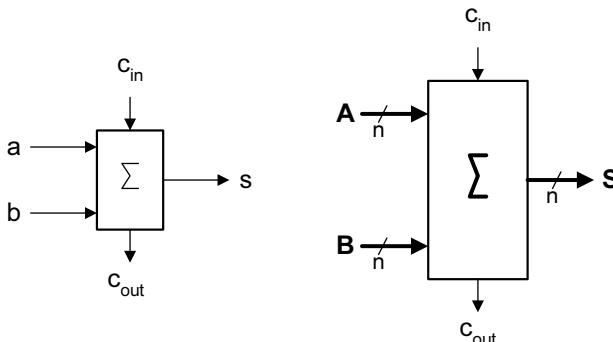
Výpis 8.11 Komparátor pomocí konstrukce ***when-else*** a rovnic

8.6 Sčítáčka

Sčítáčka provádí aritmetický součet dvou čísel. Pro jednobitovou sčítáku se vstupy scítaných čísel a a b , vstupem přenosu z nižšího řádu c_{in} , výstupem **aritmetického** součtu s a výstupem přenosu do vyššího řádu c_{out} platí tab. 8.3.

Tab. 8.3 Tabulka jednobitové sčítáčky

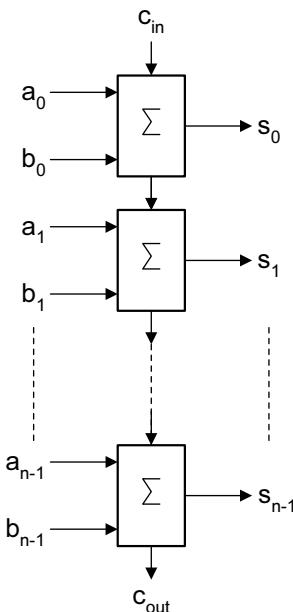
a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



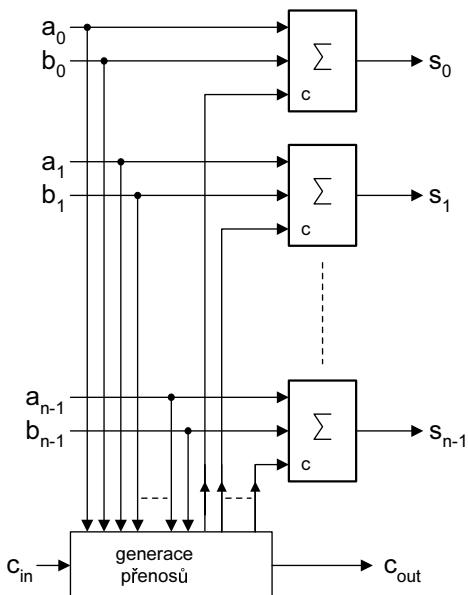
Obr. 8.10 Sčítáčka jednobitová (vlevo) a vícebitová (vpravo)

Vícebitová sčítáčka může být sestavena z jednobitových sčítáček. Přenosy mezi jednobitovými sčítáčkami jsou realizovány uvnitř celé sčítáčky a ven je vyveden jen přenos do nejvyššího bitu a přenos z nejnižšího bitu. Tak lze i s těmito vícebitovými sčítáčkami pracovat jako se stavebními bloky. Vstup a výstup přenosu jsou ale užitečné i jinak. Vstup c_{in} je v běžné funkci sčítáčky uzemněn, ale při stavu jedna vlastně zvyšuje hodnotu čísla S o jedničku – tak se provádí operace „inkrementace čísla o jedničku“. Výstup c_{out} může informovat o přetečení čísla, což je situace, kdy součet dvou n -bitových čísel je tak velký, že jej nelze vyjádřit n -bitovým výsledkem. Potřebný bit $n+1$ ale sčítáčka nemá a proto je výsledné číslo S chybné – schází mu nejvyšší bit.

Vnitřní zapojení sčítáčky složené z jednobitových sčítáček ukazuje obr. 8.11. Důležité je **zpoždění výstupu S** po změnách čísel A a B . Podstatná přitom není cesta od vstupů a_i nebo b_i na výstupy c_i , ale zpoždění celého **řetězce přenosů**. Ten je dlouhý, prochází všemi dílčími sčítáčkami a samozřejmě ovlivňuje všechny byty výsledku (obr. 8.11). Proto čím více bitů má sčítáčka této konstrukce, tím větší bude mít zpoždění. To je samozřejmě podstatná nevýhoda.



Obr. 8.11 Přenosový řetězec



Obr. 8.12 Zrychlení přenosů

Rychlé sčítáčky jsou proto konstruovány jinak. Přenosy nejsou propojeny do řetězce, kterým postupně procházejí, nýbrž jsou generovány **současně** pro všechny stupně obvodem, který zpracovává všechny vstupní bity a_i a b_i . Sčítáčka se pak nazývá „sčítáčka se zrychlením přenosů“ a je podstatně rychlejší, ale i složitější (obr. 8.12).

Následující výpis ukazuje realizaci parametrizovatelné sčítáčky (s volitelnou bitovou šířkou) s výstupem pro přenos v jazyce VHDL. Sčítáčka je realizována pomocí operátoru „+“ nad typem **unsigned**. Parametritzovatelnost je zajištěna pomocí konstruktu **generic**.

```

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

entity add is
    generic (
        width : positive := 8 -- sirka slova scitancu
    );
    port (
        a : in std_logic_vector (width - 1 downto 0);      -- 1. scitanec
        b : in std_logic_vector (width - 1 downto 0);      -- 2. scitanec
        s : out std_logic_vector (width - 1 downto 0);      -- vysledny soucet
        cout : out std_logic      -- vystup prenosu (carry out)
    );
end add;

architecture behav of add is
    signal a_uns, b_uns, s_uns : unsigned (width downto 0);
begin
    -- typova konverze na unsigned a rozsireni o 1 bit kvuli vystupu prenosu
    a_uns <= unsigned ('0' & a);
    b_uns <= unsigned ('0' & b);

    -- soucet behavioralne
    s_uns <= a_uns + b_uns;

    -- zpetna typova konverze na std_logic_vector
    s <= std_logic_vector (s_uns(width - 1 downto 0));

    -- vystupni prenos
    cout <= s_uns(width);
end behav;

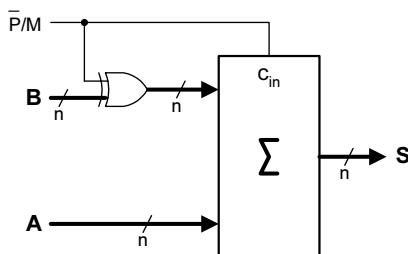
```

*Výpis 8.12 Sčítáčka s výstupem pro přenos realizovaná pomocí typu **unsigned** a operátoru +*

8.7 Odčítáčka

Odečítání čísla se řeší jako přičítání tohoto čísla s opačným znaménkem. Pokud jsou záporná čísla vyjádřena ve **dvojkovém doplňku**, je obrácení znaménka jednoduché – znamená negaci všech bitů a následné přičtení jedničky. *Obr. 8.13* ukazuje obvodové řešení, umožňující sčítání i odčítání dvou čísel.

Vstupem \bar{P}/M se volí operace $A+B$ (stavem 0) nebo $A-B$ (stavem 1). Skupina n členů XOR pracuje jako řízená negace. Při $\bar{P}/M = 1$ je B negováno a současně je vstup přenosu $c_{in} = 1$. Tím je přičtena jednička a je tak obráceno znaménko B . Při $\bar{P}/M = 0$ není B negováno a na vstupu přenosu je stav 0.



Obr. 8.13 Sčítacka a odčítacka čísel ve dvojkovém doplňku

Následující výpis ukazuje realizaci parametrizovatelné přepínatelné sčítácky/odčítácky v jazyce VHDL pomocí řízené negace druhého operandu a sčítácky.

```

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

entity add_sub is
  generic (
    width : positive := 8    -- sirka slova operandu
  );
  port (
    a : in std_logic_vector (width - 1 downto 0);    -- 1. operand
    b : in std_logic_vector (width - 1 downto 0);    -- 2. operand
    pm : in std_logic;      -- prepínac '0' - add / '1' - sub
    s : out std_logic_vector (width - 1 downto 0)    -- vysledek
  );
end add_sub;

architecture prep_invert_add of add_sub is
  signal a_uns, b_uns, cin_uns, r_uns : unsigned (width - 1 downto 0);
begin
  -- typova konverze na unsigned
  a_uns <= unsigned (a);

  -- rizene negovani druheho op. ('0' - buffer / '1' - invert)
  neg: for i in 0 to width - 1 generate
    b_uns(i) <= b(i) xor pm;
  end generate neg;

  -- vstupni prenos '0' - scitani / '1' - odcitani
  cin_uns(0) <= pm;
  cin_uns(width - 1 downto 1) <= (others => '0');

  -- soucet
  r_uns <= a_uns + b_uns + cin_uns;

  -- zpetna typova konverze na std_logic
  s <= std_logic_vector (r_uns);
end prep_invert_add;

```

Výpis 8.13 Přepínatelná sčítácka/odčítácka pomocí řízené negace a operátoru +

Následující výpis ukazuje rovněž realizaci parametrizovatelné přepínatelné sčítáčky/odčítáčky pro čísla se znaménkem. Přepínatelná sčítáčka/odčítáčka je však realizována pomocí sčítáčky a odčítáčky, jejichž výsledek se přepíná na výstup.

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

entity add_sub is
    generic (
        width : positive := 8      -- sirka slova operandu
    );
    port (
        a : in std_logic_vector (width - 1 downto 0);      -- 1. operand
        b : in std_logic_vector (width - 1 downto 0);      -- 2. operand
        pm : in std_logic;        -- prepinac '0' - add / '1' - sub
        s : out std_logic_vector (width - 1 downto 0)      -- vysledek
    );
end add_sub;

architecture prep_add_sub of add_sub is
    signal a_uns, b_uns, s_uns : unsigned (width - 1 downto 0);
begin
    -- typova konverze na unsigned
    a_uns <= unsigned (a);
    b_uns <= unsigned (b);

    -- behavioralni vypocet souctu / rozdilu
    s_uns <= a_uns + b_uns when pm = '0' else a_uns - b_uns;

    -- typova konverze na std_logic
    s <= std_logic_vector (s_uns);
end prep_add_sub;
```

Výpis 8.14 Přepínatelná sčítáčka/odčítáčka

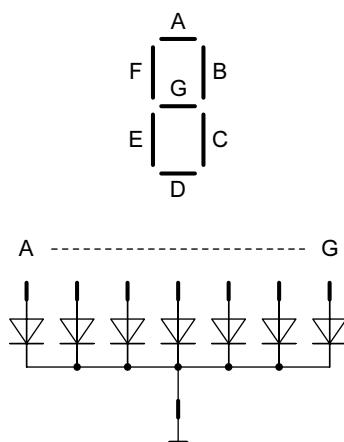
8.8 Převodník kódu

K převodu binárního kódu na kód *IzN* existuje obvod s názvem „decodér“. Pro převod mezi jinými kódy se používá obecný název „převodník kódu“. Řeší se případ od případu jako obecný kombinační obvod s několika vstupy a několika výstupy, sestaví se pravidlostní tabulka s maximálním využitím všech zjednodušení, použije se skupinová minimalizace a další potřebné úpravy funkcí.

Velmi často je třeba zobrazit dekadická čísla pomocí segmentovek s LED diodami. Jelikož je třeba zobrazit každou dekadickou číslici zvlášť, nepracuje se v binárním kódu, nýbrž v **kódu BCD** (binárně kódovaná dekadická číslice). Tento kód se pro čísla 0 až 9 shoduje s kódem binárním o čtyřech bitech, ale kombinace příslušející číslům 10 až 15 v něm neexistují. Segmentovky s LED diodami mají běžně 7 segmentů, rozmištěných podle *obr. 8.14*.

Tab. 8.4 Převod kódu BCD na 7segmentový

DEK	x_3	x_2	x_1	x_0	A	B	C	D	E	F	G
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1



Obr. 8.14 Zapojení jednotlivých segmentů

Převodník kódu bude mít vstupní proměnné x_3, x_2, x_1, x_0 (= LSB), vyjadřující číslo v kódu BCD, a výstupní proměnné A až G pro ovládání jednotlivých segmentů. Předpokládáme zapojení se společnými katodami, takže k rozsvícení segmentu slouží stav 1. Pravdivostní tab. 8.4 dává úplnou informaci. Vzhledem k malému počtu vstupních proměnných lze minimalizaci ještě provést v mapách. Existují i segmentovky v zapojení se společnou anodou – pak by sloupce hodnot A až G byly negované.

Převodník kódu BCD/7seg je vzácným případem, kdy hazardy v kombinačním obvodu nejsou významné.

Následující výpis kódu ukazuje realizaci převodníku BCD kódu na 7segmentový kód. Převodník kódu využívá konstrukci procesu s **case** (realizuje pravdivostní tabulkou).

```

-- Zobrazeni BCD na sedmsegmentovy displej
--   -a-
-- f|   |b
--   -g-
-- e|   |c
--   -d-
--
--      bit: 6 5 4 3 2 1 0
-- segment: a b c d e f g

library ieee;
use ieee.std_logic_1164.all;

entity segment is
  port (
    data : in std_logic_vector (3 downto 0);
    seg : out std_logic_vector (6 downto 0)
  );
end segment;

architecture a_segment of segment is
begin
  process (data) begin
    case data is
      when "0000" => seg <= "1111110"; -- 0
      when "0001" => seg <= "0110000"; -- 1
      when "0010" => seg <= "1101101"; -- 2
      when "0011" => seg <= "1111001"; -- 3
      when "0100" => seg <= "0110011"; -- 4
      when "0101" => seg <= "1011011"; -- 5
      when "0110" => seg <= "1011111"; -- 6
      when "0111" => seg <= "1110000"; -- 7
      when "1000" => seg <= "1111111"; -- 8
      when "1001" => seg <= "1111011"; -- 9
      when others => seg <= "0000000"; -- vsechny led zhasnute
    end case;
  end process;
end a_segment;

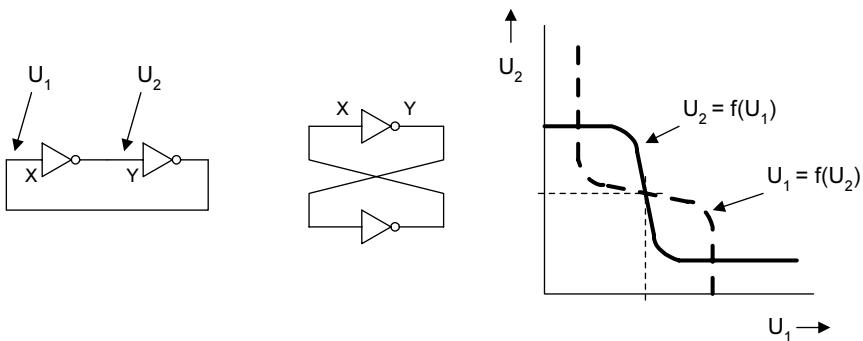
```

Výpis 8.15 Převodník kódů BCD na 7segmentový kód

8.9 Asynchronní klopné obvody

Klopné obvody jsou nejjednodušší sekvenční součástky. Asynchronní klopné obvody reagují na změny vstupních signálů okamžitě, synchronní klopné obvody vždy až působením synchronizačního (hodinového) impulzu.

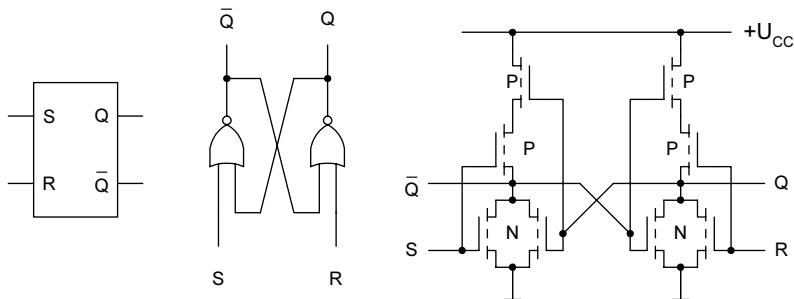
Klopný obvod vznikne spojením dvou negujících logických členů do okruhu. Je tak vytvořena zpětnovazební smyčka s celkovým fázovým posunem 360° , takže se jedná o zpětnou vazbu kladnou. K rozkmitání v tomto případě nedojde, neboť zpětná vazba udržuje obvod ve dvou možných stabilních stavech. Obvod se nazývá **bistabilní**. Obr. 8.15 vlevo ukazuje spojení dvou členů do kruhu, obrázek uprostřed totéž spojení, jen překreslené do běžnějšího tvaru. Obrázek vpravo ukazuje vliv zpětné vazby.



Obr. 8.15 Princip klopného obvodu

Napětí U₁ v bodě X vyvolá napětí U₂ v bodě Y. Vzájemný vztah je dán převodní charakteristikou U₂ = f(U₁), znázorněnou plnou tlustou čarou. Naopak napětí U₂ v bodě Y vyvolá napětí U₁ v bodě X. Vzájemný vztah je dán převodní charakteristikou U₁ = f(U₂), znázorněnou čárkovánou tlustou čarou. Při vychýlení napětí U₁ nebo U₂ pěs bod, vyznačený čárkovánými souřadnicemi, se obvod překlopí. Pokud vychýlení nedosahuje do tohoto bodu, obvod zůstává v původním stavu. V uvedené jednoduché verzi je pro překlopení obvodu zapotřebí přivést na vstup X nebo Y potřebné napětí. Každý vstup je ale propojen s výstupem členu předcházejícího, takže napětí pro překlopení je vnučeno i na výstup předcházejícího členu. Proto je pro překlopení potřebný i proud, i když jen jako krátký impulz. Toto řešení klopného obvodu je běžně využíváno u statických polovodičových pamětí.

K překlápení klopného obvodu lze využít i další vstupy logických členů, jak je ukázáno na obr. 8.16 pro případ členů NOR – vlevo je schematická značka, uprostřed zapojení s logickými členy, vpravo detaily zapojení v technologii CMOS.



Obr. 8.16 Klopny obvod se členy NOR

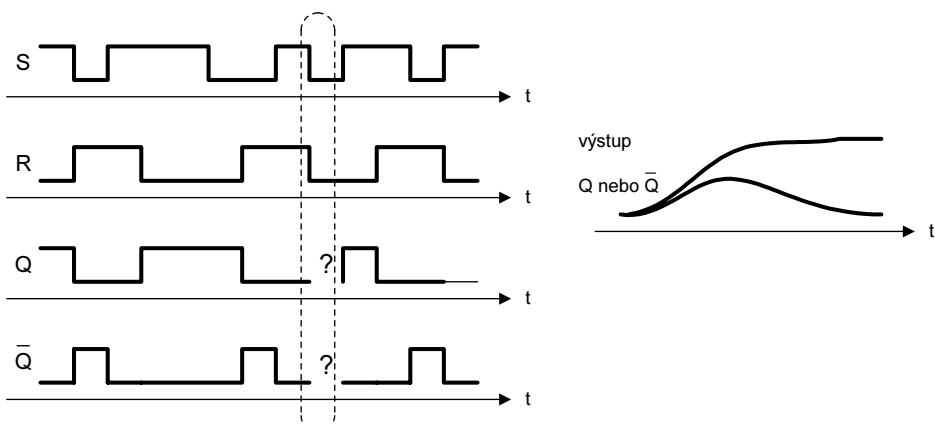
Pokud je na obou vstupech, označených jako R a S, stav 0, není průchod signálu zpětnovazební smyčkou nijak blokován a obvod bude trvale v jednom ze dvou stabilních stavů. Při S = 1 je vždy $\bar{Q} = 0$ a je-li přitom R = 0, bude Q = 1. Vzhledem k symetrii obvodu je při opačné situaci na vstupech (tj. S = 0 a R = 1) na výstupech $Q = 0$ a $\bar{Q} = 1$. Platí pravdivostní tab. 8.5.

Tab. 8.5 Pravdivostní tabulka klopného obvodu RS se členy NOR

S	R	Q	\bar{Q}
0	0	Q	\bar{Q}
0	1	0	1
1	0	1	0
1	1	0	0

První řádek dokládá **paměťovou** vlastnost klopného obvodu. Je třeba mu rozumět tak, že na výstupu Q zůstává stále tatáž hodnota, stejně tak i na výstupu \bar{Q} . Druhý řádek znamená **vynulování** výstupu Q , třetí řádek **nastavení** výstupu (na stav 1). Poslední řádek vyžaduje komentář. Ze schématu na obr. 8.16 je zřejmé, že při $R = S = 1$ musí být na obou výstupech stav 0. To sice protirečí označení výstupů (neboť Q nemůže být rovno \bar{Q}), ale není to podstatné, neboť přejmenováním signálů by byl problém vyřešen. Problém však působí **přechod** ze vstupního stavu $R = S = 1$ do stavu $R = S = 0$. Při $R = S = 1$ je klopný obvod uveden do symetrického stavu, když oba jeho členy mají na vstupech i výstupech stejné napětí. Po **současné** změně R i S na 0 se plně projeví kladná zpětná vazba a překlopení do jednoho nebo druhého stavu je výsledkem náhodné nesymetrie obou členů. Vstupní stav $R = S = 1$ je někdy považován za zakázaný stav, což není správné. Zakázaný je přechod z $R = S = 1$ do $R = S = 0$. Ostatní přechody (z $R = S = 1$ do $R = 1, S = 0$ nebo z $R = S = 1$ do $R = 0, S = 1$) jsou zcela bez problémů. Výše popsaný klopný obvod se nazývá **obvod RS** (angl. Reset – nulování, Set – nastavení) a je definován v normě IEEE, Standard 91-1984 (Explanation of Logic Symbols), jako „**Case 5**“.

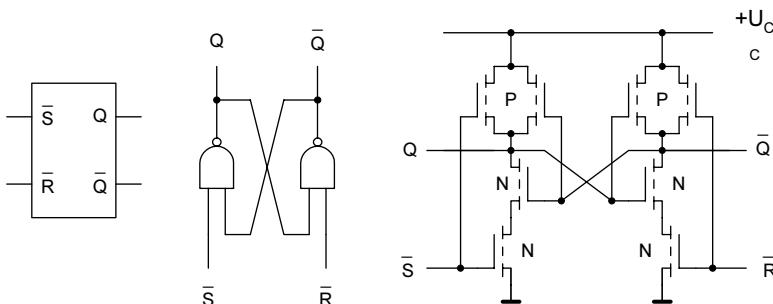
Obr. 8.17 ukazuje reakci obvodu RS na vstupní signály R a S.



Obr. 8.17 Reakce klopného obvodu RS na vstupní signály R a S

Jen v jednom okamžiku není stav na výstupech definován. Stav na výstupech bude záviset na **symetrii** obvodu a na nepatrých **časových posuvech** (rádově nanosekund) mezi dobežnými hranami signálů S a R . Dva z mnoha možných průběhů jsou na obrázku vpravo. Stav, který je vyvolán částečným vychýlením klopného obvodu a vede na pomalý průběh s nepředvídatelným výsledkem, se nazývá **metastabilní** stav. Je naprostě nežádoucí. Lze mu čelit jedině správným časováním všech vstupních signálů klopného obvodu.

Klopný obvod RS lze sestavit i z členů NAND, jak ukazuje obr. 8.18.



Obr. 8.18 Klopný obvod RS se členy NAND

Pokud je na obou vstupech, označených jako \bar{R} a \bar{S} , stav 1, není průchod signálu zpětnovazební smyčkou nijak blokován a obvod bude trvale v jednom ze dvou stabilních stavů – obvod si pamatuje stav. Při $\bar{S} = 0$ je vždy $Q = 1$ a je-li přitom $\bar{R} = 1$, bude $Q = 0$. Vzhledem k symetrii obvodu je při opačné situaci na vstupech (tj. $\bar{S} = 1$ a $\bar{R} = 0$) na výstupech $Q = 0$ a $\bar{Q} = 1$. Platí pravdivostní tab. 8.6.

Tab. 8.6 Pravdivostní tabulka klopného obvodu RS se členy NAND

\bar{S}	\bar{R}	Q	\bar{Q}
0	0	1	1
1	0	0	1
0	1	1	0
1	1	Q	\bar{Q}

I zde dochází k nejistotě při současné změně stavu na obou vstupech, a to z $\bar{S} = \bar{R} = 0$ do $\bar{S} = \bar{R} = 1$. Tento přechod je **zakázaný**. Žádný vstupní stav však sám o sobě zakázany není.

Tabulka, platná pro klopný obvod se členy NAND, není definována ve Standardu 91, je tam však definována tabulka („Case 4“), od které se liší jen negací sloupců pro S a R . Norma „IEEE Standard 91“ popisuje sice ještě další typy klopných obvodů RS, ty jsou však daleko méně časté, než obvody se dvěma členy NOR či NAND, popsané výše.

Následující výpis kódu ukazuje realizaci klopného obvodu RS v jazyce VHDL pomocí logického operátoru NOR.

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

entity rs_nor_err is
    port (
        r, s : in std_logic;
        q, n_q : buffer std_logic
    );
end rs_nor_err;

architecture struct of rs_nor_err is
begin
    q <= r nor n_q;
    n_q <= s nor q;
end struct;
```

Výpis 8.16 Klopny obvod RS realizovaný operátorem NOR (vznik smyčky – chybná realizace)

Rovnice pro Q a N_Q jsou však sestaveny tak, že dochází ke vzniku smyčky mezi signály Q a N_Q (N_Q je v rovnici pro Q a Q je v rovnici pro N_Q). V případě vzniku smyčky mezi signály dojde při simulaci takového kódu k zacyklení simulátoru. Z tohoto důvodu je v dalším výpisu kódu rovnice pro signál Q sestavena tak, aby se v ní signál N_Q vůbec nevyskytoval. Podobně je sestavena rovnice pro signál N_Q .

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

entity rs_nor is
    port (
        r, s : in std_logic;
        q, n_q : buffer std_logic
    );
end rs_nor;

architecture struct of rs_nor is
begin
    q <= r nor (s nor q);
    n_q <= s nor (r nor n_q);
end struct;
```

Výpis 8.17 Klopny obvod RS realizovaný operátorem NOR

Druhý výpis ukazuje realizaci klopného obvodu RS pomocí logické funkce NAND. Rovnice pro signály Q a N_Q jsou opět napsány tak, aby nedošlo ke vzniku smyčky.

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

entity rs_nand is
    port (
        n_r, n_s : in std_logic;
```

```

    q, n_q : buffer std_logic
);
end rs_nand;

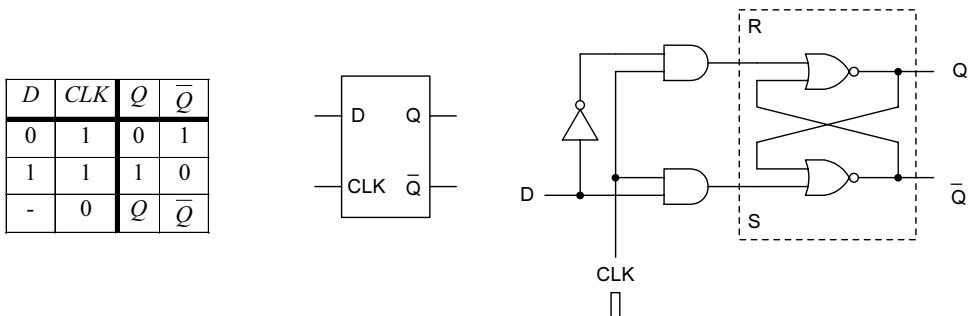
architecture struct of rs_nand is
begin
    q <= n_s nand (n_r nand q);
    n_q <= n_r nand (n_s nand n_q);
end struct;

```

Výpis 8.18 Klopny obvod RS realizovaný logickým operátorem NAND

8.10 Synchronní klopné obvody

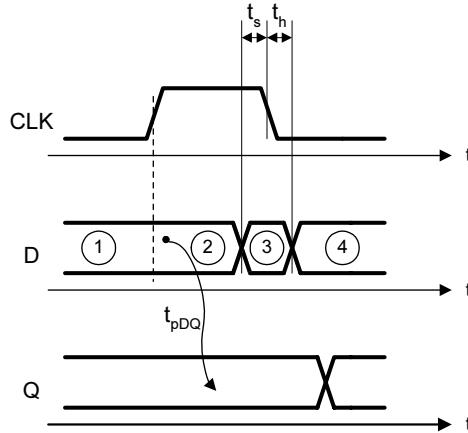
Klopny obvod D (viz obr. 8.19) je na rozdíl od předešlých obvodů RS **synchronní**, neboť mění stav jen v součinnosti vstupního signálu (D) a synchronizačního či **hodinového impulzu** (CLK). Během trvání hodinového impulzu ($CLK = 1$) je $S = D$ a $R = \bar{D}$, a tedy $Q = D$. Po ukončení hodinového impulzu, kdy $CLK = 0$, je $R = S = 0$, a to je kombinace hodnot, při které klopny obvod RS s členy NOR nemění stav. Stav, dosažený během hodinového impulzu, tak zůstane zachycen. Klopny obvod s tímto způsobem řízení se nazývá **úrovni řízený** (angl. **latch** – „západka“).



Obr. 8.19 Pravdivostní tabulka, schematická značka a zapojení klopného obvodu řízeného úrovní

Symbol X v tabulce značí libovolnou hodnotu 0 nebo 1. Na obr. 8.20 je časový diagram, platný pro všechny klopny obvody řízené úrovní. Jsou to nejen obvody D, ale (spíše výjimečně) i RS.

V oblasti 1 nemají změny vstupu vliv, v oblasti 2 (po náběhu CLK) se přenášejí na výstup se zpožděním t_{pDQ} , v oblasti 3 jsou změny zakázány a v oblasti 4 opět nemají změny vstupu vliv. Důležité jsou doby t_s – **doba předstihu** (angl. setup time) a t_h – **doba přesahu** (angl. hold time). Tyto doby vymezují oblast, ve které se vstupní stav nesmí měnit, jinak může dojít k vyvolání **metastabilního stavu**. Při dodržení doby předstihu a přesahu je zaručeno, že poslední změnou na vstupu bude rádně ovlivněn výstup, který pak zůstane konstantní až do dalšího hodinového impulzu.



Obr. 8.20 Časový diagram pro klopné obvody řízené úrovní

Základní vlastnosti klopného obvodu řízeného úrovní jsou:

- Po dobu trvání hodinového impulzu je obvod volně průchozí (transparentní) pro vstupní signál.
- Po ukončení hodinového impulzu zůstane poslední stav na výstupu uchován.
- Vstupní signál se nesmí měnit v oblasti kolem doběžné hrany hodinového impulzu.

Následující výpis kódu ukazuje realizaci klopného obvodu typu Latch pomocí procesu. Proces má v citlivostním seznamu uvedeny signály *d* a *en*. Proces je tedy spouštěn kdykoliv se změní signál *d* nebo *en*, přičemž do *q* se zapisuje pokud je signál *en* roven log. 1.

```

library ieee;
use ieee.std_logic_1164.all;

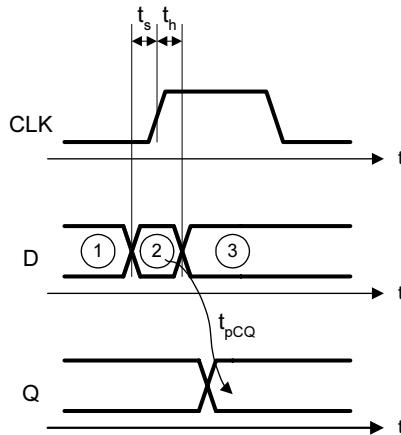
entity latch_pr is
  port (
    d : in std_logic;
    en : in std_logic;
    q : out std_logic
  );
end latch_pr;

architecture struct of latch_pr is
begin
  -- proces se spusti pri zmene signalu d nebo en
  process (d, en)
  begin
    -- do q se zapisuje d dokud je en rovno log. 1
    if en = '1' then
      q <= d;
    end if;
  end process;
end struct;

```

Výpis 8.19 Klopný obvod typu Latch realizovaný pomocí procesu

Další způsob řízení klopného obvodu je **řízení hranou** hodinového impulzu. Časový diagram ukazuje obr. 8.21.

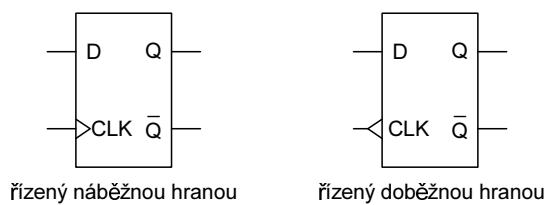


Obr. 8.21 Časový diagram klopného obvodu řízeného hranou hodinového impulzu

Změny na vstupu v oblastech 1 a 3 se na výstupu neprojevují. Stav na vstupu v oblasti 2 se přenese na výstup se zpožděním t_{pCQ} po náběžné hraně CLK. Stav na vstupu se nesmí měnit po dobu od t_s po t_h od náběžné hraně CLK, jinak může být vyvolán **metastabilní stav**.

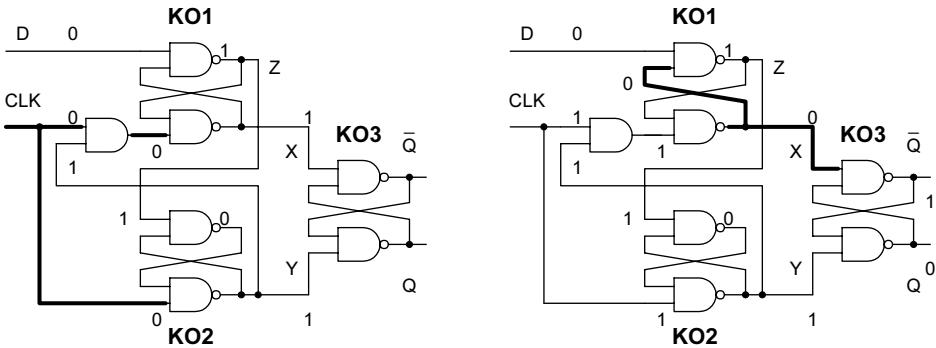
V pravidlostní tabulce klopného obvodu řízeného náběžnou hranou je tato hranou označena šipkou \uparrow , ve schematické značce šipkou dovnitř obvodu. Vedle řízení náběžnou hranou existuje i řízení doběžnou hranou. Tabulka i časové diagramy jsou pro ně platné po záměně CLK za \overline{CLK} a \uparrow za \downarrow .

D	CLK	Q	\bar{Q}
0	\uparrow	0	1
1	\uparrow	1	0
-	0	Q	\bar{Q}
-	1	Q	\bar{Q}
-	\downarrow	Q	\bar{Q}



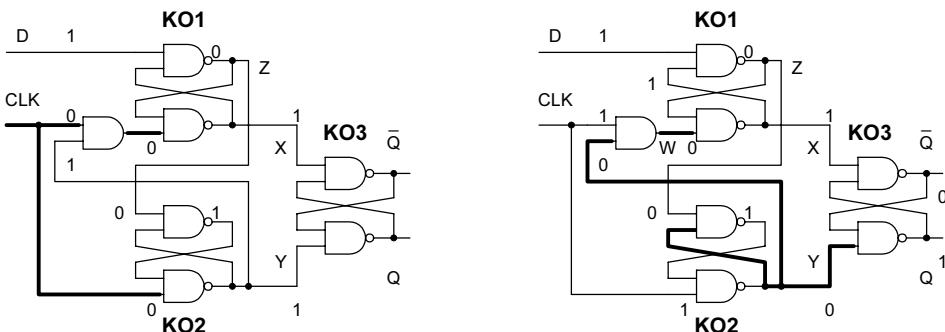
Obr. 8.22 Pravidlostní tabulka a schematické značky klopných odvodů řízených hranou

Klopný obvod řízený hranou je podstatně **složitější**, než klopný obvod řízený úrovni. Základní zapojení KO řízeného náběžnou hranou ukazuje obr. 8.23. Vlevo je znázorněna situace po dobu $CLK = 0$. Tlusté čáry ukazují signály, které hrají dominantní roli. Zřejmě $X = Y = 1$ a tudíž KO3 nemění stav bez ohledu na stav na vstupu D . Vpravo je ukázána situace po náběhu $CLK = 1$, kdy bylo $D = 0$. X se změní na 0 a Y zůstane 1, takže KO3 se překlopí do $Q = 0$. Signálem $X = 0$ jsou blokovány změny v bodě Z , takže změny D po dobu $CLK = 1$ nemají vliv.



Obr. 8.23 Funkce klopného odvodu řízeného hranou při $D = 0$

Další obr. 8.24 ukazuje situaci při $D = 1$. Vlevo je vyznačen stav před náběhem CLK , kdy bylo $D = 1$. Vpravo je situace po náběhu CLK . Nejprve se v bodě W objeví stav 1 (v obr. 8.24 nenařazeno), to ale neovlivní stav KO1, takže bude stále $Z = 0$. Tím se překlopí KO2 a $Y = 0$, následně bude $W = 0$. Tím je zajištěno $X = 1$ bez ohledu na případné změny na D . Současně se překlopí KO3 do $Q = 1$.



Obr. 8.24 Funkce klopného odvodu řízeného hranou při $D = 1$

Podle stavu na D v okamžiku změny CLK z 0 do 1 se tedy buď na X nebo na Y objeví stav 0 po dobu trvání $CLK = 1$. Tím se překlopí KO3. Během trvání $CLK = 1$ již změny D nemají vliv. Po návratu CLK do 0 se obnoví situace naznačená v levých obrázcích. Při té je vždy $X = Y = 1$, takže KO3 nemůže změnit stav.

Následující výpis kódu ukazuje realizaci klopného obvodu řízeného hranou pomocí procesu. Proces má v citlivostním seznamu uveden signál clk . Proces je tedy spouštěn kdykoliv se signál clk změní, přičemž do q se zapisuje jen pokud došlo na signálu clk k události (atribut **event** vrátí hodnotu **true**) a současná hodnota signálu clk je rovna log. 1. Tj. na signálu clk nastala náběžná hrana.

```

library ieee;
use ieee.std_logic_1164.all;

entity dff_up is
  port (
    d, clk : in std_logic;
    q : out std_logic
  );
end dff_up;

architecture struct of dff_up is
begin
  -- proces se spousti pri zmene signalu clk
  process (clk)
  begin
    -- klopný obvod s hodinami aktivními pri nabezne hrane
    if clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end struct;

```

Výpis 8.20 Klopny obvod řízený hranou realizovaný pomocí procesu

Hranové řízení existuje i pro jiné typy klopných obvodů. Obr. 8.25 ukazuje **klopny obvod JK**.

<i>J</i>	<i>K</i>	<i>CLK</i>	<i>Q</i>	\bar{Q}
0	0	\uparrow	Q	\bar{Q}
1	0	\uparrow	1	0
0	1	\uparrow	0	1
1	1	\uparrow	\bar{Q}	Q
-	-	0	Q	\bar{Q}
-	-	\downarrow	Q	
-	-	1	Q	\bar{Q}

Obr. 8.25 Pravdivostní tabulka a schematická značka klopného obvodu JK řízeného hranou

První a poslední tři řádky znamenají, že KO nemění stav. Čtvrtá řádka znamená, že KO střídá stav při každé náběžné hraně *CLK*. Klopny obvod JK je velmi univerzální.

Následující výpis kódu ukazuje realizaci klopného obvodu JK řízeného hranou pomocí procesu a konstrukce s **case**. Proces má v citlivostním seznamu uveden signál *clk*. Proces je tedy spouštěn kdykoliv se signál *clk* změní. Do proměnné *ff* se zapisuje jen pokud došlo na signálu *clk* k události (atribut **event** vrátí hodnotu **true**) a současná hodnota signálu *clk* je rovna log. 1 (nastala na *clk* náběžná hrana). Pravdivostní tabulka je realizována pomocí **case**.

```

library ieee;
use ieee.std_logic_1164.all;

entity jk_ff is
  port (
    j : in std_logic;
    k : in std_logic;
    clk : in std_logic;
    q : out std_logic;
    n_q : out std_logic
  );
end jk_ff;

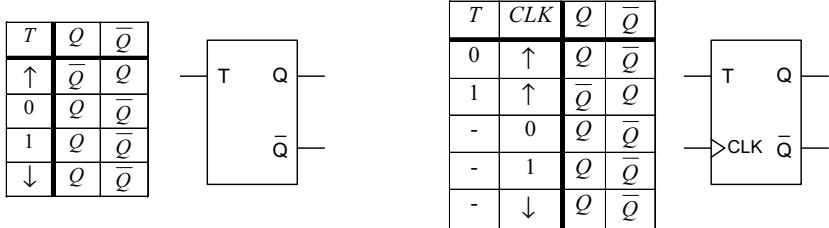
architecture struct of jk_ff is
  signal jk : std_logic_vector (1 downto 0);
begin
  -- sloučení signálu j a k do jednoho dvobitového signálu jk
  jk <= j & k;

  -- proces se spustí při změně signálu clk
  process (clk)
    -- proměnná udržující hodnotu klopného obvodu
    variable ff : std_logic;
  begin
    if clk'event and clk = '1' then
      -- realizace pravdivostní tabulky klopného obvodu JK pomocí case
      case jk is
        when "01" =>
          ff := '0';
        when "10" =>
          ff := '1';
        when "11" =>
          ff := not ff;
        when others =>
          null;
      end case;
      -- výstup hodnoty klopného obvodu na signály q a n_q
      q <= ff;
      n_q <= not ff;
    end if;
  end process;
end struct;

```

Výpis 8.21 Klopný obvod JK řízený hranou realizovaný pomocí procesu a case

Dalším typem klopného obvodu s hranovým řízením je **klopný obvod T**. Existuje ve dvou variantách – asynchronní a synchronní (viz obr. 8.26). **Asynchronní T** obvod s každou náběžnou hranou na vstupu T střídá stav. **Synchronní T** obvod střídá stav s každou náběžnou hranou CLK , ale jen při $T = 1$.



Obr. 8.26 Pravdivostní tabulky a schematické značky klopného obvodu T řízeného hranou – vlevo T asynchronní, vpravo T synchronní

Následující výpis kódu ukazuje realizaci asynchronního klopného obvodu typu T pomocí procesu. Proces má v citlivostním seznamu uveden signál r a t . V případě, že signál r má hodnotu log. 1 je vnitřní registr ff nulován. Vstup pro nulování má největší prioritu. V případě, že na signálu t nastala náběžná hrana dochází k negování vnitřního registru ff .

```

library ieee;
use ieee.std_logic_1164.all;

entity t_ff_async is
  port (
    t : in std_logic;
    r : in std_logic;
    q : out std_logic;
    n_q : out std_logic
  );
end t_ff_async;

architecture struct of t_ff_async is
begin
  -- proces se spusti pri zmene signalu r nebo t
  process (r, t)
    -- promenna udrzujujici hodnotu klopnego obvodu
    variable ff : std_logic;
  begin
    -- reset
    if r = '1' then
      ff := '0';
    -- realizace asynchronniho klopnego obvodu T
    elsif t'event and t = '1' then
      ff := not ff;
    end if;
    -- vystup hodnoty klopnego obvodu na signaly q a n_q
    q <= ff;
    n_q <= not ff;
  end process;
end struct;

```

Výpis 8.22 Asynchronní klopný obvod typu T realizovaný pomocí procesu

Následující výpis kódu ukazuje realizaci synchronního klopného obvodu typu T pomocí procesu. Proces má v citlivostním seznamu uveden signál *clk* a *r*. V případě, že signál *r* má hodnotu log. 1 je vnitřní registr *ff* nulován. Vstup pro nulování má největší prioritu. V případě, že na signálu *clk* nastala náběžná hrana a vstup *t* je v log. 1 dochází k negování vnitřního registru *ff*.

```

library ieee;
use ieee.std_logic_1164.all;

entity t_ff_sync is
  port (
    t : in std_logic;
    r : in std_logic;
    clk : in std_logic;
    q : out std_logic;
    n_q : out std_logic
  );
end t_ff_sync;

architecture struct of t_ff_sync is
begin
  -- proces se spustí pri zmene signalu clk nebo r
  process (clk, r)
    -- promenna udrzujici hodnotu klopneho obvodu
    variable ff : std_logic;
  begin
    -- asynchronni reset
    if r = '1' then
      ff := '0';
    -- realizace synchronniho klopneho obvodu T
    elsif clk'event and clk = '1' then
      if t = '1' then
        ff := not ff;
      end if;
    end if;
    -- vystup hodnoty klopneho obvodu na signaly q a n_q
    q <= ff;
    n_q <= not ff;
  end process;
end struct;

```

Výpis 8.23 Synchronní klopný obvod typu T realizovaný pomocí procesu

Pro správnou činnost klopného obvodu s hranovým řízením je nutné dodržet některé dynamické parametry vstupních signálů. Předně je to **doba předstihu** a **doba přesahu**. Dále je to minimální **strmost** hrany hodinového impulzu, minimální doba jeho **trvání** (ve stavu 1) a jeho minimální **prodleva** (ve stavu 0). Nedodržení některé z těchto podmínek může vyvolat **metastabilní** stav, kdy výsledné překlopení je náhodné.

Základní vlastnosti klopného obvodu řízeného hranou jsou:

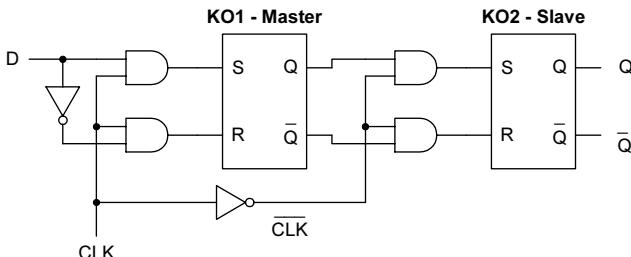
- Rozhodující je stav na vstupech v těsném okolí náběžné hrany hodinového impulzu.
- V jiných okamžicích jsou změny na vstupech bezvýznamné.
- Nevhodné časování hodinových impulzů a vstupních signálů může vyvolat **metastabilní stav**.

Metastabilní stav má za následek **nespolehlivou funkci** klopného obvodu. Klopný obvod se může rozkmitat, nebo se může překlopit do nesprávného stavu, nebo se může překlopit do správného stavu s velkým zpožděním. Dobu zpoždění nelze předem předvídat.

Jako příklad dynamických parametrů klopného obvodu řízeného náběžnou hranou mohou sloužit parametry obvody 74AHCT74:

$t_W \geq 5 \text{ ns}$... délka hodinového impulzu
$f_{CLK} \leq 160 \text{ MHz}$... kmitočet hodinových impulzů
$t_s \geq 5 \text{ ns}$... doba předstihu
$t_h \geq 0$... doba přesahu
$t_{pCQ} = 6,3 \dots 8,8 \text{ ns}$... doba zpoždění výstupu za hranou CLK
$\Delta t/\Delta U \leq 20 \text{ ns/V}$... strmost hrany CLK

Třetí způsob řízení klopného obvodu je řízení **dvojfázové** nebo „**Master Slave**“. Obvod D-Master Slave (zkráceně MS) ukazuje obr. 8.27. Klopný obvod je složen ze dvou dílčích obvodů, při čemž jejich hodinové impulzy jsou vzájemně negované.

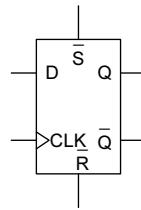


Obr. 8.27 Klopný obvod typu D – Master Slave

Během $CLK = 1$ je KO1 otevřen pro změny na vstupu, ale vzhledem k obrácené fázi \overline{CLK} je KO2 při tom blokován, takže na výstupech Q a \overline{Q} klopného obvodu KO2 se nezmění stav. Až při $CLK = 0$ se stav vnitřního KO1 přesune na KO2. Vnitřní stav KO1 se tedy nastavuje **po celou dobu** trvání $CLK = 1$ (srovnej s KO řízeným úrovní), ale je **viditelný** na výstupu až při $CLK = 0$. Klopný obvod MS tedy není nikdy průchozí pro vstupní signály – v tom se podobá KO řízenému hranou. Rozdílnost je v tom, že jeho budoucí stav je dán stavem vstupů na konci prvej poloviny taktu, kdy $CLK = 1$, nikoliv stavem vstupů při náběžné hraně CLK . Princip MS se neomezuje jen na obvod D, ale může být využit i u jiných typů (JK).

U obvodů velké integrace, kde se vyskytuje velký počet klopných obvodů, je důležitá jejich jednoduchá konstrukce. Tomu vyhovují dvojfázové klopné obvody MS, do kterých jsou zavedeny hodinové impulzy ve dvou fázích, tedy CLK i \overline{CLK} . Uspoří se tak invertor v každém KO.

Některé klopné obvody D, T a JK mají kromě vstupů CLK , D , T , J a K ještě asynchronní vstupy **nulování**, případně nulování i **nastavení**. Jsou účinné vždy, bez ohledu na ostatní vstupy. Vstup nulování, kterým je vnučen stav $Q = 0$, je značen jako R (Reset) nebo CLR (Clear), aktivní ve stavu 0. Vstup nastavení, kterým je vnučen stav $Q = 1$, je značen jako S (Set) nebo PRE (Preset), aktivní rovněž ve stavu 0.



Obr. 8.28 Synchronní klopný obvod typu D s asynchronními vstupy pro nastavení a nulování

Současné vyvolání mazání i nastavení, $\bar{R} = \bar{S} = 0$, může vést na **nedefinovaný** výsledný stav klopného obvodu (srovnej s klopným obvodem RS).

Následující výpis kódu ukazuje realizaci synchronního klopného obvodu typu D s asynchronními vstupy pro nastavení a nulování. Proces má v citlivostním seznamu uveden signál *clk*, *r* a *s*. Podmínky realizující asynchronní reakci na vstupy musí být uvedeny před vlastní synchronní podmínkou realizující vlastní klopný obvod. Na rozdíl od předchozího výkladu je obvod realizován tak, že asynchronní vstup pro nulování má největší prioritu, takže při současném vyvolání nastavení a nulování je klopný obvod nulován.

```

library ieee;
use ieee.std_logic_1164.all;

entity dff_up_sr is
  port (
    d, clk, r, s : in std_logic;
    q : out std_logic
  );
end dff_up_sr;

architecture struct of dff_up_sr is
begin
  -- proces se spustí pri zmene signalu clk nebo r nebo s
  process (clk, r, s)
  begin
    -- realizace asynchronniho nulovani (ma vetsi prioritu nez nastaveni)
    if r = '0' then
      q <= '0';
    -- realizace asynchronniho nastaveni
    elsif s = '0' then
      q <= '1';
    -- realizace synchronniho klopneho obvodu
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end struct;

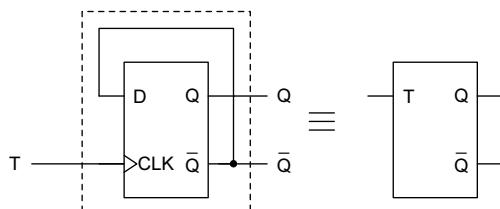
```

Výpis 8.24 Synchronní klopný obvod typu D s asynchronními vstupy pro nastavení a nulování

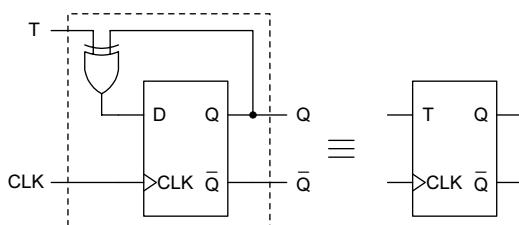
8.11 Transformace klopných obvodů

Klopný obvod jednoho typu lze převést – transformovat – na klopný obvod jiného typu. Prvním případem je **transformace klopného obvodu D** řízeného hranou na asynchronní T (viz obr. 8.29). Na vstupu D je opačný stav než na výstupu Q , proto příštím hodinovým impulzem bude KO překlopen do opačného stavu. Stav na výstupu Q se tak stále střídá.

Dalším případem je transformace D na T synchronní (viz obr. 8.30). Při $T = 0$ obvod EX-OR neneguje, takže $D = Q$ a stav KO se nezmění. Při $T = 1$ se na D dostane negace Q a KO střídá stav.

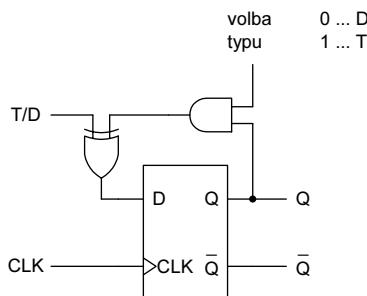


Obr. 8.29 Transformace klopného obvodu typu D řízeného hranou na asynchronní klopný obvod typu T

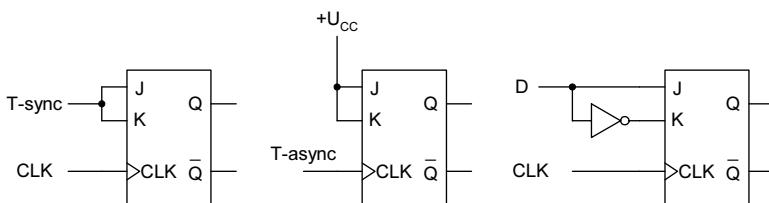


Obr. 8.30 Transformace klopného obvodu typu D řízeného hranou na synchronní klopný obvod typu T

Obvod na obr. 8.31 je univerzální a může pracovat jako D nebo T synchronní. To je využíváno v některých programovatelných obvodech. Je-li žádán typ D, pak $X = 0$ a EX-OR neneguje vstupní signál T/D – obvod pracuje jako typ D. Je-li žádán typ T, pak $X = Q$ a nastává případ naznačený na předchozím obrázku – obvod pracuje jako typ T.



Obr. 8.31 Univerzální klopný obvod D/T



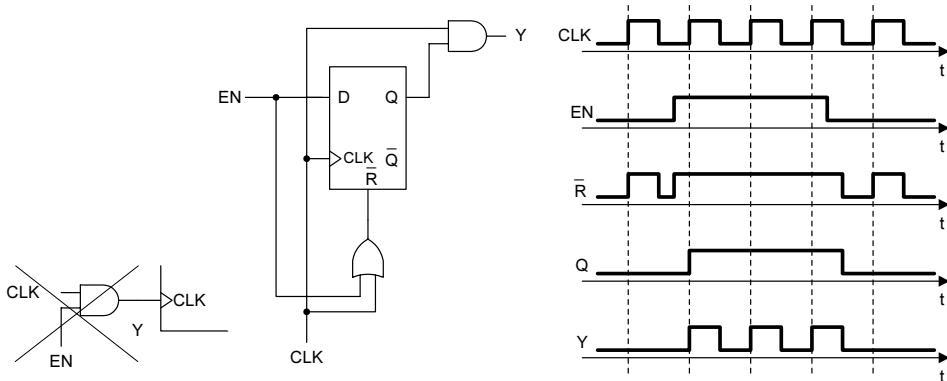
Obr. 8.32 Transformace klopného obvodu typu JK na T synchronní, T asynchronní a D

Řadu možností pro transformace dává klopný obvod JK, který je velmi univerzální. Obr. 8.32 shrnuje jednotlivé možnosti.

Výše uvedené transformace nejsou jedinými možnými, patří však mezi nejčastější.

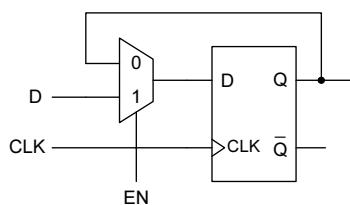
8.12 Blokování klopných obvodů

Pro spolehlivou činnost klopného obvodu je bezpodmínečně nutné dodržet podmínky správného časování vstupních signálů vzhledem k hodinovým impulzům. Často je potřeba hodinové impulzy **propouštět či blokovat**. Jednoduché uspořádání se součinovým členem, jak ukazuje obr. 8.33 vlevo, **nemůže vyhovět**. Pokud totiž signál EN, povolující průchod impulzů, nebude vhodně časován, může z hodinového impulzu „useknout“ jen krátký úsek, nedostatečný pro spolehlivé překlopení klopného obvodu. Ten se tak může dostat do metastabilního stavu s nahodilým výsledným stavem. Vyhovující řešení představuje zapojení podle obr. 8.33 uprostřed s časovým diagramem na obrázku vpravo.



Obr. 8.33 Synchronní spínač impulzů

Hodinové impulzy, které projdou na výstup Y , jsou vždy celé, bez ohledu na okamžiky změn signálu EN . Stav $EN = 1$ však musí trvat nejméně 1 periodu CLK , aby byl vždy účinný. Celé zapojení představuje **synchronní spínač impulzů**.



Obr. 8.34 Blokace činnosti klopného obvodu

Jinou možností, jak **zablokovat činnost klopného** obvodu D, je na obr. 8.34. Zde se nejedná o blokování dodávky hodinových impulzů, ale o vhodné přepnutí signálu na vstupu D. Při $EN = 1$ pracuje KO normálně, při $EN = 0$ je $D = Q$ a KO je blokován. Signál EN se může měnit kdykoliv s výjimkou oblasti kolem náběžné hrany hodinového impulzu.

Následující výpis kódu ukazuje realizaci synchronního klopného obvodu typu D se vstupem *en* pro blokování činnosti klopného obvodu. Proces má v citlivostním seznamu uveden pouze signál *clk*. Podmínka realizující blokování funkce klopného obvodu je umístěna uvnitř synchronní podmínky realizující vlastní klopný obvod.

```

library ieee;
use ieee.std_logic_1164.all;

entity dff_up_en is
  port (
    d, clk, en : in std_logic;
    q : buffer std_logic
  );
end dff_up_en;

architecture struct of dff_up_en is
begin
  -- proces se spousti pri zmene signalu clk
  process (clk)
  begin
    -- realizace synchronniho klopneho obvodu
    if clk'event and clk = '1' then
      -- blokovani funkce klopneho obvodu
      if en = '1' then
        q <= d;
      else
        q <= q;
      end if;
    end if;
  end process;
end struct;

```

Výpis 8.25 Blokování činnosti klopného obvodu

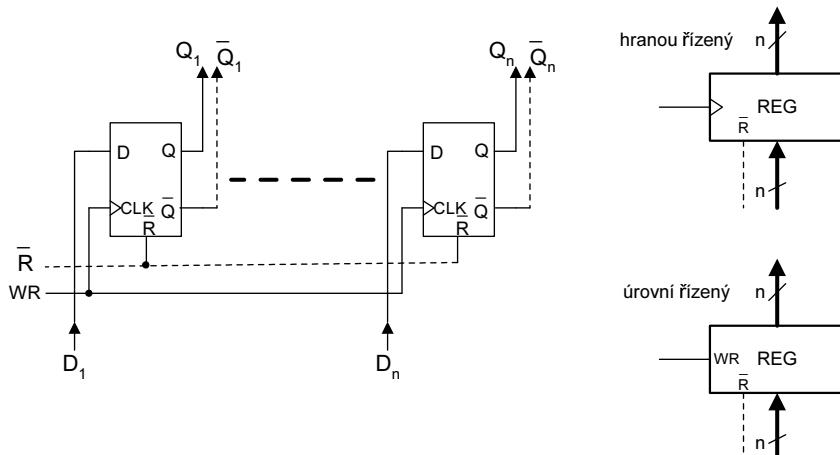


REGISTRY A ČÍTAČE

Registry jsou velmi častým stavebním blokem v číslicových systémech. Jsou založeny na synchronních klopných obvodech, nejčastěji typu D. Dva hlavní typy registrů jsou datový registr a posuvný registr. Datový registr slouží k zachycení dat ve vhodných okamžicích a k jejich dočasnému pamatování. Posuvný registr slouží k posouvání dat o jistý počet pozic doleva či doprava. Čítače slouží k odpočítávání počtu impulzů a k dělení kmitočtu celým číslem.

9.1 Datové registry

Datový registr (nebo častěji jen „registr“) je jednoduché seskupení klopných obvodů D, které mají společný rozvod hodinových impulzů, případně i společné nulování a někdy i nastavení. Obr. 9.1 ukazuje spojení obvodů a symbolickou značku. Klopné obvody mohou být řízeny **hranou** (symbol vpravo nahoře) nebo **úrovni** (symbol vpravo dole), takže i celý registr je řízen buď hranou (tak je tomu na obrázku vlevo) nebo úrovní. To je vyjádřeno i v symbolické značce. **Nulování**, které je zavedeno jen u některých registrů, je vyznačeno čárkovanou. Rovněž tak negované výstupy klopných obvodů. Místo označení „hodinový impulz“ se zpravidla používá „zápisový impulz“, označený jako *WR* (Write).



Obr. 9.1 Datový registr

Jak bylo ukázáno v kapitole o klopných obvodech, řízení hranou i řízení úrovní má své specifické podmínky pro správnou funkci. Údaje o klopných obvodech platí v plné míře i oregistrech. Zvláště důležité jsou **doby předstihu a přesahu**, a též **tvar** hodinového impulzu.

Oba způsoby řízení – hranou i úrovní – mají své uplatnění. Registr s úrovňovým řízením je jednodušší a umožňuje jednak zachycení dat, jednak průchod dat ze svých vstupů na výstupy po celou dobu, kdy $WR=1$ – registr je „**transparentní**“. To se hodí např. u převodníků D/A, kdy v synchronním režimu převodu jsou vstupní data vzorkována zápisovými impulzy *WR* a pak převáděna na analogový signál, ale ve druhém možném režimu jsou převáděna okamžitě (*WR* je trvale 1). Naopak registr s hranovým řízením vyžaduje vždy přísun zápisových impulzů, takže registr nikdy není transparentní. To je nutná vlastnost v mnoha synchronních sekvenčních obvodech.

Následující výpis kódu ukazuje realizaci n -bitového registru řízeného hranou v jazyce VHDL pomocí procesu. Proces má v citlivostním seznamu uveden signál clk a n_r . Podmínka realizující asynchronní nulování registru je uvedena před synchronní podmínkou realizující vlastní hranový registr. Proces je spouštěn kdykoliv se signál clk nebo n_r změní, přičemž do q se zapisuje jen pokud není aktivní signál n_r a došlo-li na signálu clk k události (atribut **event** vrátí hodnotu **true**) a současná hodnota signálu clk je rovna log. 1 (tj. na signálu clk nastala náběžná hrana). Počet bitů registru lze jednoduše měnit při mapování komponentu změnou hodnoty parametru n v generické části.

```

library ieee;
use ieee.std_logic_1164.all;

entity reg_n_hr is
  generic (
    n : positive := 8
  );
  port (
    d : in std_logic_vector (1 to n);
    clk, n_r : in std_logic;
    q : out std_logic_vector (1 to n)
  );
end reg_n_hr;

architecture struct of reg_n_hr is
begin
  -- registr s hodinami aktivními pri nabezne hrane a asynchronnim nulovanim
  process (clk, n_r)
  begin
    -- realizace asynchronniho nulovani
    if n_r = '0' then
      q <= (others => '0');
    -- realizace synchronniho registru
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end struct;

```

Výpis 9.1 Parametrizovatelný hranový registr s asynchronním nulováním

Následující výpis kódu ukazuje realizaci n -bitového hladinového registru. Proces má v citlivostním seznamu uvedeny signály n_r , d a wr . Podmínka realizující nulování registru má největší prioritu. Proces je spouštěn kdykoliv se signál n_r , d nebo wr změní. Do q se zapíše hodnota d pokud není aktivní signál n_r a je-li aktivní signál wr . Počet bitů registru lze opět jednoduše měnit změnou hodnoty parametru n při mapování generické části.

```

library ieee;
use ieee.std_logic_1164.all;

entity reg_n_hl is
  generic (
    n : positive := 8
  );

```

```

port (
    d : in std_logic_vector (1 to n);
    wr, n_r : in std_logic;
    q : out std_logic_vector (1 to n)
);
end reg_n_hl;

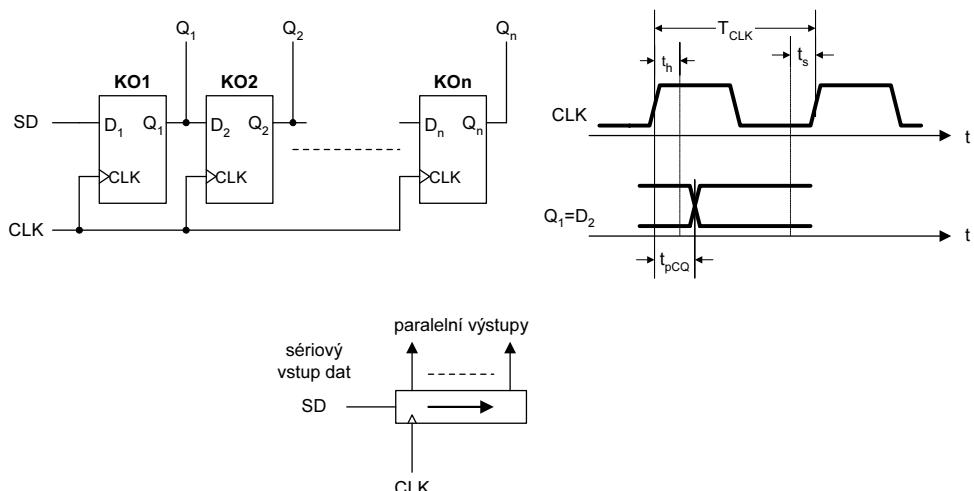
architecture struct of reg_n_hl is
begin
    -- hladinovy registr s nulovanim
    process (n_r, d, wr)
    begin
        -- realizace asynchronniho nulovani
        if n_r = '0' then
            q <= (others => '0');
        -- realizace hladinoveho registru
        elsif wr = '1' then
            q <= d;
        end if;
    end process;
end struct;

```

Výpis 9.2 Parametrizovatelný hladinový registr s nulováním

9.2 Posuvné registry

Spojením klopných obvodů typu D do kaskády se společným rozvodem hodinových impulzů vznikne posuvný registr. Základní zapojení a grafický symbol ukazuje obr. 9.2.



Obr. 9.2 Posuvný registr

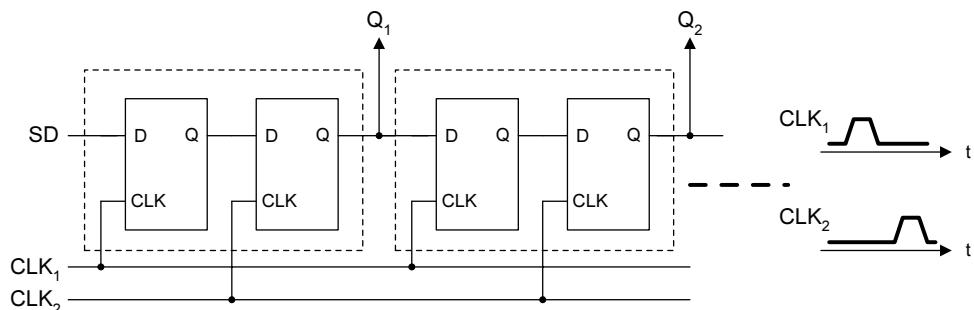
Hodinovým impulzem se stav na vstupu D klopného obvodu KO1 přesune na jeho výstup Q_1 , ale současně se původní stav na výstupu Q_1 obvodu KO1 přesune na výstup Q_2

obvodu KO2, atd. Výsledkem je, že celý obsah registru se **posunuje doprava**, zleva je doplnován vstupními daty, a za posledním KO_n vpravo se **ztrácí**. Tak např. pro $n = 4$ ze stavu registru 0011 a vstupu 1 se přejde do stavu 1001, poslední jednička vpravo se ztrácí. Tato jednoduchá činnost však není samozřejmá a vyžaduje dodržení jistých podmínek **správného časování**, jak ukazuje obrázek vpravo. Impulz CLK je stejný pro všechny KO. Po jeho náběžné hraně, se zpožděním t_{pCQ} , se překlopí KO1, ale pro KO2 musel být signál D_2 stabilní ještě po dobu přesahu t_h po hraně CLK. Dále musí být signál D_2 stabilní minimálně po dobu t_s před hranou dalšího CLK. Shodně je tomu i u dalších KO. To vede k soustavě podmínek:

$$\begin{aligned} t_{pCQ} &\geq t_h \\ T_{CLK} &\geq t_{pCQ} + t_s \end{aligned}$$

Například pro klopný obvod 74AHCT74 s parametry $t_s \geq 5$ ns, $t_h \geq 0$, $t_{pCQ} = 6,3$ ns by uvedené podmínky bylo možno splnit při $T_{CLK} \geq 11,3$ ns. U mnoha klopných obvodů je povoleno $t_h=0$, takže prvá podmínka je splněna automaticky.

V usporádání podle předchozího obrázku musí být všechny KO řízeny **hranou**. Řízení úrovní nelze použít, neboť během doby, kdy $CLK=1$, by u všech KO platilo $Q_i=D_i$. Výsledkem by bylo zaplnění celého registru hodnotou na vstupu SD. Řešením je **dvojfázové** řízení registru podle obr. 9.3 (srovnej s klopnými obvody Master-Slave). CLK_1 a CLK_2 se nesmějí překrývat.



Obr. 9.3 Posuvný registr s dvojfázovým řízením

Jedna z nejčastějších aplikací posuvného registru je převod **sériového** kódu na kód **paralelní**. Sériová data se přivádějí na vstup SD bit po bitu a vždy jsou doprovázena impulzem CLK. Po patřičném počtu impulzů jsou paralelní data k dispozici na výstupech Q. Opačná aplikace, tj. převod **paralelních** dat na data **sériová**, vyžaduje zaplnění posuvného registru daty a následně přivedení patřičného počtu impulzů CLK. Sériová data jsou k dispozici na posledním KO vpravo.

Pro rychlé naplnění posuvného registru daty jsou některé registry vybaveny obvody pro paralelní vložení dat či **přednastavení** (angl. *preload*), běžné je též **nulování**. Jednu z možností ukazuje obr. 9.4.

Vstupem PE=1 (Preload Enable) se přepne multiplexor a následující impulz CLK neposune data doprava, nýbrž nastaví KO podle hodnot D_i na paralelních vstupech. Pak se PE změní na 0 a na náběžnou hranu signálu CLK registr posouvá data. Jelikož přednastavení se

děje pomocí synchronizačního impulzu *CLK*, jedná se o **synchronní přednastavení**. Nulo-vání je nezávislé na *CLK* a je proto **asynchronní**.

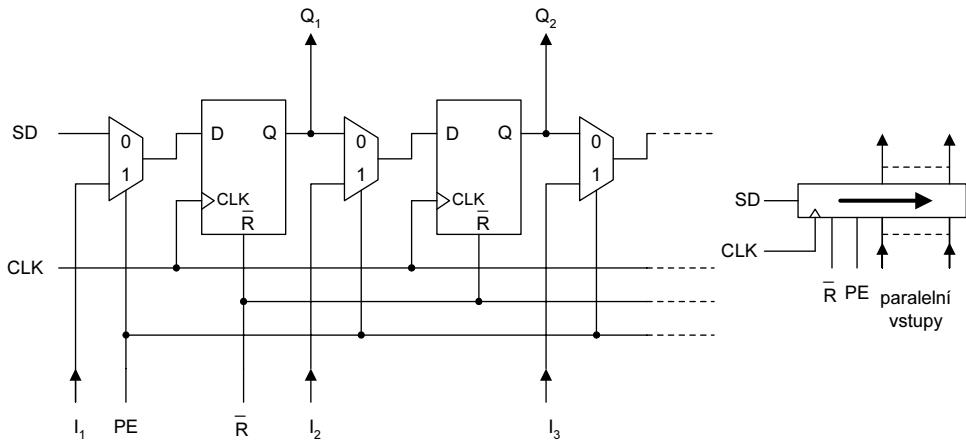
Následující výpis kódu ukazuje realizaci *n*-bitového posuvného registru vpravo s asyn-chronním nulováním a synchronním nastavením v jazyce VHDL pomocí procesu. Proces má v citlivostním seznamu uveden signál *clk* a *n_r*. Podmínka realizující asynchronní nulování registru je uvedena před synchronní podmínkou realizující nastavení a vlastní posuvný registr. Proces je spouštěn kdykoliv se signál *clk* nebo *n_r* změní, přičemž registr se posouvá vpravo jen pokud není aktivní signál *n_r* a došlo-li na signálu *clk* k události (atribut **event** vrátí hodnotu **true**) a současná hodnota signálu *clk* je rovna log. 1 (tj. na signálu *clk* nastala náběžná hrana). V případě, že je v té době aktivní signál *pe* dojde místo posunu registru k zápisu hodnoty ze vstupu *d* do registru. Počet bitů registru lze opět jednoduše měnit při mapování změnou hodnoty parametru *n* v generické části.

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

entity posuv_reg is
    generic (
        n : positive := 8
    );
    port (
        clk : in std_logic;
        sd : in std_logic;
        n_r : in std_logic;
        pe : in std_logic;
        d : in std_logic_vector (1 to n);
        q : out std_logic_vector (1 to n)
    );
end posuv_reg;

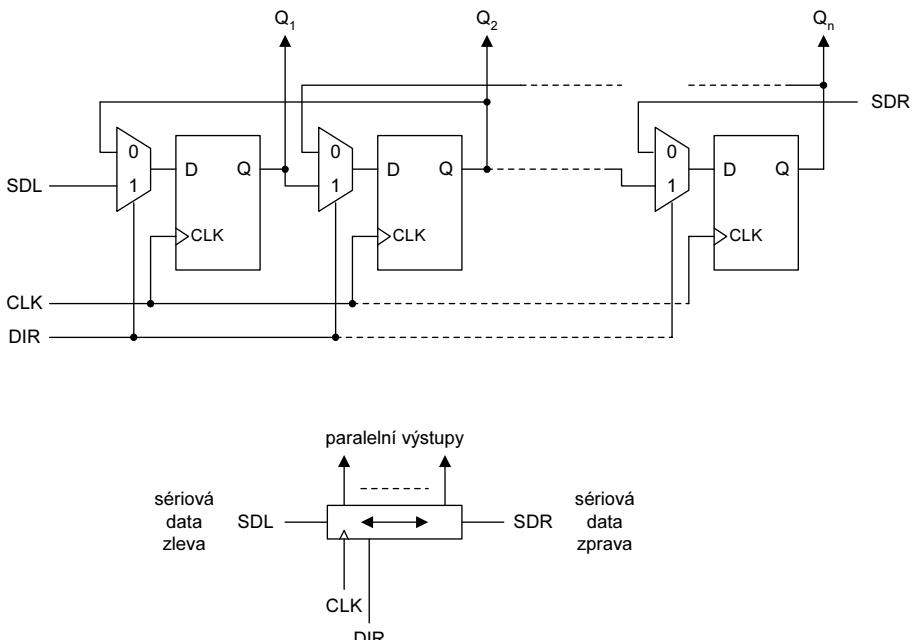
architecture struct of posuv_reg is
    signal reg : std_logic_vector (1 to n);
begin
begin
    process (clk, n_r)
    begin
        -- realizace asynchronniho nulovani
        if n_r = '0' then
            reg <= (others => '0');
        elsif clk'event and clk = '1' then
            -- realizace synchronniho nastaveni registru
            if pe = '1' then
                reg <= d;
            else
                -- posun registru vpravo a nasunuti novych dat zleva
                reg <= sd & reg (1 to n - 1);
            end if;
        end if;
    end process;
    q <= reg;
end struct;
```

Výpis 9.3 Parametrizovatelný posuvný registr s asynchronním nulováním a synchronním nastavením



Obr. 9.4 Synchronní přednastavení a asynchronní nulování posuvného registru

Posouvání dat jen jedním směrem a jen o jedno místo není jedinou alternativou pro posuvné registry. Obr. 9.5 ukazuje posuvný registr s volbou posunu **vlevo** či **vpravo**. Podle hodnoty na vstupu DIR multiplexory přepnou vazby mezi sousedními KO. Při posunu vlevo jsou sériová data dodávána do posledního KO zprava, za prvním KO zleva se data ztrácí.



Obr. 9.5 Dvojsměrný posuvný registr

Pomocí vhodně zapojených multiplexorů je možné přepínat vazby nejen mezi sousedními KO zleva či zprava, ale i mezi KO – ob jeden, ob dva, atd. Tak lze realizovat posuny

o volitelný počet míst doleva či doprava. To je důležité např. v aritmetických jednotkách optimalizovaných pro číslicové zpracování signálu.

Následující výpis kódu ukazuje realizaci n -bitového dvojsměrného posuvného registru s asynchronním nulováním a synchronním nastavením v jazyce VHDL pomocí procesu. Proces má v citlivostním seznamu uveden pouze signál *clk*. Proces je spouštěn kdykoliv se signál *clk* změní. Posun registru nastává při vzestupné hraně na signálu *clk*, přičemž směr posouvání určuje hodnota signálu *dir*. Počet bitů registru lze opět jednoduše měnit změnou hodnoty parametru *n* v generické části deklarace entity.

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

entity posuv_reg_bi is
    generic (
        n : positive := 8
    );
    port (
        clk : in std_logic;
        dir : in std_logic;
        sdl : in std_logic;
        sdr : in std_logic;
        q : out std_logic_vector (1 to n)
    );
end posuv_reg_bi;

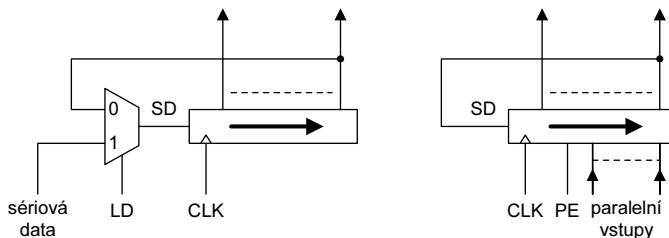
architecture struct of posuv_reg_bi is
    signal reg : std_logic_vector (1 to n);
begin
    -- realizace dvojsmerneho posuvneho registru
    process (clk)
    begin
        if clk'event and clk = '1' then
            if dir = '1' then
                -- posun registru pravo a nasunuti novych dat zleva
                reg <= sdl & reg (1 to n - 1);
            else
                -- posun registru vlevo a nasunuti novych dat zprava
                reg <= reg (2 to n) & sdr;
            end if;
        end if;
    end process;
    q <= reg;
end struct;
```

Výpis 9.4 Parametrizovatelný dvojsměrný posuvný registr

9.3 Posuvné registry se zpětnou vazbou

Pokud je na sériový vstup dat přiveden signál z některých výstupů posuvného registru – ať již přímo, nebo po zpracování kombinačním obvodem – vzniká zpětná vazba, se kterou získává posuvný registr nové vlastnosti.

Nejjednodušším případem je **kruhový registr**, u kterého je zpětná vazba vedena přímo z posledního klopného obvodu – viz obr. 9.6. Data, vložená předem do posuvného registru, pak neustále rotují vždy o jedno místo s každým hodinovým (posouvacím) impulzem. Data lze do registru vložit buď prostřednictvím paralelních vstupů (obr. 9.6 vpravo), nebo sériově, bit za bitem, přes vstupní multiplexor (obr. 9.6 vlevo).



Obr. 9.6 Kruhové registry se sériovým a paralelním vstupem

Rotující obsah kruhového registru lze využít např. pro generaci vícefázových řídicích impulzů, pro běžící náписy na zobrazovacích panelech, apod. Výhodou takovýchto generátorů řídicích impulzů je jednak možnost jejich rychlého přaprogramování pouhým vložením jiného počátečního obsahu do registru, jednak čistota průběhu impulzů na výstupech, neboť při dodržení zásad správného provozu klopných obvodů na jejich výstupech nevznikají falešné impulzy.

Zajímavý efekt vznikne při zpětné vazbě z **negovaného** výstupu posledního klopného obvodu a počátečním stavu registru 00...00. Posloupnost stavů pro příklad 4bitového registru je pak následující:

	Q0	Q1	Q2	Q3	\bar{Q}_3	
→	0	0	0	0	1	
	1	0	0	0	1	
	1	1	0	0	1	
	1	1	1	0	1	
	1	1	1	1	0	
	0	1	1	1	0	
	0	0	1	1	0	
	0	0	0	1	0	
	0	0	0	0	1	
opakuje se						

Zřejmě se jedná o obvod se sekvencí osmi stavů (obecně $2 \cdot N$ stavů), která se cyklicky opakuje. Zapojení se nazývá **Johnsonův čítač** (angl. též *twisted ring counter*).

Následující výpis kódu ukazuje realizaci čtyřbitového Johnsonova čítače v jazyce VHDL pomocí procesu. Proces má v citlivostním seznamu uvedeny signály *reset* a *clk*. V případě, že signál *reset* má hodnotu log. 1 je registr *reg* nulován. Posun registru nastává při vzestupné hraně na signálu *clk*, přičemž směr posuvu je vpravo a zleva se nasouvá do registru negovaná hodnota posledního pravého bitu registru tj. bit *reg(3)*.

```

library ieee;
use ieee.std_logic_1164.all;

entity johnson is
    port (
        reset : in std_logic;
        clk : in std_logic;
        q : out std_logic_vector (0 to 3)
    );
end johnson;

architecture struct of johnson is
    signal reg : std_logic_vector (0 to 3);
begin
begin
    process (reset, clk)
    begin
        -- asynchronni reset
        if reset = '1' then
            reg <= "0000";
        -- realizace Jonhsonova citace
        elsif clk'event and clk = '1' then
            reg <= not reg(3) & reg(0 to 2);
        end if;
    end process;
    q <= reg;
end struct;

```

Výpis 9.5 Johnsonův čítač

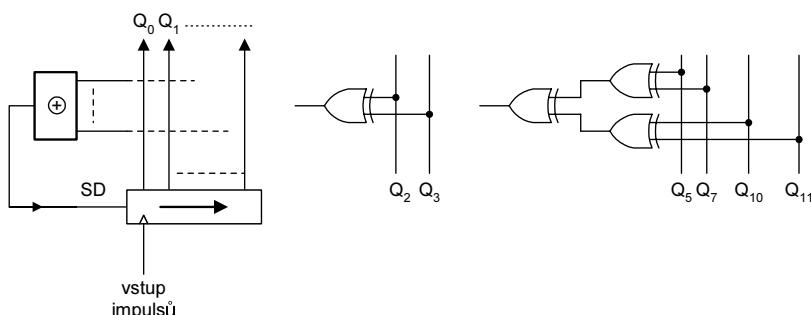
Ve zpětné vazbě může být zapojen i složitější kombinační obvod. Zvláštní důležitost má zapojení se členy XOR (neekvivalence). Jednoduchý člen XOR má jen dva vstupy, jednotlivé členy však lze seskupovat, neboť pro funkci XOR platí asociativní zákon:

$$a \oplus b \oplus c \oplus d \dots = (a \oplus b) \oplus (d \oplus e) \dots$$

nebo též:

$$a \oplus b \oplus c \oplus d \dots = (((a \oplus b) \oplus c) \oplus d) \oplus \dots$$

Obecný princip ukazuje obr. 9.7 vlevo, konkrétní příklady zapojení s vazbou ze dvou a ze čtyř klopných obvodů ukazují obrázky vpravo.



Obr. 9.7 Zpětná vazba přes obvody XOR

Celé toto zapojení se nazývá **lineární čítač** a patří do kategorie lineárních sekvenčních obvodů. Jejich charakteristickou vlastností je to, že jsou sestaveny výhradně ze tří typů součástek – z klopných obvodů D, členů XOR a z obvodů pro násobení konstantou (ty se však v lineárním čítači nevyskytují).

Lineární čítač prochází cyklicky posloupností stavů, jejíž délka je vždy menší nebo rovna $2^N - 1$. Její délka závisí na volbě odkoček posuvného registru, využitých pro zpětnou vazbu. Platí pravidlo, že pro libovolnou délku posuvného registru lze vždy nalézt alespoň jednu kombinaci odkoček tak, že posloupnost stavů je dlouhá právě $2^N - 1$. Je to tzv. **posloupnost maximální délky**. Tím se lineární čítač z hlediska úspornosti využití klopných obvodů přibližuje binárnímu čítači s 2^N stavů. Obvody XOR ve zpětné vazbě tvoří vlastně generátor liché parity. Jediný stav, který v posloupnosti maximální délky chybí, je stav 00...0, neboť v tom případě zpětnovazební obvod vydá nulu a po posunu obsahu registru je v něm opět stav 00...0. Pro správnou funkci lineárního čítače je po náběhu napájení nutné **nastavit počáteční stav** registru jiný, než 00...0.

Dále je uvedena tab. 9.1, která udává funkci ve zpětné vazbě pro registry různých délek. Q_0 značí prvý klopný obvod (nejblíže k sériovému vstupu).

Tab 9.1 Pozice odkoček pro posloupnost maximální délky

N	výraz pro ZV
2	$Q_0 \oplus Q_1$
3	$Q_1 \oplus Q_2$
4	$Q_2 \oplus Q_3$
5	$Q_2 \oplus Q_4$
6	$Q_4 \oplus Q_5$
7	$Q_5 \oplus Q_6$
8	$Q_3 \oplus Q_4 \oplus Q_5 \oplus Q_7$
9	$Q_4 \oplus Q_8$
10	$Q_6 \oplus Q_9$
11	$Q_8 \oplus Q_{10}$
12	$Q_5 \oplus Q_7 \oplus Q_{10} \oplus Q_{11}$
.....	

Jako příklad je dále ukázána posloupnost stavů pro $N = 4$ a počáteční stav 0001. Pořadí klopných obvodů je $Q_0 Q_1 Q_2 Q_3$:

0001 → 1000 → 0100 → 0010 → 1001 → 1100 → 0110 → 1011 → 0101 → 1010 → 1101 → 1110 → 1111 → 0111 → 0011



Ve stavu 0001 je $Q_2 = 0$ a $Q_3 = 1$, tedy $Q_2 \oplus Q_3 = 1$ a jednička vstupuje zleva do posuvného registru na pozici Q_0 . Další stav po posuvu vpravo je tedy 1000. Podobně pro další stavy. Pokud se na stav registru díváme jako na dekadické číslo (Q_0 = MSB), dostaneme sekvenci:

1 → 8 → 4 → 2 → 9 → 12 → 6 → 11 → 5 → 10 → 13 → 14 → 15 → 7 → 3



Zajímavý je i průběh signálu na výstupu kteréhokoli klopného obvodu. Na Q_0 je to takto:

010011010111100 a opakováně 010.....

Tento průběh se přibližuje průběhu náhodného signálu – je ovšem periodický, tedy nikoliv náhodný, ale pseudonáhodný. Generace pseudonáhodných posloupností je jednou z aplikací lineárních čítačů.

Schéma lineárního čítače je extrémně jednoduché, což je jeho výhodou a přispívá i k jeho rychlosti.

Následující výpis kódu ukazuje realizaci čtyřbitového lineárního čítače v jazyce VHDL pomocí procesu. Posun registru nastává při vzestupné hraně na signálu *clk*, přičemž směr posuvu je vpravo a zleva se nasouvá do registru hodnota vypočtená zpětnovazební funkcí $Q_2 \oplus Q_3$. V případě, že signál *reset* má hodnotu log. 1 je registr nulován.

```

library ieee;
use ieee.std_logic_1164.all;

entity lsfr is
  port (
    reset : in std_logic;
    clk : in std_logic;
    q : out std_logic_vector (0 to 3)
  );
end lsfr;

architecture struct of lsfr is
  signal reg : std_logic_vector (0 to 3);
begin
  process (reset, clk)
  begin
    -- asynchronni reset
    if reset = '1' then
      reg <= "0001";
    -- realizace lsfr citace
    elsif clk'event and clk = '1' then
      reg <= (reg(2) xor reg(3)) & reg(0 to 2);
    end if;
  end process;
  q <= reg;
end struct;

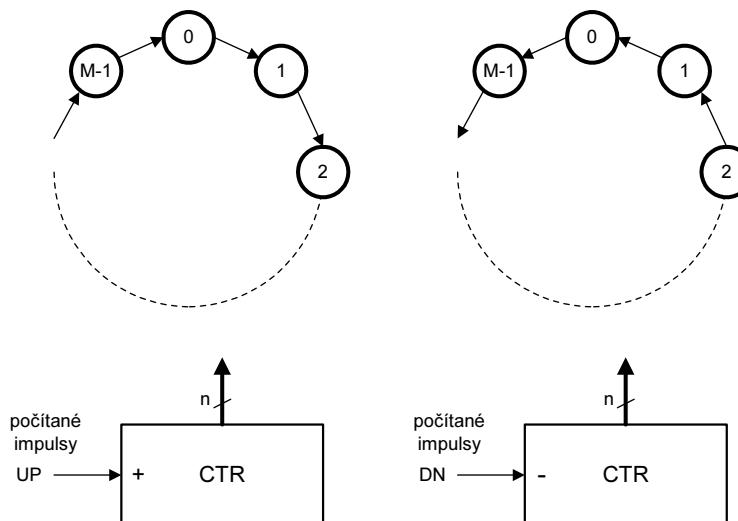
```

Výpis 9.6 Čtyřbitový lineární čítač

9.4 Charakteristika čítačů

Čítače slouží k počítání impulzů. Stavem čítače je soubor hodnot n výstupních signálů. Čítač při každém vstupním impulzu změní stav a postupně tak prochází cyklem M stavů. Po M impulzech se cyklus opakuje, ale počet opakování již nelze samotným čítačem zjistit. Během cyklu se žádné dva stavы neopakují. Obr. 9.8 ukazuje jednoduchý graf, popisující přechody mezi stavы čítače. Stavy jsou v obrázku označeny čísly. Každému stavu odpovídá jedna kombinace hodnot výstupních signálů. Tato kombinace hodnot může být chápána jako binární číslo, či jeho dekadický ekvivalent. Pak je zcela přirozené přiřadit tato čísla stavům čítače tak, že s každým vstupním impulzem číslo roste – čítač **počítá nahoru**. Graf přechodů takového čítače je na obr. 9.8 vlevo, pod ním je symbol čítače se zkratkou CTR (z angl.

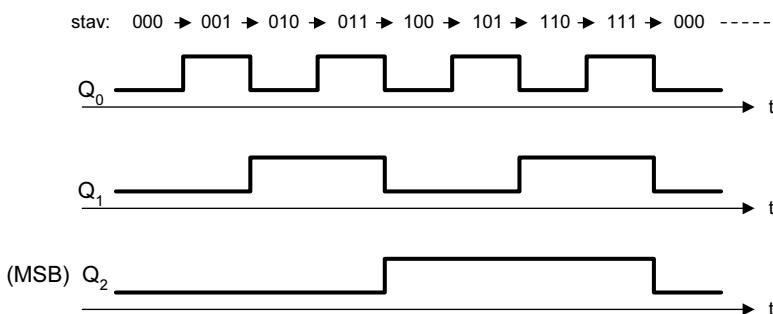
Counter) – jeho vstup je označen *UP*. Druhá samozřejmá možnost je seřadit stavy v sestupném pořadí – čítač **počítá dolů** (na obr. 9.8 vpravo, vstup označen *DN* z anglického *Down*). Přiřazení však může být i jiné, např. existují čítače počítající v **Grayově kódu** (kód se změnou v jedné proměnné).



Obr. 9.8 Graf přechodů čítače a jeho symbolická značka

Kromě čítačů počítajících jedním směrem existují i čítače umožňující počítání nahoru i dolů, tzv. čítače **vratné** neboli **reverzivní**. Směr počítání lze řídit dvěma způsoby. V prvním případě se směr řídí zvláštním signálem, ve druhém případě existují dva vstupy pro počítané impulzy – při impulzech na jednom (*UP*) čítač počítá nahoru, při impulzech na druhém (*DN*) počítá dolů.

Čítač, který počítá v binárním kódu a s cyklem o délce $M = 2^k$, se nazývá **binární**. Čítač s cyklem $M=10$ se nazývá **dekadický**, počítá v kódu BCD (Binary Coded Decimal). Pro čítače s cykly jiných délek se používá název „čítač **modulo M**“.



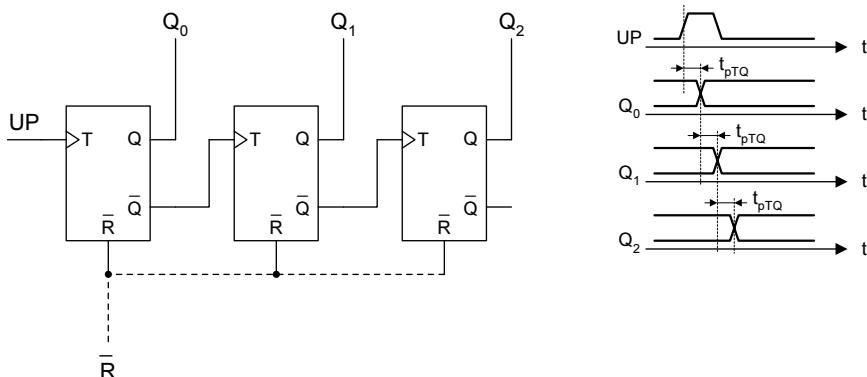
Obr. 9.9 Průběh signálů binárního čítače

Posloupnost stavů u binárního čítače s počítáním nahoru je ukázána na příkladu tříbitového čítače. Pro kódování 8 stavů jsou zapotřebí 3 dvojhodnotové signály Q_2 , Q_1 , Q_0 . Seřazeny v tomto pořadí (tj. Q_2 = MSB, nejvýznamnější bit) budou vyjadřovat binární číslo o třech bitech. *Obr. 9.9* ukazuje střídání stavů vyvolané vstupními impulzy.

Zřejmě kmitočet impulzů na Q_0 je polovinou kmitočtu vstupních impulzů, na Q_1 je jeho čtvrtinou, na Q_2 je jeho osminou, atd. Čítač tedy lze využít jako **dělič kmitočtu**.

9.5 Asynchronní čítače

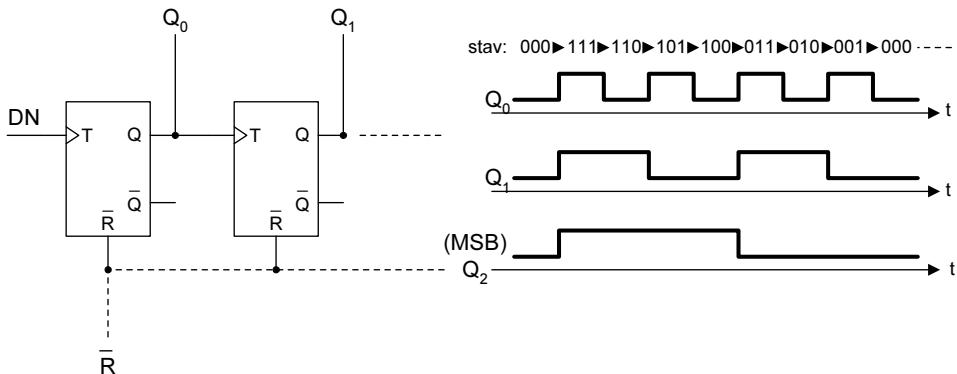
Čítač lze realizovat velmi jednoduchým spojením asynchronních klopových obvodů T . Pro čítač z předchozího příkladu platí *obr. 9.10*.



Obr. 9.10 Tříbitový asynchronní čítač a zpoždění jednotlivých signálů

Pro požadovaný průběh signálů je nutné, aby se první klopový obvod překlápel při každém impulzu UP , druhý KO při překlopení prvého z 1 do 0, třetí KO při překlopení druhého z 1 do 0, atd. Pokud se klopové obvody T překlápejí na náběžnou hranu (jak je v obrázku vyznačeno), lze požadovaný průběh signálu docílit propojením vstupu T s Q předcházejícího KO. Pokud by byly použity klopové obvody překlápné doběžnou hranou, bylo by třeba spojit T s Q . Z časového diagramu je zřejmé, že překlápení probíhá postupně po klopových obvodech zleva doprava jako **vlna**, vždy se zpožděním t_{pTQ} . Nejdelší cesta je při přechodu ze stavu 11...11 do 00...00, kdy vlna proběhne všemi n klopovými obvody. Klopové obvody v tomto čítači se nepřeklápejí všechny současně – odtud vyplynává název **asynchronní čítač**. Doba na ustálení stavu (všech výstupních signálů) je dlouhá, její maximum je $n \cdot t_{pTQ}$. To je nevýhodou asynchronního čítače v aplikacích, vyžadujících krátkou dobu na ustálení stavu. Není to však na závadu v aplikaci jako **dělič kmitočtu**. Nejvyšší kmitočet překlápení musí mít jen první klopový obvod, druhý KO pracuje již jen s polovičním kmitočtem, atd.

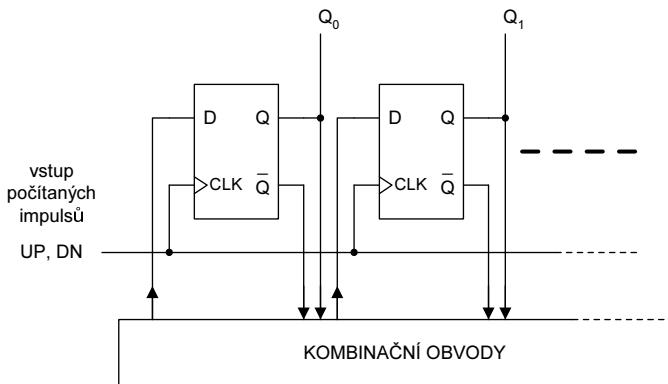
Jednoduchou změnou vazeb mezi klopovými obvody lze docílit počítání **dolů**. Požadovaný časový diagram a zapojení je na *obr. 9.11*.



Obr. 9.11 Vazby klopných obvodů pro počítání dolů

9.6 Synchronní čítače

U synchronních čítačů jsou všechny KO překlápeny současně. Obecně platné schéma ukazuje obr. 9.12. Vstupní (tj. počítané) impulzy jsou zavedeny do spojených vstupů hodinových impulzů KO. Vstupní signály pro klopné obvody jsou vytvářeny v řídicích (kombinačních) obvodech na základě stavu KO. To dává možnost použití libovolných typů KO (tj. nejen D podle obrázku, ale i T, JK, RS), vždy řízených **stejnou** hranou na vstupech CLK . Pro každý typ KO samozřejmě musí být navrženo jiné zapojení řídicích obvodů – někdy jednodušší a někdy složitější, ale vždy garantující požadovanou funkci. Volba vhodného typu KO proto nemá vliv na funkci celého čítače, ale na jeho **složitost**.



Obr. 9.12 Zjednodušené zapojení synchronního čítače

Vzhledem k současnému působení vstupních impulzů na všechny KO je **doba ustálení** nového stavu čítače dána bez ohledu na počet KO dobou zpoždění t_{pCQ} , tj. od hrany CLK do změny stavu na výstupu KO. Tato doba je pro všechny KO v čítači **téměř shodná** – malé odchylinky jsou důsledkem výrobních tolerancí. Pokud impulzy na vstupech CLK klopných

obvodů splňují podmínky kladené na strmost hran a délku impulzů, **nevznikají** na výstupech KO (a tedy na výstupech čítače) nikdy falešné impulzy.

Pokud však je čítač zdrojem signálů pro následující kombinační obvod, mohou odchylky v dobách zpoždění jednotlivých KO způsobit vznik falešného impulzu v tomto kombinačním obvodu. Jedná se o typický případ reakce obvodu na změny několika vstupních signálů. Tak např. při přechodu ze stavu 7 do 8 – binárně z 0111 do 1000 – se mění stavy všech KO, ale ne nutně absolutně současně. Mezi stavy 0111 a 1000 se tak na krátký okamžik mohou vyskytnout všechny možné kombinace hodnot. Na tyto „mezistavy“ může kombinační obvod reagovat vytvořením krátkého falešného impulzu.

Dále bude ukázáno řešení čtyřbitového binárního synchronního čítače, počítajícího nahoru. V **tabulce přechodů** (tab. 9.2) jsou vyjmenovány všechny stavy čítače a k nim následující stavy („starý“ stav a „nový“ stav). Stav čítače je dán kombinací hodnot signálů Q_3, Q_2, Q_1, Q_0 ($Q_3 = \text{MSB}$). Vpravo jsou stanoveny **budicí funkce** jednotlivých KO, potřebné k vyvolání daných přechodů. Uvažují se jednak klopné obvody T , jednak D .

Tab. 9.2 Tabulka přechodů a budicích funkcí čtyřbitového binárního čítače pro počítání nahoru

stav:		budicí funkce:							
starý	nový	T_3	T_2	T_1	T_0	D_3	D_2	D_1	D_0
$Q_3Q_2Q_1Q_0$	$\rightarrow Q_3Q_2Q_1Q_0$								
0 0 0 0	$\rightarrow 0 0 0 1$	0	0	0	1	0	0	0	1
0 0 0 1	$\rightarrow 0 0 1 0$	0	0	1	1	0	0	1	0
0 0 1 0	$\rightarrow 0 0 1 1$	0	0	0	1	0	0	1	1
0 0 1 1	$\rightarrow 0 1 0 0$	0	1	1	1	0	1	0	0
0 1 0 0	$\rightarrow 0 1 0 1$	0	0	0	1	0	1	0	1
0 1 0 1	$\rightarrow 0 1 1 0$	0	0	1	1	0	1	1	0
0 1 1 0	$\rightarrow 0 1 1 1$	0	0	0	1	0	1	1	1
0 1 1 1	$\rightarrow 1 0 0 0$	1	1	1	1	1	0	0	0
1 0 0 0	$\rightarrow 1 0 0 1$	0	0	0	1	1	0	0	1
1 0 0 1	$\rightarrow 1 0 1 0$	0	0	1	1	0	1	0	0
1 0 1 0	$\rightarrow 1 0 1 1$	0	0	0	1	1	0	1	1
1 0 1 1	$\rightarrow 1 1 0 0$	0	1	1	1	1	1	0	0
1 1 0 0	$\rightarrow 1 1 0 1$	0	0	0	1	1	1	0	1
1 1 0 1	$\rightarrow 1 1 1 0$	0	0	1	1	1	1	1	0
1 1 1 0	$\rightarrow 1 1 1 1$	0	0	0	1	1	1	1	1
1 1 1 1	$\rightarrow 0 0 0 0$	1	1	1	1	0	0	0	0

Klopný obvod **typu T** musí mít na vstupu hodnotu 1, má-li se měnit stav na jeho výstupu Q při impulzu CLK . Tak pro Q_3 platí, že má měnit svůj stav, jen je-li „starý“ stav čítače $Q_3Q_2Q_1Q_0 = 0111$ a též 1111. Proto T_3 musí mít hodnotu 1 právě v těchto stavech čítače, v ostatních musí mít hodnotu 0. T_3 tedy závisí na výstupních signálech **všech** KO, jinak řečeno je funkcí proměnných Q_3, Q_2, Q_1, Q_0 . Jedná se o kombinační funkci a tu lze nalézt jako součet mintermů, který se dále upraví. Dostáváme:

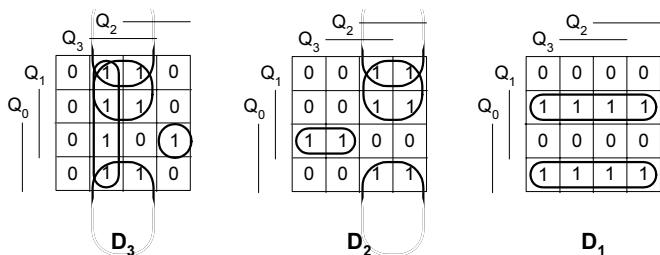
$$T_3 = \overline{Q_3}Q_2Q_1Q_0 + Q_3\overline{Q_2}Q_1Q_0 = Q_2Q_1Q_0$$

Obdobně získáme:

$$T_2 = \overline{Q_3}\overline{Q_2}Q_1Q_0 + \overline{Q_3}Q_2\overline{Q_1}Q_0 + Q_3\overline{Q_2}\overline{Q_1}Q_0 + Q_3Q_2Q_1Q_0 = \dots = Q_1Q_0$$

Z tabulky zjistíme, že T_1 má mít hodnotu 1 vždy, když je $Q_0=1$, tedy:

$$T_1 = Q_0$$



Obr. 9.13 Mapy pro nalezení budicích funkcí D čtyřbitového čítače

Nakonec zjistíme, že T_0 má mít hodnotu 1.

$$T_0 = 1$$

Klopý obvod **typu D** musí mít na vstupu D takovou hodnotu, kterou má nabýt výstup Q po impulzu CLK . Proto ve sloupci pro budicí funkci D_3 se opakuje sloupec pro D_3 v „novém“ stavu. Opět platí, že D_3 je funkcí proměnných Q_3, Q_2, Q_1, Q_0 (viz sloupec „starý“ stav). Pro změnu budeme hledat budicí funkce D v mapách – viz obr. 9.13.

Ze smyček v mapách vyčteme:

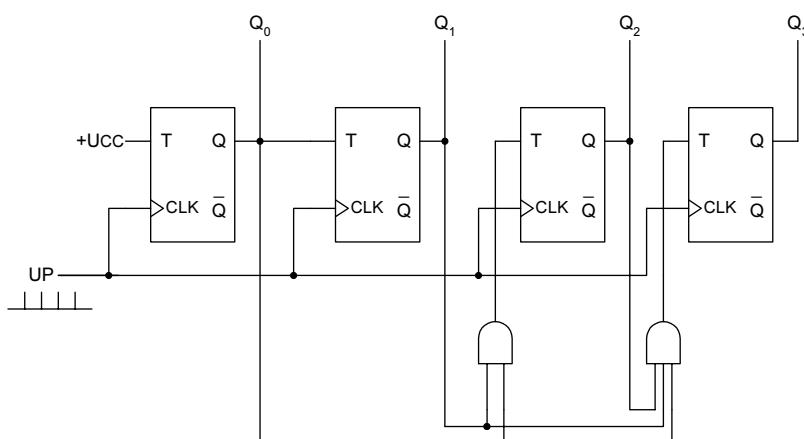
$$D_3 = Q_3\bar{Q}_2 + Q_3\bar{Q}_1 + Q_3\bar{Q}_0 + \bar{Q}_3Q_2Q_1Q_0$$

$$D_2 = Q_2\bar{Q}_1 + Q_2\bar{Q}_0 + \bar{Q}_2Q_1Q_0$$

$$D_1 = Q_1\bar{Q}_0 + \bar{Q}_1Q_0$$

$$D_0 = \bar{Q}_0$$

Funkci D_0 nebylo nutné řešit, neboť z tabulky lze rovnou vyčíst, že sloupec pro D_0 je negací sloupců pro Q_0 ve „starém“ stavu.



Obr. 9.14 Výsledné schéma čtyřbitového binárního čítače

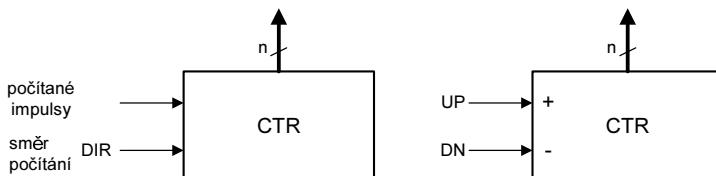
Srovnáním budicích funkcí T a D zjistíme, že v tomto případě bude čítač s KO typu T **jednodušší**. Jeho schéma je na obr. 9.14.

Stejným způsobem by se navrhoval čítač počítající dolů, samozřejmě s jinou tabulkou přechodů – viz tab. 9.3.

Tab. 9.3 Tabulka přechodů binárního čítače pro počítání dolů

stav:	
starý	nový
$Q_3Q_2Q_1Q_0$	$Q_3Q_2Q_1Q_0$
<hr/>	
0 0 0 0	-> 1 1 1 1
0 0 0 1	-> 0 0 0 0
0 0 1 0	-> 0 0 0 1
0 0 1 1	-> 0 0 1 0
.....
1 1 1 1	-> 1 1 1 0

V synchronní verzi lze realizovat čítač umožňující počítání nahoru i dolů, tzv. čítač **vratný** neboli **reverzivní**. Z hlediska vnějších signálů lze směr počítání řídit dvěma způsoby – viz obr. 9.15. V prvním případě se směr řídí zvláštním signálem, ve druhém případě existují dva vstupy pro počítané impulzy – při impulzech na jednom čítač počítá nahoru, při impulzech na druhém počítá dolů.



Obr. 9.15 Dvě varianty reverzivního čítače

Následující výpis kódu ukazuje realizaci n-bitového vratného čítače v jazyce VHDL pomocí procesu. Proces má v citlivostním seznamu uvedeny signály *reset* a *clk*. V případě, že signál *reset* má hodnotu log. 1 je čítač nulován. Čítač čítá na vzestupnou hranu na signálu *clk*, přičemž směr čítání určuje hodnota signálu *dir*. Počet bitů registru lze opět jednoduše měnit změnou hodnoty parametru *n* v generické části deklarace entity.

```

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

entity vratny is
    generic (
        n : positive := 8
    );
    port (
        reset : in std_logic;
        clk : in std_logic;
        dir : in std_logic;
        q : out std_logic_vector (n - 1 downto 0)
    );
end vratny;

```

```

architecture struct of vratny is
begin
    -- vratny citac
    process (reset, clk)
        variable cnt : unsigned (n - 1 downto 0);
    begin
        -- asynchronni reset
        if reset = '1' then
            cnt := (others => '0');
        -- synchronni citac
        elsif clk'event and clk = '1' then
            if dir = '1' then
                cnt := cnt + 1;
            else
                cnt := cnt - 1;
            end if;
        end if;
        -- konverze na typ std_logic_vector
        q <= std_logic_vector (cnt);
    end process;
end struct;

```

Výpis 9.7 Synchronní reverzivní čítač

9.7 Nulování a přednastavení čítače

Před začátkem počítání impulzů je u čítačů zpravidla nutné nastavit **počáteční stav**. Výjimkou může být snad jen aplikace jako dělič kmitočtu, kdy počáteční stav nemá smysl. Jako počáteční stav je někdy postačující **vynulování**, jindy je však nutné přednastavit v čítači stav jiný než nula. Existují dvě varianty přednastavení – asynchronní a synchronní. V obou případech se vložení počátečního stavu vyvolá řídicím signálem (nazvaným *PE* – Preload Enable, *LD* – Load, aj.). Data pro přednastavení jsou přiváděna na vstupy D_i .

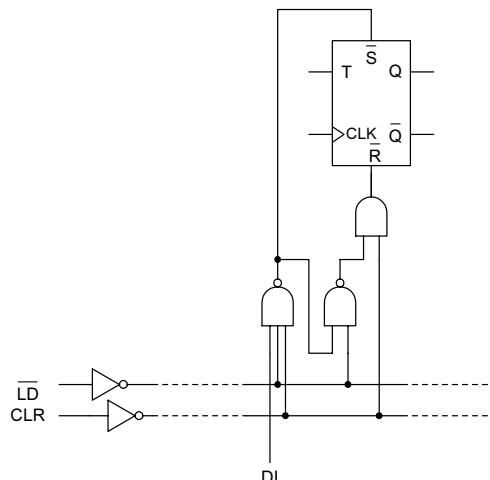
Asynchronní přednastavení (příp. nulování) má přednost před všemi ostatními vstupními signály a je možné v kterémkoliv okamžiku. Potřebné obvody pro asynchronní přednastavení jednoho KO ukazuje obr. 9.16. Stejné obvody by byly doplněny i u ostatních KO.

Signál $\overline{LD} = 0$ při $CLR = 0$ vyvolá zápis dat DI do klopného obvodu – při $DI = 1$ bude aktivní signál \overline{S} , takže $Q = 1$. Při $DI = 0$ bude naopak aktivní signál \overline{R} , takže $Q = 0$. Vstup nulování CLR však má ve stavu 1 vždy prioritu bez ohledu na signál \overline{LD} . Vstup nulování může být případně v negaci.

Synchronní přednastavení je u čítače obdobné jako u posuvného registru podle obr. 9.4. Do vstupu klopného obvodu (D, T) se zařadí multiplexor, přepínaný signálem LD . Pro vložení dat je nutné využít jeden impulz CLK , tedy u synchronního čítače jeden vstupní impulz. Pak se multiplexor přepne.

Následující výpis kódu ukazuje realizaci n -bitového čítače s asynchronním nulováním a přednastavením v jazyce VHDL pomocí procesu. Proces má v citlivostním seznamu uveden signál clk , clr a n_ld . Podmínka realizující asynchronní nulování registru je uvedena před asynchronní podmínkou realizující přednastavení a synchronní podmínkou realizující

vlastní čítač. Čítání probíhá při vzestupné hraně signálu *clk*, pokud není aktivní signál *clr* a *n_ld*. V případě, že je aktivní signál *n_ld* je do čítače zapsána hodnota z *d*. Počet bitů registru lze opět jednoduše měnit při mapování změnou hodnoty parametru *n* v generické části.



Obr. 9.16 Obvody pro asynchronní přednastavení a nulování čítače

```

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

entity citac is
  generic (
    n : positive := 8
  );
  port (
    clk : in std_logic;
    clr : in std_logic;
    n_ld : in std_logic;
    d : in std_logic_vector (n - 1 downto 0);
    q : out std_logic_vector (n - 1 downto 0)
  );
end citac;

architecture struct of citac is
begin
  -- citac vpred s asynchronnim nulovanim a prednastavenim
  process (clk, clr, n_ld)
    variable cnt : unsigned (n - 1 downto 0);
  begin
    -- asynchronni reset
    if clr = '1' then
      cnt := (others => '0');
    end if;
    if rising_edge(clk) and n_ld = '0' then
      cnt := d;
    end if;
    report "Value of counter is " & integer'image(cnt);
    loop
      if falling_edge(clk) then
        report "Value of counter is " & integer'image(cnt);
        exit;
      end if;
      if cnt = n - 1 then
        report "Counter overflow";
        exit;
      end if;
      cnt := cnt + 1;
    end loop;
  end process;
end;
  
```

```

-- asynchronni prednastaveni
elsif n_ld = '0' then
    cnt := unsigned (d); -- konverze na typ unsigned
-- synchronni citac
elsif clk'event and clk = '1' then
    cnt := cnt + 1;
end if;
-- konverze na typ std_logic_vector
q <= std_logic_vector (cnt);
end process;
end struct;

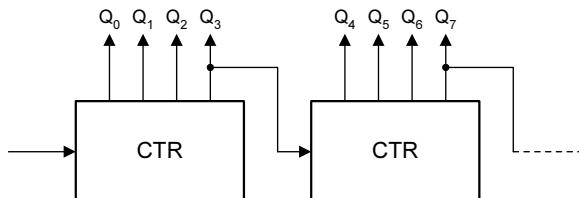
```

Výpis 9.8 Synchronní čítač vpřed s asynchronním nulováním a přednastavením

9.8 Přenosy z čítače

Přenos z čítače je výstupní signál informující o přechodu do stavu 00...00 při počítání nahoru, nebo ze stavu 00...00 při počítání dolů. Tak např. u čtyřbitového čítače počítajícího nahoru vzniká přenos při přechodu ze stavu 1111 do stavu 0000, u čítače počítajícího dolů vzniká přenos při přechodu ze stavu 0000 do stavu 1111. Přenos je využíván pro spojení s dalšími (vyššími) řady čítače s velkým počtem bitů. Takový čítač je složen z menších stavebních jednotek – dílčích čítačů – o menším počtu bitů, spojených do kaskády. Přenos ale je často využíván i v jiných obvodech, do kterých je čítač začleněn ... např. časovačích, filtroch, modulátorech, atd.

Nejjednodušší je **asynchronní přenos**, spočívající pouze ve vyvedení výstupního signálu z nejvyššího klopného obvodu. Předpokládá se, že následující obvody budou reagovat na hranu tohoto signálu. U čítače s počítáním nahoru je to doběžná hrana. Obr. 9.17 ukazuje asynchronní přenos jako vazbu do vyšších řad čítače, složeného z jednotek o čtyřech bitech.



Obr. 9.17 Asynchronní přenos z čítače

Asynchronní přenos prodlužuje dobu ustálení stavu celého čítače, neboť musí postupně procházet jednotlivými stupni.

Následující výpis kódu ukazuje realizaci osmibitového čítače sestávajícího ze dvou čtyřbitových čítačů navázaných na sebe asynchronním přenosem pomocí dvou procesů. První proces realizující první čtyřbitový čítač má v citlivostním seznamu uveden signál *clk*, druhý proces má pak v citlivostním seznamu uveden nejvyšší bit prvního čítače – tj. signál *q(3)*. Oba čítače čítají na sestupnou hranu. Čítače jsou inicializovány pomocí signálu *reset* na hodnotu "0000".

```

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

entity async_prenos is
  port (
    reset : in std_logic;
    clk : in std_logic;
    q : buffer std_logic_vector (7 downto 0)
  );
end async_prenos;

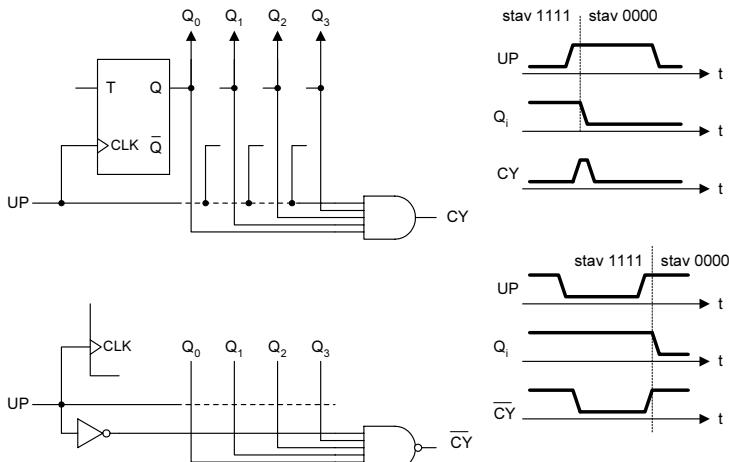
architecture behav of async_prenos is
begin
  -- prvni citac
  process (reset, clk)
    variable cnt : unsigned (3 downto 0);
  begin
    -- asynchronni reset
    if reset = '1' then
      cnt := "0000";
    -- cita na sestupnou hranu clk
    elsif clk'event and clk = '0' then
      cnt := cnt + 1;
    end if;
    -- konverze na typ std_logic_vector
    q(3 downto 0) <= std_logic_vector (cnt);
  end process;

  -- druhý citac
  process (reset, q(3))
    variable cnt : unsigned (3 downto 0);
  begin
    -- asynchronni reset
    if reset = '1' then
      cnt := "0000";
    -- cita na sestupnou hranu q3
    elsif q(3)'event and q(3) = '0' then
      cnt := cnt + 1;
    end if;
    -- konverze na typ std_logic_vector
    q(7 downto 4) <= std_logic_vector (cnt);
  end process;
end behav;

```

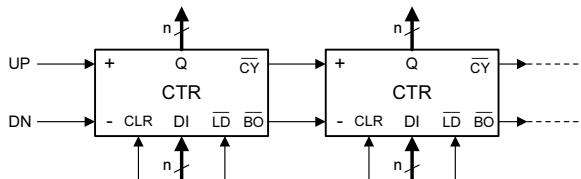
Výpis 9.9 Dva čtyřbitové čítače v kaskádě s asynchronní vazbou

Pro **synchronní přenos** je v čítači vytvořen samostatný výstup, na kterém je při patřičném přechodu vydán krátký impulz. Princip ukazuje obr. 9.18. Nahoře je ukázán obvod pro generaci přenosového impulzu CY (angl. *carry*). Při stavu čítače 1111 a $UP = 1$ bude $CY = 1$, ale současně již čítač přechází do stavu 0000, takže impulz CY skončí. Je tedy velmi krátký, což může být na závadu při jeho zpracování dalšími obvody. Proto se využívá jiné zapojení podle obrázku dole. Stejně jako u obvodu nahoře zde náběžná hrana CY signalizuje změnu stavu čítače, ale přenosový impulz je vytvářen již v předstihu, během stavu 1111, a je mnohem delší.



Obr. 9.18 Obvody pro generaci synchronního přenosu z čítače

V případě čítače s počítáním dolů zůstává princip stejný, jen místo stavu 11...11 je rozšířen stav 00...00 – do součinu jsou zavedeny negované výstupy KO. Výstupní signál je pak označen jako BO (angl. borrow – „výpůjčka“, záporný přenos). U reverzivních čítačů jsou generovány CY i BO. Další obr. 9.19 ukazuje spojení reverzivních čítačů do kaskády se všemi již dříve zmíněnými vstupy a výstupy.



Obr. 9.19 Přenosy u reverzivních čítačů

9.9 Čítače „modulo M“

Čítač modulo M má proti binárnímu čítači **zkrácený** cyklus. To lze docílit dvěma způsoby:

- Při návrhu čítače vhodně definovat tabulku přechodů a odvodit budicí funkce KO. To je možné jen u čítačů synchronních.
- Využít z čítače binárního a doplnit jej o obvody, které čítač vynuluje při dosažení vhodného stavu, dříve než na konci cyklu. To je možné u čítačů asynchronních i synchronních.

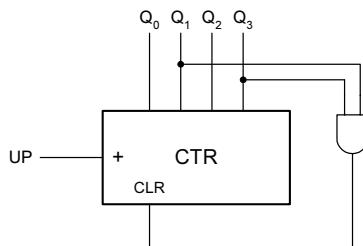
První případ je ilustrován na příkladu čítače dekadického, synchronního – viz tab. 9.4.

Jelikož nejsou využity stavy čítače 1010 až 1111, nebudou budicí funkce T definovány pro tyto stavy a tudíž v nich budou existovat neurčené stavy. Pro takto jednoduchou úlohu lze využít metodu map.

Druhý případ bude ilustrován opět na dekadickém čítači, jehož vnitřní zapojení zde není podstatné – viz obr. 9.20.

Tab. 9.4 Tabulka přechodů dekadického čítače (modulo 10)

stav:					
starý	nový	T ₃	T ₂	T ₁	T ₀
Q ₃ Q ₂ Q ₁ Q ₀	→ Q ₃ Q ₂ Q ₁ Q ₀				
0 0 0 0	→ 0 0 0 1	0	0	0	1
0 0 0 1	→ 0 0 1 0	0	0	1	1
0 0 1 0	→ 0 0 1 1	0	0	0	1
0 0 1 1	→ 0 1 0 0	0	1	1	1
0 1 0 0	→ 0 1 0 1	0	0	0	1
0 1 0 1	→ 0 1 1 0	0	0	1	1
0 1 1 0	→ 0 1 1 1	0	0	0	1
0 1 1 1	→ 1 0 0 0	1	1	1	1
1 0 0 0	→ 1 0 0 1	0	0	0	1
1 0 0 1	→ 0 0 0 0	1	0	0	1



Obr. 9.20 Zkrácení cyklu čítače vynulováním

Při kombinaci hodnot $Q_3 = 1$, $Q_1 = 1$, ke které dojde nejdříve ve stavu 1010 (dekadicky 10), se čítač vynuluje a tím nulování skončí. Stav, při kterém bylo vyvoláno nulování, tedy trvá jen krátkou dobu. Čítač prochází stavy 0 až 9, a ze stavu 9 do stavu 0 po velmi krátkém výskytu stavu 10. V cyklu je tak deset stabilních stavů.

Následující výpis kódu ukazuje realizaci čtyřbitového čítače modulo 10 v jazyce VHDL pomocí procesu. Čítač je inicializován pomocí signálu *reset* na hodnotu 0 a čítá na vzestupnou hranu signálu *clk*.

```

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

entity modulo10 is
    port (
        reset : in std_logic;
        clk : in std_logic;
        q : out std_logic_vector (3 downto 0)
    );
end modulo10;

architecture struct of modulo10 is
begin
    process (reset, clk)
        variable cnt : unsigned (3 downto 0);
    begin
        if reset = '1' then
            cnt := (others => '0');
        elsif rising_edge(clk) then
            if cnt = 9 then
                cnt := (others => '0');
            else
                cnt := cnt + 1;
            end if;
        end if;
        q := std_logic_vector(cnt);
    end process;
end;

```

```

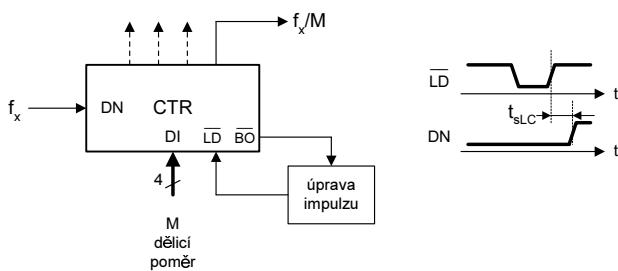
begin
    -- asynchronni reset
    if reset = '1' then
        cnt := (others => '0');
    elsif clk'event and clk = '1' then
        -- realizace citani modulo 10
        if cnt = 9 then
            cnt := (others => '0');
        else
            cnt := cnt + 1;
        end if;
    end if;
    -- konverze na typ std_logic_vector (3 downto 0)
    q <= std_logic_vector (cnt);
end process;
end struct;

```

Výpis 9.10 Čítač modulo 10

Čítače mod M se často využívají jako **děliče kmitočtu** celým číslem M . Pro tuto aplikaci není podstatné, kterými stavami čítač prochází – jen jich musí být M v cyklu a jednou za cyklus musí být generován výstupní impulz. Dělicí poměr je však často třeba měnit (programovat), což u předchozího zapojení nelze.

Jiné zapojení, tentokráte s čítačem počítajícím dolů, ukazuje obr. 9.21. Předpokládá se čítač s asynchronním předenastavením. Ve stavu 0000 se při $DN = 0$ generuje přenosový impulz \overline{BO} . Vazba z \overline{BO} na \overline{LD} způsobí nastavení stavu M ještě během stavu 0000. Při náběžné hraně na vstupu DN se odečte jednička, následující stav bude tedy $M-1$. Dohromady je v cyklu M stavů. V obrázku je vyznačena úprava impulzů, která musí podle potřeby zajistit správné časování. Synchronní čítače totiž vyžadují ukončení aktivního stavu LD minimálně o dobu předstihu t_{SLC} před aktivní hranou DN , jak ukazuje časový diagram.



Obr. 9.21 Programovatelný dělič kmitočtu

426410

A large, stylized number '10' is centered on the page. The '1' is a vertical line with a small diagonal stroke at the top. The '0' is a thick circle. The entire graphic is rendered in a light gray color against a white background.

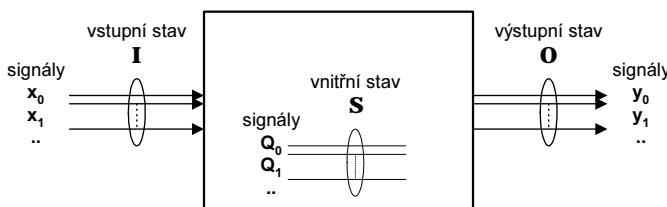
SEKVENČNÍ OBVODY

Základní vlastností sekvenčních obvodů, kterou se liší od obvodů kombinačních je to, že na jednu a tutéž kombinaci hodnot vstupních signálů může obvod reagovat po každé jinak. Výstupní signály sekvenčního obvodu jsou tedy závislé nejen na signálech vstupních v daném okamžiku, ale též na vstupních signálech **předcházejících**. Jinými slovy – sekvenční obvod má **vnitřní paměť**.

10.1 Přechodová a výstupní funkce

Abstraktním modelem sekvenčního obvodu je **konečný automat** – má konečný počet vstupních stavů, výstupních stavů a vnitřních stavů. Jako stav je rozuměna kombinace hodnot signálů. V technické praxi pracujeme se signály dvojhodnotovými (0 a 1), kterých je vždy konečný počet. Proto i počet možných kombinací hodnot těchto signálů je konečný. Každou z kombinací označíme kvůli úspornému zápisu symbolem – např. písmenem.

Obr. 10.1 ukazuje automat se vstupními signály x_i , výstupními signály y_i a vnitřními signály Q_i . Vstupní stav je označen jako I , výstupní jako O a vnitřní jako S . Jednotlivé konkrétní kombinace hodnot vstupních signálů označíme jako I_0, I_1 , atd. Obdobně budou označeny kombinace hodnot výstupních signálů jako O_0, O_1 , atd., a vnitřních signálů jako S_0, S_1 , atd.



Obr. 10.1 Signály a stavы v sekvenčním obvodu

Závislost následujícího vnitřního stavu na současném vnitřním stavu a vstupním stavu vyjadřuje **přechodová funkce**:

$$S^{t+1} = f(S^t, I^t),$$

kde symbol S^{t+1} značí vnitřní stav následující, S^t značí vnitřní stav současný, I^t značí vstupní stav současný.

Výstupní stav konečného automatu může být generován několika způsoby. Každý z nich je definován příslušnou **výstupní funkcí**. V nejobecnějším případě je výstupní stav funkcí vstupního stavu a vnitřního stavu:

$$O^t = g(S^t, I^t)$$

Shodné indexy t u všech proměnných znamenají, že výstupní stav je dán vnitřním stavem a vstupním stavem ve stejném okamžiku, a tedy se jedná o kombinační funkci. Automat s takto definovanou výstupní funkcí se nazývá **Mealyho automat**.

Další možnost generace výstupního stavu je definována výstupní funkcí, závislou jen na vnitřním stavu:

$$O^t = g(S^t)$$

Zřejmě se opět jedná o kombinační funkci. Automat s takto definovanou výstupní funkcí se nazývá **Mooreho automat**. Na první pohled by se mohlo zdát, že u Mooreho automatu chybí závislost výstupů na minulosti. Závislost zde ale je, a to prostřednictvím vnitřního stavu, který je závislý na minulých vstupních stavech.

Naopak závislost výstupního stavu jen na současném vstupním stavu není přípustná, neboť by se již nejdalo o sekvenční systém – byl by to zřejmě systém kombinační. Zvláštním případem je **autonomní automat**. Jeho přechodová funkce je:

$$S^{t+1} = f(S^t)$$

Následující vnitřní stav nezávisí na vstupním stavu. Systém se zcela obejde bez vstupních signálů. Je to např. případ čítačů.

Je zřejmé, že mezi současným a následujícím vnitřním stavem musí existovat časový odstup. Ten může být důsledkem zpoždění vnitřních obvodů, které vytvářejí stavové signály. Automaty tak mění svůj vnitřní stav (s malým zpožděním) po změně vstupního stavu. Automaty, které reagují tímto způsobem, se nazývají **asynchronní**. Zpoždění lze zavést i tak, že stavové signály jsou generovány klopnými obvody. Jejich **synchronizační impulzy** (nebo hodinové impulzy) pak udávají okamžik změny vnitřního stavu. Automaty, které reagují tímto způsobem, se nazývají **synchronní**. U nich je časování vnitřních signálů daleko lépe kontrolováno. Automaty jsou proto navrhovány převážně jako synchronní.

Pro stručnost se v následujícím textu bude používat slovo „stav“ s významem „vnitřní stav“, slovo „vstup“ s významem „vstupní stav“ a slovo „výstup“ s významem „výstupní stav“.

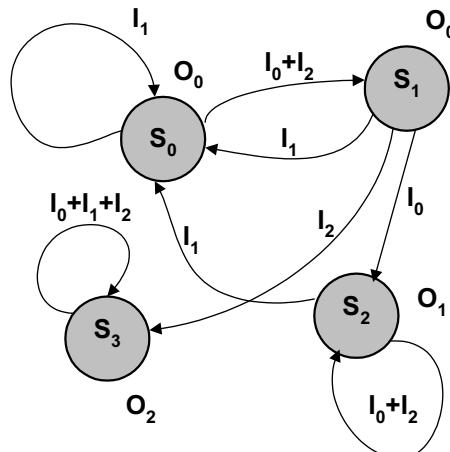
Pro konkrétní návrh sekvenčního obvodu je nutné nejprve vhodným způsobem popsat jeho činnost. Nejpřirozenějším způsobem popisu by se zdálo vyjmenování všech posloupností vstupů a jim odpovídajících posloupností výstupů. Je však třeba si uvědomit, že přestože konečný automat má konečný počet vstupních, výstupních i vnitřních stavů, může docela dobře zpracovávat posloupnosti **nekonečné délky**. Na délku vstupní posloupnosti totiž nejsou obecně kladena žádná omezení. Takovýto popis je tedy naprostě nepraktický a použitelný jen ve zcela výjimečných případech. Použitelné jsou následující způsoby popisu:

- Popis pomocí grafických prostředků, jako jsou grafy a vývojové diagramy.
- Popis soustavou rovnic.
- Popis tabulkami.
- Popis v některém programovacím jazyku.

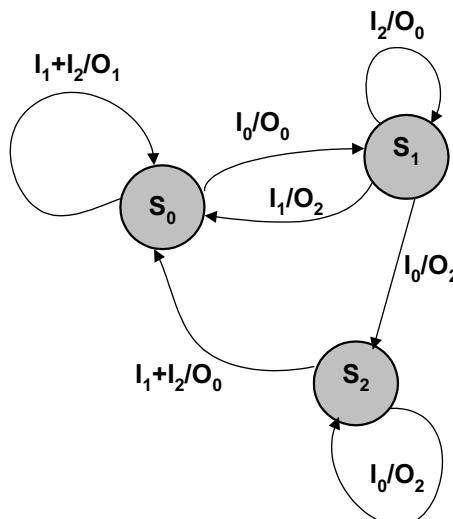
10.2 Popis pomocí grafu

Orientovaný graf vyjadřuje přehledně **přechodovou** funkci i **výstupní** funkci. Orientované grafy používané pro popis automatů se nazývají **stavové diagramy** (angl. *state*

diagrams). Z uzlu S_i do uzlu S_j vede hrana označená I_k , jestliže stav S_i při vstupu I_k přechází do stavu S_j . Výstupy u Mealyho automatu se vyznačují u jednotlivých hran grafu (přechodů), neboť výstup závisí na stavu a vstupu. Vstupní stav odpovídá příslušné hraně. Naopak u Mooreho automatu se výstupy vyznačují u uzlů grafu (stavů), neboť výstup závisí jen na stavu. Obr. 10.2 ilustruje případ Mooreho automatu, obr. 10.3 případ Mealyho automatu.



Obr. 10.2 Graf Mooreho automatu



Obr. 10.3 Graf Mealyho automatu

V obrázcích jsou ilustrovány různé možnosti tvaru grafů. Vstupy spojené znaménkem „+“ u některých hran znamenají, že dva nebo více vstupních stavů vyvolají při daném vnitřním stavu tentýž přechod. U Mooreho automatu jsou výstupy uvedeny u stavů, u Mealyho

automatu jsou uvedeny u jednotlivých hran za lomítkem. Tak např. na obr. 10.2 je ve stavu S_1 vydáván výstup O_0 a při vstupu I_2 automat přejde do stavu S_3 , ve kterém bude pak vydáván výstup O_2 . Na obr. 10.3 je ve stavu S_1 při vstupu I_0 vydáván výstup O_2 , při vstupu I_1 výstup O_2 a při vstupu I_2 výstup O_0 . Ze stavu S_3 na obr. 10.2 neexistuje přechod do jiného stavu a je tedy **konecovým** stavem. Automat z něj může být vyveden jen vypnutím napájení nebo působením signálů pro nucené nastavení stavu, což v grafu není nijak vyjádřeno.

10.3 Popis pomocí soustavy rovnic

Pro symbolický popis přechodu ze stavu v čase t do stavu ve stavu $t+1$ bude použit symbol „ \rightarrow “, pro současný výskyt stavu S a vstupu I bude použit symbol „ α “. **Přechodová funkce** je pak popsána rovnicemi ve tvaru:

$$S_i \alpha I_j \rightarrow S_k$$

Konkrétně pro Mealyho automat podle grafu z obr. 10.3:

$$\begin{aligned} S_0 \alpha I_0 &\rightarrow S_1 \\ S_0 \alpha (I_1 + I_2) &\rightarrow S_0 \\ S_1 \alpha I_0 &\rightarrow S_2 \\ S_1 \alpha I_1 &\rightarrow S_0 \\ S_1 \alpha I_2 &\rightarrow S_1 \\ S_2 \alpha I_0 &\rightarrow S_2 \\ S_2 \alpha (I_1 + I_2) &\rightarrow S_0 \end{aligned}$$

Pro Mooreho automat platí stejný formální zápis.

Výstupní funkce je funkcí kombinační a pro Mealyho automat je zapsána soustavou rovnic ve tvaru:

$$S_i \alpha I_i : O_k$$

Symbol „ $:$ “ zde značí okamžité vytvoření výstupu. Pro Mooreho automat má výstupní funkce tvar:

$$S_i : O_k$$

Konkrétně pro Mealyho automat z obr. 10.3 je výstupní funkce popsána rovnicemi:

$$S_0 \alpha I_0 : O_0$$

$$S_0 \alpha (I_1 + I_2) : O_1$$

$$S_1 \alpha I_0 : O_2$$

$$S_1 \alpha I_1 : O_2$$

.....

.....

atd.

Pro Mooreho automat z obr. 10.2 je výstupní funkce:

$$\begin{aligned}S_0 &: O_0 \\S_1 &: O_0 \\S_2 &: O_1 \\S_3 &: O_2\end{aligned}$$

10.4 Popis pomocí tabulek

Tabulka přechodů je konstruována tak, že v prvém sloupci jsou současné stavy, v dalších sloupcích pod příslušnými vstupy jsou následující stavy. Tabulka přechodů je konstruována u Mealyho automatu stejně jako u automatu Mooreho. Tabulka výstupů u Mooreho automatu má v prvém sloupci stavy a ve druhém výstupy. U Mealyho automatu má v prvém sloupci stavy a v dalších sloupcích pod příslušnými vstupy má výstupy.

Tab. 10.1 a tab. 10.2 patří Mooreho automatu z obr. 10.2, tab. 10.3 a tab. 10.4 patří Mealyho automatu z obr. 10.3.

Tab. 10.1 Tabulka přechodů Mooreho automatu

	I ₀	I ₁	I ₂
S ₀	S ₁	S ₀	S ₁
S ₁	S ₂	S ₀	S ₃
S ₂	S ₂	S ₀	S ₂
S ₃	S ₃	S ₃	S ₃

Tab. 10.2 Tabulka výstupů Mooreho automatu

S ₀	O ₀
S ₁	O ₀
S ₂	O ₁
S ₃	O ₂

Tab. 10.3 Tabulka přechodů Mealyho automatu

	I ₀	I ₁	I ₂
S ₀	S ₁	S ₀	S ₀
S ₁	S ₂	S ₀	S ₁
S ₂	S ₂	S ₀	S ₀

Tab. 10.4 Tabulka výstupů Mealyho automatu

	I ₀	I ₁	I ₂
S ₀	O ₀	O ₁	O ₁
S ₁	O ₂	O ₂	O ₀
S ₂	O ₂	O ₀	O ₀

Tabulky odpovídají dříve uvedeným grafům a soustavám rovnic.

10.5 Popis v některém programovacím jazyku

Tento způsob popisu je závislý na tom, v jakém jazyku má být úloha programována. Pro úlohy realizované počítači, mikrořadiče apod. je vhodným programovacím jazykem „C“. Pro realizaci programovatelnými logickými obvody CPLD nebo FPGA jsou vhodné jazyky **VHDL** nebo **VERILOG**, pro velmi jednoduché úlohy vyhoví i jazyky **PALASM** nebo **ABEL**.

Následující výpis kódu ukazuje realizaci výše popsaného Moorova automatu v jazyce VHDL. Vstupní signály $i0, i1, i2$ jsou typu **std_logic** a předpokládá se, že je aktivní vždy jen jeden z nich. Výstupní signály $o0, o1, o2$ jsou rovněž typu **std_logic** a je vždy také aktivní jen jeden z nich. Stavový registr je realizován pomocí výčtového typu a může nabývat hodnot $s0, s1, s2$ a $s3$. O použitém kódování stavového registru rozhoduje při syntéze zdrojového kódu buď automaticky použitý syntezátor (volí většinou pro daný automat nejvhodnější kódování) a nebo na základě vlastního zásahu uživatel (změnou přepínače syntezátoru, apod.) U většiny syntezátortů lze zvolit kódování 1 z N (označováno většinou jako „One Hot“), binární kódování a Grayovo kódování, případně další jako „Two Hot“ nebo náhodné kódování (někdy označované jako minimální počet bitů). Podrobnější informace ke kódování stavového registru jsou uvedeny v podkapitole 10.7. Celý stavový automat je tvořen třemi procesy. První proces provádí při každé náběžné hraně signálu *clk* zápis hodnoty následujícího stavu *dalsi_stav* do stavového registru *stav_reg*. V případě že je hodnota *dalsi_stav* odlišná od hodnoty v registru *stav_reg* dochází ke změnu stavu automatu. Druhý proces realizuje přechodovou funkci stavového automatu pomocí konstrukce s **case**. Pro správnou funkci automatu je nutné, aby v citlivostním seznamu tohoto procesu byl uveden stavový registr a všechny vstupní signály. Třetí proces pak realizuje výstupní funkci stavového automatu. Výstupní funkce je opět realizována pomocí konstrukce s **case**. Protože se jedná v tomto případě o stavový automat typu Moore, musí být v citlivostním seznamu tohoto procesu uveden pouze stavový registr, v našem případě tedy identifikátor *stav_reg*. Velmi doporučujeme při popisu přechodové i výstupní funkce pomocí **case** uvést vždy podmítku **when others**. V případě přechodové funkce přiřazující nějakou hodnotu do stavového registru (v našem případě zůstáváme v současném stavu), v případě výstupní funkce pak do výstupních signálů. Podmínka **when others** nám osetří případ, kdy opomeneme popsat některý stav, což by při syntéze vedlo ke generování klopných obvodů typu latch namísto kombinacní logiky. V případě, že máme popsány reakce pro všechny hodnoty, vypíše většina syntezátorů navíc informační zprávu o tom, že podmínka **when others** nebude nikdy zvolena.

```

library ieee;
use ieee.std_logic_1164.all;

entity moore is
  port (
    clk : in std_logic;
    i0, i1, i2 : in std_logic;
    o0, o1, o2 : out std_logic
  );
end moore;

architecture behav of moore is
  type stavy is (s0, s1, s2, s3);
  signal stav_reg, dalsi_stav : stavy := s0;
begin
  -- realizace paměti stavu automatu
  process (clk)
  begin
    if clk'event and clk = '1' then
      stav_reg <= dalsi_stav;
    end if;
  end process;

```

```

-- prechodova funkce automatu
process (stav_reg, i0, i1, i2)
begin
    -- implicitne zustava automat ve stavu v ktere se prave nachazi
    dalsi_stav <= stav_reg;

    -- prechodove podminky vedouci na jiny stav
    case stav_reg is
        -- stav s0
        when s0 =>
            if i0 = '1' or i2 = '1' then
                dalsi_stav <= s1;
            end if;
        -- stav s1
        when s1 =>
            if i0 = '1' then
                dalsi_stav <= s2;
            elsif i1 = '1' then
                dalsi_stav <= s0;
            elsif i2 = '1' then
                dalsi_stav <= s3;
            end if;
        -- stav s2
        when s2 =>
            if i1 = '1' then
                dalsi_stav <= s0;
            end if;
        -- stav s3
        when s3 =>
            dalsi_stav <= s3;
        -- pri vsech ostatnich stavech zustava v soucasnem stavu
        when others =>
            dalsi_stav <= stav_reg;
    end case;
end process;

-- vystupni funkce automatu
process (stav_reg)
begin
    case stav_reg is
        -- stav s0
        when s0 =>
            o0 <= '1';
            o1 <= '0';
            o2 <= '0';
        -- stav s1
        when s1 =>
            o0 <= '1';
            o1 <= '0';
            o2 <= '0';
        -- stav s2
        when s2 =>
            o0 <= '0';
            o1 <= '1';
            o2 <= '0';
    end case;
end process;

```

```

-- stav s3
when s3 =>
    o0 <= '0';
    o1 <= '0';
    o2 <= '1';
-- v ostatnich pripadech nic
when others => null;
end case;
end process;
end behav;

```

Výpis 10.1 Konečný automat typu Moore

Následující výpis kódu ukazuje realizaci výše popsaného Mealyho automatu v jazyce VHDL. Vstupní signály i_0, i_1, i_2 i výstupní signály o_0, o_1, o_2 jsou opět typu **std_logic** a předpokládá se, že je aktivní vždy jen jeden z nich. Stavový registr je realizován pomocí výčtového typu a může nabývat hodnot s_0, s_1 a s_2 . Celý stavový automat je opět tvořen třemi procesy. První proces provádí při každé náběžné hraně signálu clk zápis hodnoty následujícího stavu *dalsi_stav* do stavového registru *stav_reg*. Druhý proces realizuje přechodovou funkci stavového automatu pomocí konstrukce s **case**. Pro správnou funkci automatu je opět nutné, aby v citlivostním seznamu tohoto procesu byl uveden stavový registr a všechny vstupní signály. Třetí proces pak realizuje výstupní funkci stavového automatu. Výstupní funkce je opět realizována pomocí konstrukce s **case**. Protože se nyní jedná o stavový automat typu Mealy, musí být v citlivostním seznamu tohoto procesu uveden stavový registr a všechny vstupní signály neboť hodnota výstupní funkce musí být závislá na vnitřním stavu automatu a stavu na vstupu automatu.

```

library ieee;
use ieee.std_logic_1164.all;

entity mealy is
port (
    clk : in std_logic;
    i0, i1, i2 : in std_logic;
    o0, o1, o2 : out std_logic
);
end mealy;

architecture behav of mealy is
type stav is (s0, s1, s2);
signal stav_reg, dalsi_stav : stav := s0;
begin
    -- realizace paměti stavu Mealyho automatu
process (clk)
begin
    if clk'event and clk = '1' then
        stav_reg <= dalsi_stav;
    end if;
end process;

```

```

-- prechodova funkce Mealyho automatu
process (stav_reg, i0, i1, i2)
begin
    -- prechodove podminky vedouci na jiny stav
    case stav_reg is
        -- stav s0
        when s0 =>
            if i0 = '1' then
                dalsi_stav <= s1;
            else
                dalsi_stav <= s0;
            end if;
        -- stav s1
        when s1 =>
            if i0 = '1' then
                dalsi_stav <= s2;
            elsif i1 = '1' then
                dalsi_stav <= s0;
            else
                dalsi_stav <= s1;
            end if;
        -- stav s2
        when s2 =>
            if i1 = '1' or i2 = '1' then
                dalsi_stav <= s0;
            else
                dalsi_stav <= s2;
            end if;
        -- pri vsech ostatnich stavech zustava v soucasnem stavu
        when others =>
            dalsi_stav <= stav_reg;
    end case;
end process;

-- vystupni funkce Mealyho automatu
process (stav_reg, i0, i1, i2)
begin
    case stav_reg is
        -- stav s0
        when s0 =>
            o0 <= i0;
            o1 <= i1 or i2;
            o2 <= '0';
        -- stav s1
        when s1 =>
            o0 <= i2;
            o1 <= '0';
            o2 <= i0 or i1;
        -- stav s2
        when s2 =>
            o0 <= i1 or i2;
            o1 <= '0';
            o2 <= i0;
    end case;
end process;

```

```

-- v ostatnich pripadech nedelej nic
when others => null;
end case;
end process;
end behav;

```

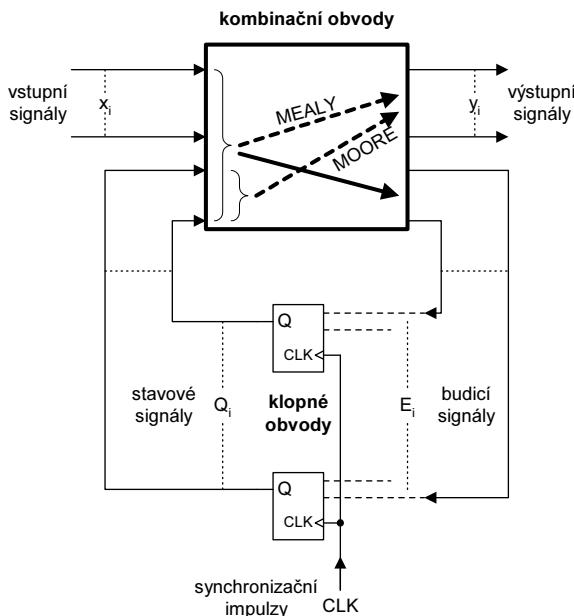
Výpis 10.2 Konečný automat typu Mealy

Stavové automaty lze samozřejmě realizovat i pouhými dvěma procesy, například jedním procesem realizujícím zápis do registru a druhým procesem realizujícím přechodovou funkci a výstupní najednou. Toto řešení se však vřele nedoporučuje, neboť zápis se tak stává, již i při poměrně malém automatu, špatně čitelný a často vede k mnoha chybám jak ve vlastním popisu automatu, tak k problémům při vlastní syntéze takto navrženého automatu. Každý stavový automat se tedy doporučuje realizovat vždy minimálně třemi procesy.

10.6 Obvodová realizace konečného automatu

Obr. 10.4 ukazuje synchronní sekvenční obvod. Vnitřní stav je dán kombinací hodnot signálů Q_i , získaných z klopných obvodů. Pokud známe typ klopných obvodů (D, T, JK, ...), vždy synchronní, je možné ke každému přechodu ze současného do následujícího stavu určit potřebné hodnoty budicích signálů klopných obvodů. Ty jsou označeny obecně jako E_i – v konkrétním případě lze dosadit D_i , T_i , J_i , K_i , atd. Jelikož následující stav je určen současným stavem a vstupem v čase t , bude platit:

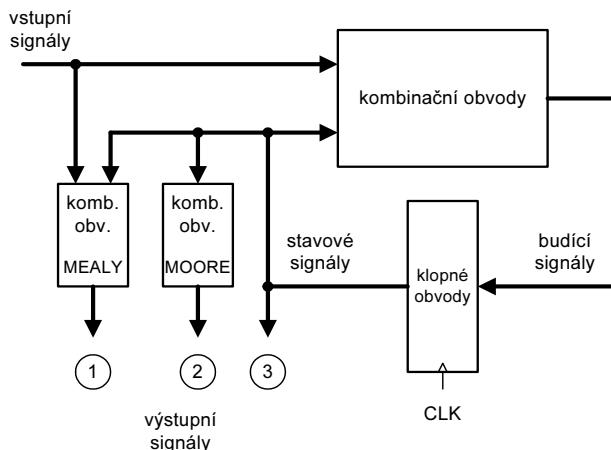
$$E_i = e_i(Q_0, Q_1, \dots, x_0, x_1, \dots)$$



Obr. 10.4 Cesty signálů v synchronním sekvenčním obvodu

Funkce e_i jsou **budicí funkce**. Všechny proměnné budicích funkcí působí současně, tedy pro vytváření signálů E_i mohou sloužit kombinační obvody. Cesta budicích signálů je v obrázku symbolicky vyznačena tlustou plnou šipkou.

Výstupy Mooreho sekvenčního obvodu jsou závislé jen na jeho stavech, tedy výstupní proměnné y_i budou kombinačními funkcemi jen proměnných Q_i , jak je v obrázku vyznačeno jednou tlustou čárkovanou šipkou. Výstupy sekvenčního obvodu Mealyho jsou závislé jak na jeho stavech, tak na vstupech, a tedy výstupní proměnné y_i budou kombinačními funkcemi proměnných Q_i i x_i , jak je v obrázku vyznačeno druhou tlustou čárkovanou šipkou. Celý komplex kombinačních obvodů lze podle jejich funkce **rozdělit na dvě dílčí složky**: jednu pro vytváření **budicích** signálů, druhou pro vytváření **výstupních** signálů. Rozdělením na



Obr. 10.5 Varianty vytváření výstupů

dvě části se zjednoduší řešení každé z nich. Snižuje se sice možnost skupinové minimalizace, to ale není podstatné.

Obr. 10.5 ukazuje takovéto rozdělení kombinačních obvodů. Výstupní obvody mohou mít tři různé podoby. Případ 1 (v kroužku) ukazuje obvody Mealyho automatu, případ 2 se týká automatu Mooreho a případ 3 je opět variantou Mooreho automatu, u které kombinační obvody mohou ve zvláštních případech zcela odpadnout a výstupní signály jsou vyvedeny přímo z výstupů klopných obvodů. Tak jsou např. řešeny synchronní čítače. V kombinačních obvodech na výstupu jsou prakticky vždy **hazardy** a na vstupech se pak mohou vyskytovat falešné impulzy. Pokud kombinační obvody odpadnou, neexistují ani tyto falešné impulzy. Řešení bez výstupních kombinačních obvodů však může vyžadovat **zvýšený počet klopných obvodů**.

10.7 Kódování stavů

Každý stav odpovídá jediné kombinaci hodnot signálů. Přiřazení stavů kombinacím hodnot signálů je nutné provést pro vstupní stavy, vnitřní stavy i výstupní stavy. Dostaneme tak **vstupní kód**, **vnitřní kód** a **výstupní kód**. Pro synchronní sekvenční obvody platí, že při

jakémkoliv kódování lze patřičně navrhnout obvody tak, že je dosaženo požadované funkce. Stanovení kódů však zpravidla má vliv na **složitost** obvodů. Minimalizace celého sekvenčního obvodu tedy vyžaduje vzít v úvahu i volbu kódů. Je to úloha velmi složitá, ale při existujících technologiích velké integrace ne vždy nutná.

Pro další návrh je nutné zjistit **počty signálů**, potřebné k zakódování stavů. Pro n stavů a k signálů platí:

$$2^{k-1} < n \leq 2^k$$

Zde k značí minimální počet signálů. Někdy je vhodné zavést větší než minimální počet signálů a pak budou existovat nevyužité kombinace hodnot signálů a jim odpovídající počet nevyužitých stavů. **Nevyužité stavy** ostatně existují i v případě, že n není mocninou dvou. Tak např. při počtu stavů 5 je minimální počet signálů 3, ale třemi signály lze kódovat $2^3 = 8$ stavů. Proto budou existovat 3 nevyužité stavy. S těmito se při návrhu nemusí počítat, mohou ale způsobit komplikace – viz další text.

Při **stanovení kódu** je třeba dodržet dvě podmínky: každému stavu smí být přiřazena jen jedna kombinace hodnot, a každé kombinaci hodnot smí být přiřazen jen jeden stav. Úloha je zdánlivě triviální, ale při velkém počtu stavů není omyl vyloučen. Jednoduchá metoda, vylučující omyly, spočívá v tom, že index u symbolu stavu se převede do podoby binárního čísla a hodnoty jednotlivých bitů se přiřadí jednotlivým signálům. Pořadí signálů se musí zvolit.

Tak např. pro kódování pěti vstupních stavů I_0, I_1, I_2, I_3, I_4 jsou zapotřebí 3 signály. Označme je x_0, x_1, x_2 a seřaďme je v pořadí (MSB) $x_2x_1x_0$. Pak index u I_0 , tj. dekadické číslo 0, odpovídá binárnímu číslu 000, tedy vstupní stav I_0 bude kódován kombinací hodnot $x_2 = 0, x_1 = 0, x_0 = 0$, zkráceně zapsáno $x_2x_1x_0$. Obdobně postupujeme i u ostatních stavů. Celkově lze tento kód shrnout v tab. 10.5.

Tab. 10.5 Příklad vstupního kódu

I_0	$\overline{x}_2 \overline{x}_1 \overline{x}_0$
I_1	$\overline{x}_2 \overline{x}_1 x_0$
I_2	$\overline{x}_2 x_1 \overline{x}_0$
I_3	$\overline{x}_2 x_1 x_0$
I_4	$x_2 \overline{x}_1 \overline{x}_0$

Zbývají stavy I_5, I_6, I_7 , které budou nevyužité, a proto se nemusí kódovat.

Je třeba zdůraznit, že ve skutečnosti vstupní, vnitřní ani výstupní signály nemají nutně význam **čísla**. Na vstupu se může jednat např. o jednotlivé signály z kontaktů a senzorů technologického zařízení, na výstupu např. o jednotlivé řídicí signály pro technologické zařízení. V některých případech je samozřejmě možné seskupení několika signálů ve významu čísla – tak tomu např. může být v aritmetických jednotkách, v řídicích automatech s analogovými vstupními signály převáděnými A/D převodníky do číselné podoby, apod.

Ačkoliv vnitřní kód není nikdy zadán, je vstupní i výstupní kód často již součástí zadání, např. takto: „...následuje-li po kombinaci vstupních signálů $A=1, B=0, C=0$ kombinace $A=0, B=0, C=1$, bude na výstupu $Y=0$ a $Z=1$, a dále při vstupní kombinaci...“ atd.

Po stanovení kódů je možné přepsat rovnice přechodů a výstupů, nebo případně tabulky přechodů a výstupů tak, že symboly vstupů, stavů a výstupů se nahradí kombinacemi hodnot signálů podle zvolených kódů. Pokračujme v řešení Mealyho automatu, popsaného grafem v obr. 10.3. Má tři vstupní stavы I_0, I_1, I_2 a tedy musí mít dva vstupní signály, které označíme jako x_1, x_0 . Dále má tři vnitřní stavы S_0, S_1, S_2 , a tedy budou dva stavové signály Q_1, Q_0 . Tabulky vstupního a vnitřního kódu jsou v tab. 10.6 a tab. 10.7.

Tab. 10.6 Tabulka vstupního kódu

Stav	Kombinace hodnot x
I_0	$\bar{x}_1 \bar{x}_0$
I_1	$\bar{x}_1 x_0$
I_2	$x_1 \bar{x}_0$

Tab. 10.7 Tabulka vnitřního kódu

Stav	Kombinace hodnot Q
S_0	$\bar{Q}_1 \bar{Q}_0$
S_1	$\bar{Q}_1 Q_0$
S_2	$Q_1 \bar{Q}_0$

Dále jsou uvedeny rovnice přechodů, které byly již odvozeny v předcházejícím textu. Za symboly stavů a vstupů jsou dosazeny kódy stavů a vstupů. Symbol „ α “, vyjadřující současnou existenci stavu a vstupu, je nahrazen logickým součinem. Dostaneme tak součiny stavových a vstupních signálů. Alternativní výskyt vstupů je vyjádřen logickým součtem (OR).

	současný stav a vstup	nový stav
$S_0 \alpha I_0 \rightarrow S_1$	$\bar{Q}_1 \bar{Q}_0 \cdot \bar{x}_1 \bar{x}_0$	$\rightarrow \bar{Q}_1 Q_0$
$S_0 \alpha (I_1 + I_2) \rightarrow S_0$	$\bar{Q}_1 \bar{Q}_0 \cdot (\bar{x}_1 x_0 + x_1 \bar{x}_0)$	$\rightarrow \bar{Q}_1 \bar{Q}_0$
$S_1 \alpha I_0 \rightarrow S_2$	$\bar{Q}_1 Q_0 \cdot \bar{x}_1 \bar{x}_0$	$\rightarrow Q_1 \bar{Q}_0$
$S_1 \alpha I_1 \rightarrow S_0$	$\bar{Q}_1 Q_0 \cdot x_1 \bar{x}_0$	$\rightarrow \bar{Q}_1 \bar{Q}_0$
$S_1 \alpha I_2 \rightarrow S_1$	$\bar{Q}_1 Q_0 \cdot x_1 \bar{x}_0$	$\rightarrow \bar{Q}_1 Q_0$
$S_2 \alpha I_0 \rightarrow S_2$	$Q_1 \bar{Q}_0 \cdot \bar{x}_1 \bar{x}_0$	$\rightarrow Q_1 \bar{Q}_0$
$S_2 \alpha (I_1 + I_2) \rightarrow S_0$	$Q_1 \bar{Q}_0 \cdot (\bar{x}_1 x_0 + x_1 \bar{x}_0)$	$\rightarrow \bar{Q}_1 \bar{Q}_0$

Tak např. druhá rovnice znamená, že ze stavu S_0 při současné existenci vstupu I_1 nebo I_2 se přejde do stavu S_0 . Stav S_0 je kódován jako $\bar{Q}_1 \bar{Q}_0$, vstup I_1 jako $\bar{x}_1 x_0$ a vstup I_2 jako $x_1 \bar{x}_0$. Následující stav S_0 je kódován jako $\bar{Q}_1 \bar{Q}_0$.

10.8 Volba klopných obvodů

Dále je třeba navrhnout typ klopných obvodů. Opět platí, že požadované funkce synchronního sekvenčního obvodu lze při patřičném návrhu dosáhnout s **jakýmkoliv** synchronními klopnými obvody. Dokonce lze pro každý vnitřní signál zvolit jiný typ KO, ovšem všechny se stejným způsobem reakce na hodinový impulz. Volba klopného obvodu však může ovlivnit složitost kombinačních obvodů. Všeobecně je možné se řídit tímto hrubým vodítkem:

- s KO typu D je odvození budicích funkcí nejjednoduší,
- s KO typu T vycházejí jednoduší kombinační obvody u těch sekvenčních obvodů, jejichž funkce se blíží funkci čítače,
- s KO typu JK vycházejí všeobecně nejjednoduší budicí funkce, jsou ale na každý KO dvě.

Po volbě KO již lze odvodit budicí funkce, které jsou funkciemi kombinačními. Každá z budicích funkcí závisí na všech vstupních signálech sekvenčního obvodu a na všech výstupních signálech klopných obvodů.

10.9 Návrh budicích funkcí

Pro každý přechod mezi dvěma vnitřními stavami lze při znalosti kódů určit, jak se má překládat daný klopný obvod (možnosti jsou: $0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 0$, $1 \rightarrow 1$). Pravidla pro jednotlivé typy KO, tj. **požadavky na budicí funkce**, lze snadno odvodit z chování obvodů. V tab. 10.8 jsou požadavky přehledně shrnutý.

Tab. 10.8 Stavy na vstupech KO pro požadované přechody

$Q^t \rightarrow Q^{t+1}$	D^t	T^t	J^t	K^t
$0 \rightarrow 0$	0	0	0	–
$0 \rightarrow 1$	1	1	1	–
$1 \rightarrow 0$	0	1	–	1
$1 \rightarrow 1$	1	0	–	0

Pro KO typu D lze jednoduše přepsat sloupec Q^{t+1} do budicí funkce D^t . Pro KO typu T má vstup T^t hodnotu 1 kdykoliv se Q má měnit. U KO typu JK jsou v budicích funkcích neurčené stavy, což umožní větší zjednodušení obvodů.

Při zvolených typech klopných obvodů lze z rovnic přechodů v kódované formě již přímo vyčíst budicí funkce KO. Budicí funkce budeme vyjadřovat v **součtové formě** (součet součinů), takže podstatné budou jen ty kombinace hodnot proměnných, ve kterých budicí funkce má mít **hodnotu 1**. Je to v podstatě stejná úloha, jako odvození výrazu v DNF z pravdivostní tabulky funkce.

Pokračujme v řešení sekvenčního obvodu z předchozího textu. Předpokládejme, že proměnná Q_1 je na výstupu klopného obvodu typu D. Z rovnic přechodů v kódovaném tvaru

vybereme ty, kde na místě Q_1 bude přechod z \bar{Q}_1 do Q_1 nebo z Q_1 do \bar{Q}_1 . Při tom musí být $D_1 = 1$. Je to 3. a 6. rovnice. Stejným postupem se odvodí i D_0 . Dostaneme tak:

$$D_1 = \bar{Q}_1 Q_0 \cdot \bar{x}_1 \bar{x}_0 + Q_1 \bar{Q}_0 \cdot \bar{x}_1 \bar{x}_0$$

$$D_0 = \bar{Q}_1 \bar{Q}_0 \cdot x_1 \bar{x}_0 + \bar{Q}_1 Q_0 \cdot x_1 \bar{x}_0$$

Kdybychom volili klopné obvody typu T, bylo by třeba z rovnic přechodů vybrat ty, ve kterých se stav na výstupu Q mění. Dostali bychom:

$$T_1 = \bar{Q}_1 Q_0 \cdot \bar{x}_1 \bar{x}_0 + Q_1 \bar{Q}_0 \cdot (\bar{x}_1 x_0 + x_1 \bar{x}_0)$$

$$T_0 = \bar{Q}_1 \bar{Q}_0 \cdot \bar{x}_1 \bar{x}_0 + \bar{Q}_1 Q_0 \cdot \bar{x}_1 x_0 + \bar{Q}_1 Q_0 \cdot \bar{x}_1 x_0 = \dots \dots = \bar{Q}_1 \bar{x}_1 \bar{x}_0 + \bar{Q}_1 Q_0 \bar{x}_1$$

Pro případ klopných obvodů typu JK je nutné nalézt dvě budicí funkce. Do funkce J **musí být** vybrány rovnice, ve kterých dochází ke změně z \bar{Q} na Q . Vzhledem k neurčeným stavům dále **mohou být** vybrány rovnice, ve kterých dochází k přechodu z Q do \bar{Q} nebo z \bar{Q} do Q . Do funkce K musí být vybrány rovnice, ve kterých dochází ke změně z Q na \bar{Q} . Vzhledem k neurčeným stavům dále mohou být vybrány rovnice, ve kterých dochází k přechodu z \bar{Q} do Q nebo z Q do \bar{Q} . Nepovinné členy jsou uzavřeny v hranatých závorkách.

$$J_1 = \bar{Q}_1 Q_0 \cdot \bar{x}_1 \bar{x}_0 + [\bar{Q}_1 \bar{Q}_0 \cdot \bar{x}_1 \bar{x}_0 + \bar{Q}_1 \bar{Q}_0 \cdot (\bar{x}_1 x_0 + x_1 \bar{x}_0)] = \dots \dots = \bar{Q}_1 Q_0 \cdot \bar{x}_1 \bar{x}_0$$

$$K_1 = \bar{Q}_1 \bar{Q}_0 \cdot (\bar{x}_1 x_0 + x_1 \bar{x}_0) + [\bar{Q}_1 \bar{Q}_0 \cdot \bar{x}_1 \bar{x}_0 + \bar{Q}_1 \bar{Q}_0 \cdot (\bar{x}_1 x_0 + x_1 \bar{x}_0) + \bar{Q}_1 Q_0 \cdot \bar{x}_1 \bar{x}_0 + \\ + \bar{Q}_1 Q_0 \cdot x_1 x_0 + \bar{Q}_1 Q_0 \cdot x_1 \bar{x}_0] = \dots \dots = \bar{Q}_1 x_1 \bar{x}_0 + \bar{Q}_1 \bar{x}_1 x_0$$

$$J_0 = \bar{Q}_1 \bar{Q}_0 \cdot \bar{x}_1 \bar{x}_0 + [\bar{Q}_1 Q_0 \cdot \bar{x}_1 \bar{x}_0 + \bar{Q}_1 Q_0 \cdot \bar{x}_1 x_0 + \bar{Q}_1 Q_0 \cdot x_1 \bar{x}_0] = \dots \dots = \bar{Q}_1 \bar{x}_1 x_0$$

$$K_0 = \bar{Q}_1 Q_0 \cdot \bar{x}_1 \bar{x}_0 + \bar{Q}_1 Q_0 \cdot \bar{x}_1 x_0 + [\bar{Q}_1 \bar{Q}_0 \cdot \bar{x}_1 \bar{x}_0 + \bar{Q}_1 \bar{Q}_0 \cdot (\bar{x}_1 x_0 + x_1 \bar{x}_0) + \\ + \bar{Q}_1 \bar{Q}_0 \cdot \bar{x}_1 x_0 + \bar{Q}_1 \bar{Q}_0 \cdot (x_1 x_0 + x_1 \bar{x}_0)] = \dots \dots = \bar{Q}_1 \bar{x}_1$$

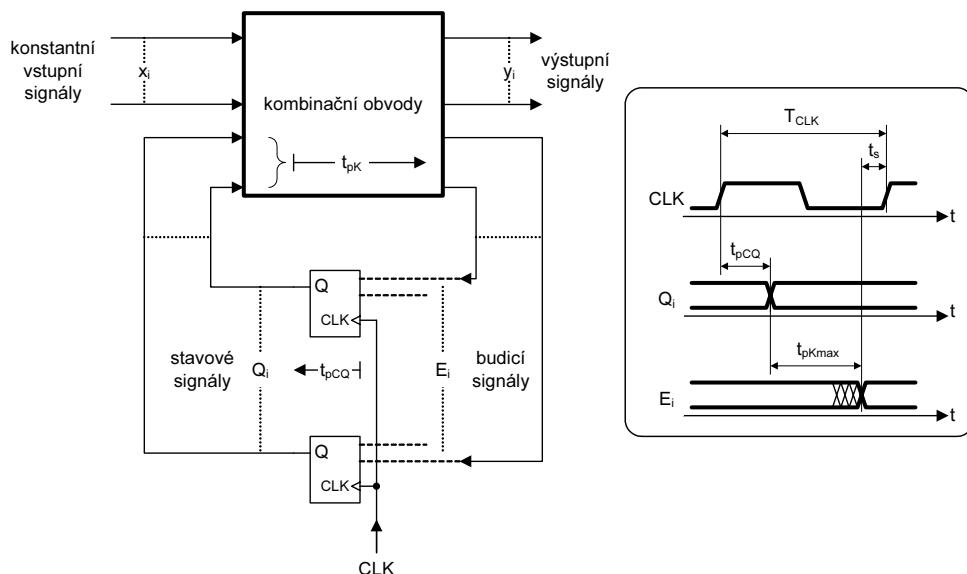
Kromě J_1 zřejmě neurčené stavy přispěly ke zjednodušení výrazů (funkce mají jen 4 proměnné, Karnaughovy mapy jsou použitelné). Nyní by bylo možné vybrat nejvhodnější budicí funkce a tím i sestavu klopných obvodů.

10.10 Časování signálů v synchronním sekvenčním obvodu

První úlohou bude zjistit **maximální kmitočet hodinových impulzů**. Pro jeho odvození budeme předpokládat, že vstupní signály se nemění. Obr. 10.6 vlevo ukazuje situaci, obrázek vpravo ukazuje časové diagramy. Po hraně CLK se překlápejí KO a se zpožděním t_{pCQ} se mění stavové signály Q . Ty dále procházejí kombinačními obvody a po odeslání přechodných dějů (zpoždění a hazardy) jsou po době t_{pKmax} vytvořeny budicí signály E_i . Tato doba vystihuje nejnepříznivější situaci s nejdelením zpožděním. Po nutné době předstihu klopných obvodů t_s lze přivést další hodinový impulz. Perioda a kmitočet impulzů CLK je tedy:

$$T_{CLK} \geq t_{pCQ} + t_{pKmax} + t_s, \quad f_{CLK} = 1/T_{CLK}$$

Při dané technologii obvodů lze periodu CLK zkrátit jen zkrácením cesty signálu přes kombinační obvody. K tomu přispěje dostupnost signálů Q na výstupech KO v **přímém i negovaném** tvaru, takže kombinační obvody mohou být realizovány jen jako **dvojstupňové** (tj. součet součinů).



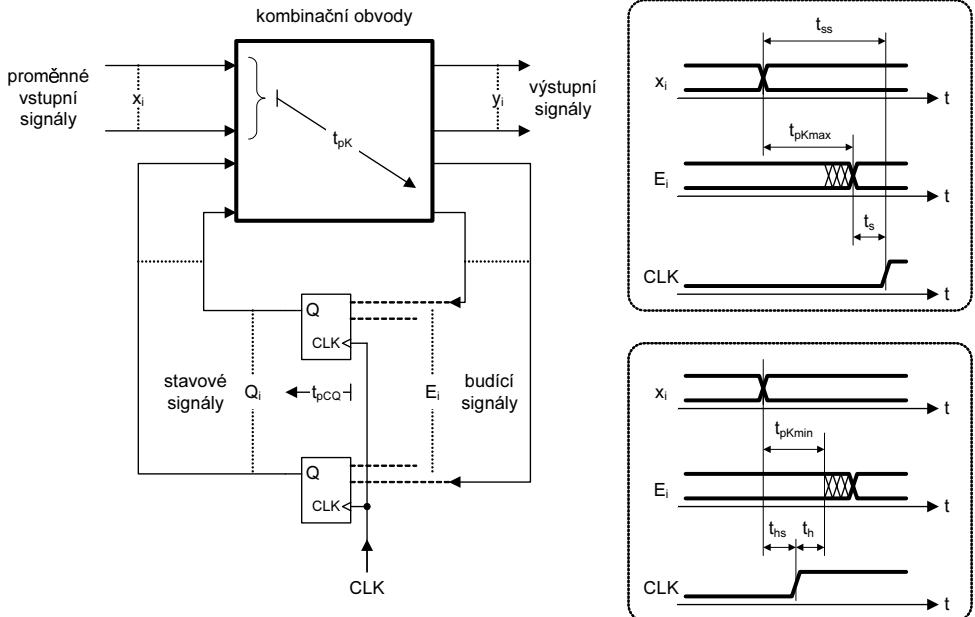
Obr. 10.6 Odvození maximálního kmitočtu hodinových impulzů

Druhou úlohou bude zjistit podmínky, za jakých se smí měnit **vstupní signály**. Obr. 10.7 vlevo ukazuje situaci, obrázek vpravo ukazuje časové diagramy. Po změně vstupu se zpožděním t_{PKmax} se ustálí budicí signály E . Po době předstihu klopných obvodů t_s lze již přivést hodinový impulz. Dostáváme tak **dobu předstihu sekvenčního obvodu** t_{ss} jako:

$$t_{ss} \geq t_{PKmax} + t_s$$

Budicí signály E_i musí zůstat po hraně CLK konstantní ještě po dobu přesahu klopných obvodů t_h . Minimální doba, po kterou se po změně vstupu ještě nezmění E , je t_{PKmin} . Z časového diagramu vpravo dole vyplývá **doba přesahu sekvenčního obvodu** t_{hs} jako:

$$t_{hs} \geq t_h - t_{PKmin}$$



Obr. 10.7 Odvození doby předstihu a přesahu vstupů vzhledem k hodinovým impulzům

Ve vztazích tedy vystupuje **maximální i minimální** zpoždění kombinačních obvodů. Minimální zpoždění přitom může být i nulové, neboť výstup některého klopného obvodu může být rovnou propojen se vstupem jiného KO (to není nijak výjimečný případ).

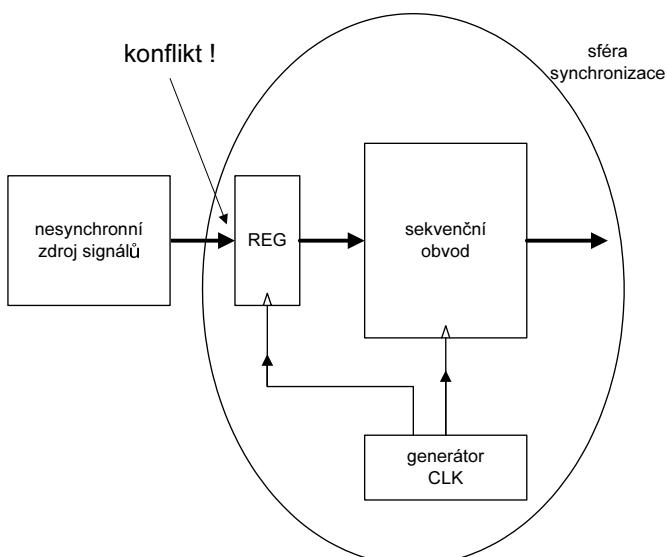
Při nesplnění výše uvedených podmínek pracuje sekvenční obvod nespolehlivě, tj. přechody mezi stavů neodpovídají vždy požadované přechodové funkci. Nejhorší je taková nespolehlivost, která se projevuje jen občas a **nahodile**. Tato selhání nemusí být odhalena při ověřování a testování systému a mohou se projevit až v provozu.

Jestliže připustíme změnu vstupů v kterémkoliv okamžiku během periody CLK , pak se může stát, že ke změně vstupů dojde právě v „zakázaném okénku“ o délce $t_{ss} + t_{hs}$. Pravděpodobnost, že k této situaci v jedné periodě CLK dojde, závisí na poměru délky „zakázaného okénka“ k periodě CLK , tedy $(t_{ss} + t_{hs})/T_{CLK}$. Vyšší kmitočet hodinových impulzů je tedy nepříznivý, naopak velmi rychlé klopné obvody situaci ulehčí. Pokud dojde ke změně vstupů během doby $t_{ss} + t_{hs}$, může u klopných obvodů dojít k metastabilnímu stavu. Ten má za následek nespolehlivou funkci klopných obvodů. Klopný obvod se může rozkmitat, nebo se může překlopit do nesprávného stavu, nebo se může překlopit do správného stavu s velkým zpožděním. Doba trvání metastabilního stavu závisí na technologii a na blízkosti okamžiku změny vstupů a aktivní hrany hodinového impulzu. Obecně platí, že čím blíže, tím hůře (bližší údaje jsou např. v [Met]). **Výskyt metastabilních stavů tak vždy snižuje spolehlivosť sekvenčního obvodu.**

Kromě výskytu metastabilních stavů může při nevhodném časování dojít k ještě horšímu jevu. Je to v případě, že počet vstupních signálů je větší než jeden. Při změnách vstupních stavů se může měnit několik vstupních signálů současně. Jejich „současnost“ však často nelze garantovat, neboť každý ze signálů může procházet cestami s různým zpožděním.

Na přechodnou dobu se tak mohou vyskytovat kombinace hodnot (mezistavy), které **nepatří ani k předchozímu, ani k následujícímu ustálenému stavu**. Pokud se tento falešný vstupní stav vyskytne během aktivní hrany hodinového impulzu, bude sekvenčním obvodem zpracován a může vyvolat naprosto chybný přechod mezi jeho vnitřními stavami.

Jisté zlepšení může přinést zařazení registru na výstupy předcházejících obvodů (u integrovaných registrů mají doby t_{PQ} jen velmi malý rozptyl) a vyrovnaní délky cest jednotlivých signálů – tím se omezí doba trvání mezistavů. Naopak žádný význam pro odstranění tohoto problému nemá zařazení registru do cesty **vstupních signálů** tak, že registr i následující sekvenční obvod jsou synchronizovány **stejným** zdrojem impulzů, jako je tomu v obr. 10.8. Tím se sice může řešit vhodné časování na vstupech samotného sekvenčního obvodu, ale celý problém se jen přesunul na vstupy registru. Ani tam totiž není možné připustit změny signálů v kterémkoliv okamžiku.



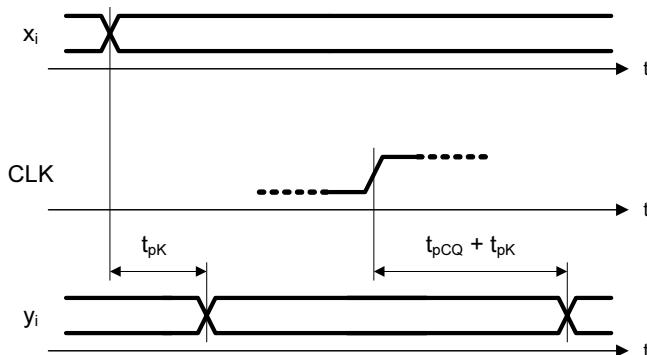
Obr. 10.8 Sekvenční obvod s předřazeným registrarem a společnou synchronizací

Spolehlivost sekvenčního systému tedy není jen otázkou spolehlivosti součástek, ale i vhodnosti či nevhodnosti jeho konstrukce. Nelze spoléhat na náhody a na statistiku. Porucha, i když třeba velmi nepravděpodobná, může nastat právě v nejnehodnějším okamžiku a třeba zrovna v příští sekundě. Konstrukce obvodu, vzájemné vazby a časování signálů musí zabránit jakýmkoliv nahodilostem. V případě sekvenčních obvodů je tedy nutné zabránit změnám vstupů v nežádoucích okamžicích.

Nejjednodušší metodou je zavést synchronizační impulzy *CLK* i do **předcházejících obvodů**, které jsou zdrojem vstupních signálů. Pak lze jedním jediným zdrojem synchronizačních impulzů určovat okamžiky změn signálů tak, aby nemohlo dojít ke kolizím. Dostáváme tak celou **sféru synchronizace** (angl. *clock domain*). Ta samozřejmě nemůže mít neomezený rozsah a tak pro spojení se zdroji signálů, které principiálně není možné vzájemně synchronizovat, musí být použit jiný princip, nazvaný „dotaz–odpověď“ (též „korespon-

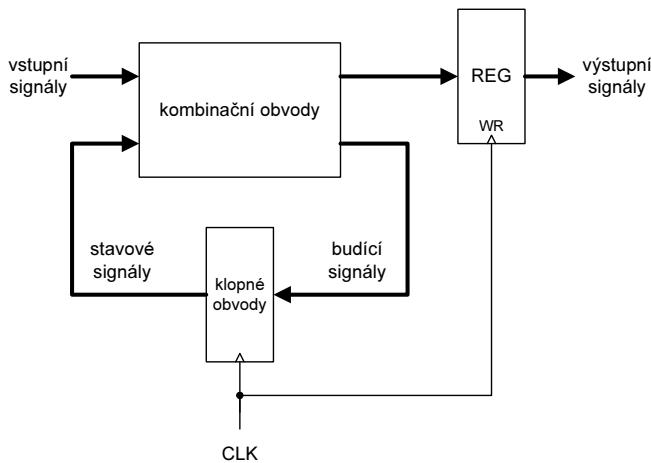
denční provoz“, angl. „*handshake*“). Další rozbor problematiky přechodu signálů přes sféry synchronizace je v kapitole 15.

Poslední důležité parametry souvisí s výstupními signály. V případě Mooreho i Mealyho automatu jsou výstupy funkcí stavu a jsou zpožděny za hranou *CLK* o zpoždění klopňových obvodů t_{pCQ} a výstupních kombinačních obvodů t_{pK} , celkem tedy o $t_{pCQ} + t_{pK}$. U Mealyho automatu jsou výstupy navíc i funkci vstupů, výstupní signály se tedy mohou měnit i mezi hodinovými impulzy se zpožděním t_{pK} . Časování u Mealyho automatu ukazuje obr. 10.9.



Obr. 10.9 Zpoždění mezi výstupem a vstupem sekvenčního obvodu

V kombinačních obvodech zpravidla existují hazardní stavy, takže ve výstupních signálech se mohou objevit falešné impulzy. K jejich odstranění lze použít některou z metod probraných v kapitole o hazardech. Nejčastěji se bude jednat o **výstupní registr**, zařazený do cesty výstupních signálů – viz obr. 10.10. Za předpokladu, že klopňové obvody výstupního registru mají stejný způsob řízení jako klopňové obvody v sekvenčním obvodu (hranové řízení), lze do nich zapisovat stejnými impulzy *CLK*, jaké jsou použity v sekvenčním obvodu. Mealyho automat bude ale postrádat jednu podstatnou vlastnost, a to reakci na změny vstupů mezi hodinovými impulzy – signály za výstupním registrum jsou teď mezi impulzy *CLK* konstantní. Jak u Mealyho, tak o Mooreho automatu budou signály za výstupním registrum **zpožděny o jeden takt *CLK*** za signály před výstupním registrum. To je třeba vzít v úvahu při návrhu automatu.

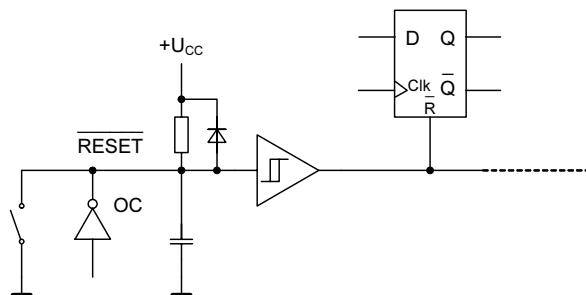


Obr. 10.10 Zařazení výstupního registru pro odstranění falešných impulzů

10.11 Nastavení počátečního stavu

Při zapnutí napájecího zdroje se jednotlivé klopné obvody dostanou do **náhodných stavů**. To je ve většině případů nežádoucí. U některých sekvenčních obvodů existují nevyužité vnitřní stavы (když počet využitých stavů není roven mocnině dvou). Ty mohou tvořit uzavřenou skupinu, která není žádým přechodem spojena s využitými stavami. Po náběhu napájecího napětí se sekvenční obvod může dostat do některého z takovýchto stavů a tím je znemožněna jeho správná činnost. Tento problém se řeší nastavením KO do vhodného počátečního stavu ihned po připojení napájecích napětí. Velmi často postačí nulování všech KO.

Možnost nastavení počátečního stavu je cenná i pro oživování a **diagnostiku** sekvenčního obvodu. Chceme-li prověřit všechny přechody z vnitřního stavu S_i tak, že nejprve vyzkoušíme přechod při vstupním stavu I_0 , dostane se obvod do nového vnitřního stavu. Pro prověření přechodů ze stavu S_i při dalších vstupních stavech je nutné převést obvod zpět do stavu S_i . Pak teprve lze pokračovat v kontrole podáním vstupního stavu I_1 , atd. Pokud nemáme možnost začít vždy znova z definovaného **počátečního stavu**, musíme převést obvod zpět do stavu S_i vhodnou vstupní sekvencí. Totéž ovšem musíme opakovat při všech vnitřních



Obr. 10.11 Automatické nulování po náběhu napájecího napětí

stavech. Nalezení vhodných vstupních sekvencí, které by obvod převedly z libovolného stavu do stavu požadovaného, je neobyčejně obtížné a často tyto sekvence ani neexistují.

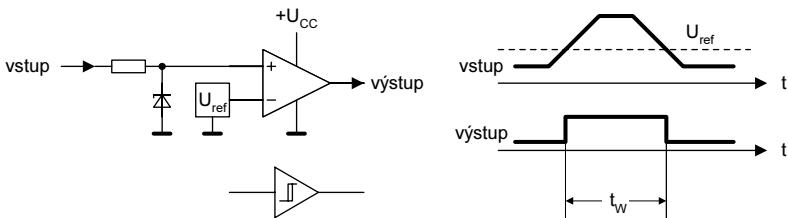
Impulz pro nastavení počátečního stavu lze při náběhu napájecího napětí generovat **automaticky**. Jednoduché řešení ukazuje obr. 10.11.

Po náběhu napájení se kondenzátor pomalu nabíjí a tak za hysterezním členem trvá po nějakou dobu stav L. Klopné obvody jsou tím nulovány. Po nabití kondenzátoru na horní prahovou úroveň se hysterezní člen překlopí do stavu H a nulování končí. Dioda umožnuje rychlé vybití kondenzátoru i při **krátkodobých výpadečích** napájení, takže obvod může ihned generovat nový nulovací impulz. Možnosti generace nulovacího podnětu jsou rozšířeny o tlačítko pro „ruční“ nulování a o obvod s otevřeným kolektorem, přes který mohou nulování vyvolat jiné obvody. Obvody pro automatické nastavení počátečního stavu jsou velmi často přímo součástí obvodů velké integrace.



TVAROVACÍ A ČASOVACÍ OBVODY

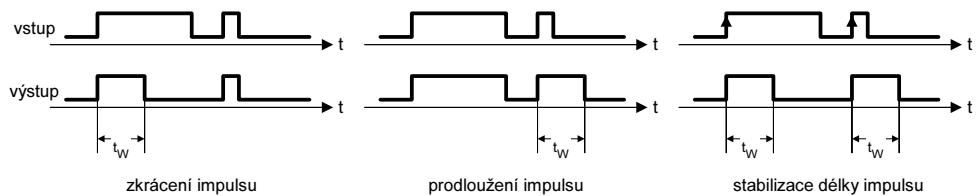
Obecná úloha tvarování impulzů je v číslicových systémech zjednodušena. Všeobecným požadavkem je docílení správného tvaru impulzu, který musí mít vyhovující úrovně U_L a U_H a co nejstrmější hrany. Mnohé vstupní signály tato kritéria nesplňují a napěťové úrovně je pak třeba upravit – příklad ukazuje obr. 11.1.



Obr. 11.1 Úprava nevhodného tvaru impulzů

Vstupní signál je analogovým komparátorem porovnáván s referenčním napětím. Omezovač se Zenerovou diodou je nutný jen v případě signálů, které by mohly vyvolat průraz vstupních obvodů komparátoru. Komparátor již sám zajišťuje správné úrovně výstupního signálu a vysokou strmost hran. Vhodným nastavením U_{ref} lze volit vstupní napětí, při kterém se komparátor překlápe. Je třeba poznamenat, že v této aplikaci nevyhoví běžný operační zesilovač, u kterého nejsou garantovány úrovně U_L a U_H na výstupu, a jehož rychlosť přeběhu je nedostatečná. Průchod úrovni U_{ref} vymezuje délku impulzu t_W . V jednoduchých případech lze komparátor nahradit členem s hysterézní charakteristikou, ovšem zde je úroveň U_{ref} pevně dána a není možné ji volit.

Uvnitř číslicového systému, kde jsou již dodrženy správné úrovně i strmosti hran, spočívá tvarování impulzu výhradně ve změně jeho délky. Tři případy ukazuje obr. 11.2. Obvod pro zkrácení impulzu zkracuje dlouhé impulzy na délku t_W , ale impulzy kratší propouští beze změny. Obvod pro prodloužení impulzu naopak dlouhé impulzy propouští beze změny a krátké impulzy prodlužuje na délku t_W . Obvod pro stabilizaci délky impulzu generuje impulz o délce t_W bez ohledu na délku vstupního impulzu. Nezávislost na délce vstupního impulzu znamená, že obvod je **spouštěn hranou**.

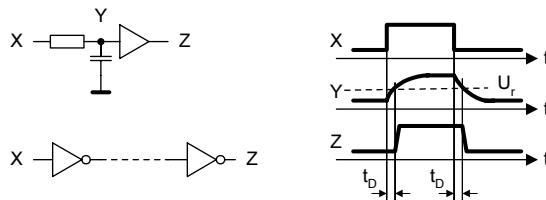


Obr. 11.2 Varianty úpravy délky impulzů

11.1 Asynchronní časovací obvody

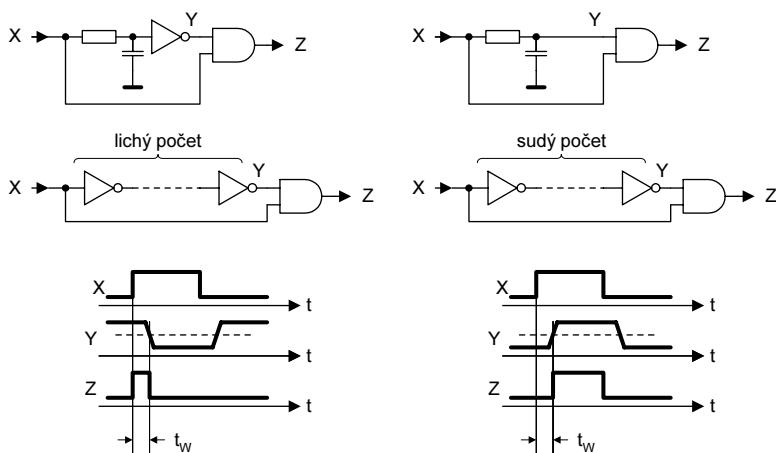
Ke zpoždění, zkrácení či posunutí impulzů lze využít obvody s RC články. Časování je závislé na přechodných dějích v těchto časovacích článcích a nevyžaduje žádné další synchronizační impulzy. Takovéto časovací obvody jsou **asynchronní**. Obr. 11.3 ukazuje obvod

pro časové posunutí impulzu. Za integračním článkem, v bodě Y, má impulz exponenciální náběh i době. Průchod referenční úrovně následujícího členu je tak zpožděn a na jeho výstupu Z jsou obě hrany impulzu zpožděny přibližně o stejnou dobu t_D . Potřebné zpoždění lze dosáhnout i spojením několika členů (invertorů nebo opakovačů) do kaskády. Výhodou jsou strmé hrany impulzů, zpoždění ale lze nastavit jen v hrubých skocích po jednotlivých t_{pd} . Pro některé aplikace to však vyhovuje.



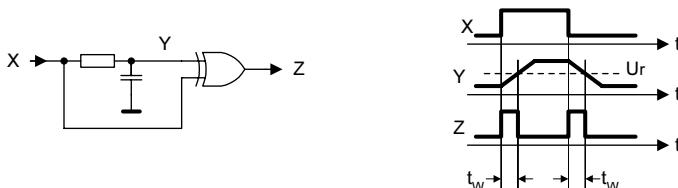
Obr. 11.3 Jednoduché asynchronní obvody pro zpoždění impulzů

Kombinací zpoždění a logické funkce lze realizovat řadu obvodů, které zkracují impulzy. Na obr. 11.4 vlevo se jedná o obvod, který zkracuje impulz tak, že odřezává jeho konec – na obrázku vpravo je naopak zpožděn začátek impulzu. Časové diagramy vše vysvětlují.



Obr. 11.4 Jednoduché asynchronní obvody pro zkrácení impulzů

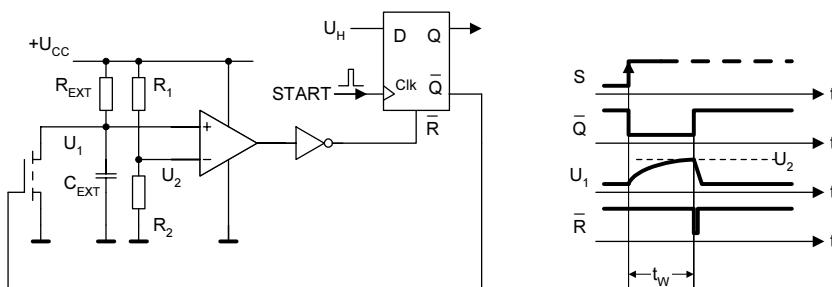
S využitím členu EX-OR lze sestavit obvod, který při každé změně stavu na vstupu generuje krátký impulz a realizuje tak funkci **detektoru změn** – viz obr. 11.5.



Obr. 11.5 Asynchronní detektor změn

Jednoduché obvody se články RC mají nedostatky. Signál na jejich výstupu má zešikměné hrany a při velkých časových konstantách může procházet přes referenční úroveň následujícího členu tak pomalu, že vyvolá jeho oscilace či jiné problémy. Vyhovující funkce je jen při malých časových konstantách, pro zpoždění řádově desítek ns. Dalším nedostatkem je malá přesnost časování a jeho časová nestálost. Pro určení zpoždění je kromě hodnot R a C nutné přesně znát napětí U_{OL} , U_{OH} , a U_r . Ani jedna z těchto hodnot není u číslicových členů garantována – zvláště napětí U_{OL} a U_{OH} jsou definována jako pásmá, ne jako přesné hodnoty. Všechna tato napětí kolísají s teplotou a napájecím napětím. V případě bipolárních obvodů jsou dalším zdrojem nepřesnosti vstupní proudy I_{IL} .

Pro prodloužení a stabilizaci délky impulzů je nutné zavést kladnou zpětnou vazbu. Princip ukazuje obr. 11.6.



Obr. 11.6 Využití přesného časovacího členu

Základem je klopný obvod a přesný časovací člen. V ustáleném stavu je $\bar{Q} = 1$, tranzistor je vybuzen a časovací kondenzátor vybit. Napětí na neinvertujícím vstupu komparátoru $U_1 = 0$. Napětí U_2 na invertujícím vstupu je vytvořeno odporovým děličem z napájecího napětí. Protože $U_2 > U_1$, je na výstupu komparátoru stav 0 a KO není nulován. Spouštěcím impulzem *START* se klopný obvod překlopí do stavu 1, tranzistor není vybuzen a U_1 exponenciálně nabíhá. Při shodě U_1 a U_2 se komparátor překlopí, tím se vynuluje KO, sepne se tranzistor a zkratuje kondenzátor. Obvod se tak opět dostane do ustáleného stavu. Na časovém diagramu jsou ukázány průběhy, charakterizující činnost obvodu. Doba trvání dočasněho stavu, kdy $Q = 1$ a $\bar{Q} = 0$, je označena jako t_W . Celý obvod má tedy jeden stabilní stav, kdy $Q = 0$, a druhý dočasný, kdy $Q = 1$. Odtud vyplývá název **monostabilní klopný obvod** (angl. „monostable“), zkratkou **MKO**. Přesnost určení délky impulzu je závislá na přesnosti použitého časovacího členu. Pro exponenciální náběh U_1 platí:

$$U_1 = U_{CC} \cdot (1 - e^{-\frac{t}{\tau}}), \quad \text{kde časová konstanta } \tau = R_{EXT} \cdot C_{EXT}.$$

Napětí U_2 je konstantní,

$$U_2 = U_{CC} \cdot \frac{R_2}{R_l + R_2}$$

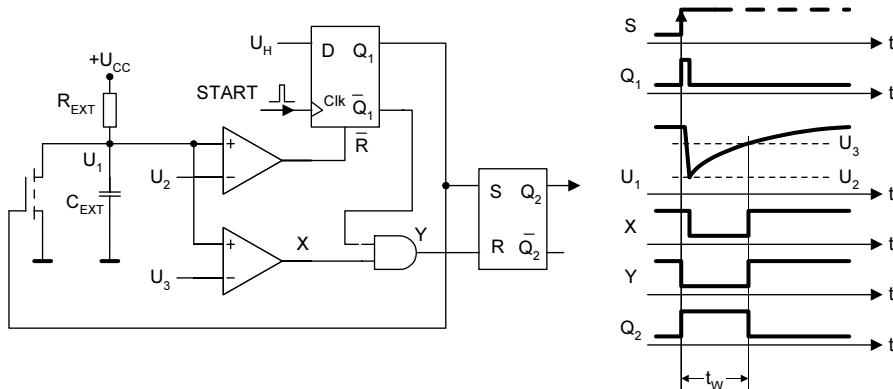
Při $t = t_w$ se obě napětí U_1 a U_2 rovnají, a tedy:

$$1 - e^{-\frac{t_w}{\tau}} = \frac{R_2}{R_l + R_2}, \quad \text{odtud } t_w = \tau \cdot \ln \frac{R_l + R_2}{R_2}$$

Důležitou vlastností tohoto obvodu je nezávislost t_W na U_{CC} i na napětích U_H a U_L .

Uvedený MKO je tedy **spouštěn hranou** vstupního impulzu a generuje jeden výstupní impulz přesně stanovené délky. Pokud se během dočasného stavu vyskytne další vstupní impulz, je **ignorován**.

Časovací člen a jeho řízení může mít i jinou podobu – viz další obr. 11.7.

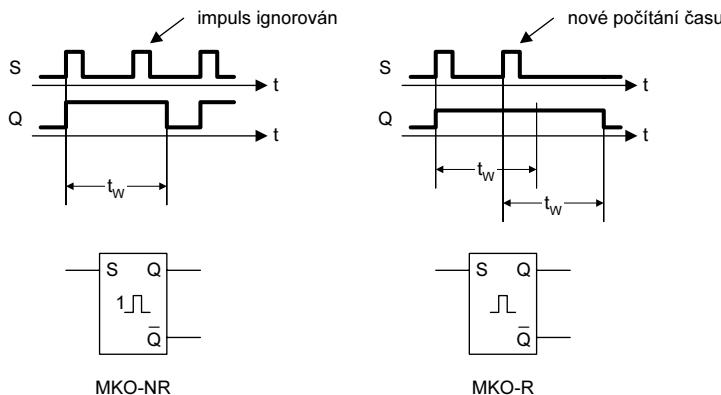


Obr. 11.7 Časovací člen s řízeným vybitím kondenzátoru

Zde jsou u časovacím členu dva komparátory a na jejich invertující vstupy jsou přivedena dvě napětí, $U_2 < U_3$, odvozená od U_{CC} dvěma odporovými děliči ($U_2 < U_3 < U_{CC}$), podobně jako v předchozím případě (děliče nejsou pro jednoduchost zakresleny). Ve stabilním stavu je $Q_1 = 0$ a $Q_2 = 0$, ale nyní je tranzistor zavřen, takže $U_1 = U_{CC}$. Na výstupech obou komparátorů je přitom stav 1, takže první klopný obvod není nulován, ale druhý ano. Spouštěcím impulzem se první klopný obvod překlopí ($Q_1 = 1$), a tím se překlopí i druhý KO ($Q_2 = 1$). Tranzistor je vybuzen a napětí U_1 rychle klesá. Jakmile je $U_1 = U_2$, překlopí se horní komparátor, tím se vynuluje Q_1 a vypne se tranzistor. Od tohoto okamžiku probíhá exponenciální náběh U_1 , tak jak byl popsán u předchozího obvodu. Při dosažení U_3 se překlopí dolní komparátor a tím se vynuluje Q_2 . O nezávislosti doby t_W na napájecím napětí platí stejná úvaha, jako u předcházejícího obvodu.

Opakovaný výskyt spouštěcího impulzu během doby t_W způsobí u tohoto obvodu **nové překlopení Q_1** do stavu 1 a tím opakované vybití kondenzátoru na hodnotu U_2 – to je totéž, jako nové počítání doby t_W . Pokud spouštěcí impulzy po sobě následují v intervalech kratších než t_W , pak se tento obvod nikdy nevrátí do stabilního stavu.

Oba typy chování shrnuje obr. 11.8. Vlevo je naznačeno chování monostabilního obvodu prvého typu, který po dobu t_W ignoruje opakované spouštěcí impulzy. Obvod není „znovu spustitelný“ (angl. Non-Retriggerable) a bude značen jako **MKO-NR**. Vpravo je naznačeno chování monostabilního obvodu druhého typu, který po každém spouštěcím impulzu začne znova odpočítávat dobu t_W . Obvod je „znovu spustitelný“ (angl. Retriggerable) a bude značen jako **MKO-R**.

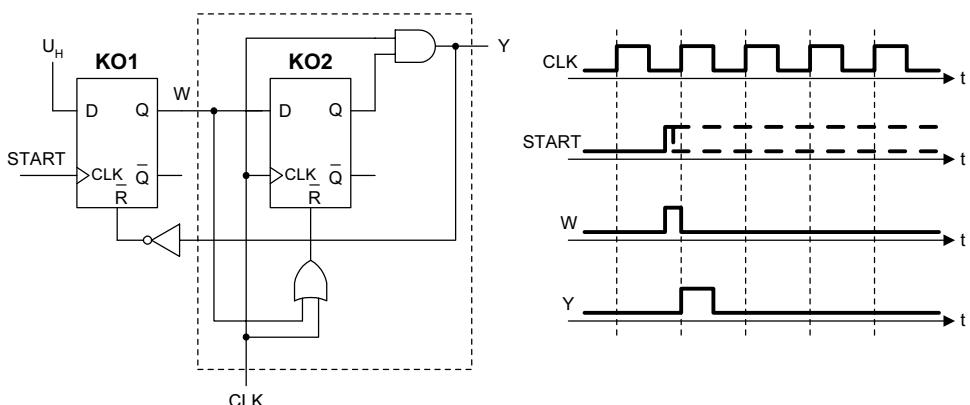


Obr. 11.8 Dva typy reakce MKO na spouštěcí impulz

U obou těchto typů MKO je doba překlopení dána velmi přesně. Přispívá k tomu jednak konstrukce časovacího členu, který je principiálně nezávislý na kolísání napájecího napětí, a jednak použití přesných komparátorů s malou teplotní závislostí. Obvody se vyskytují výhodně v **integrované** verzi. Časovací rezistor R_{EXT} a kondenzátor C_{EXT} mohou být připojeny zvenku, ostatní součástky včetně odporových děličů jsou na jednom čipu. Běžně se dosahuje kolísání t_W o zlomky procenta při kolísání U_{CC} o $\pm 5\%$ a kolísání teploty mezi 0 a 75°C .

11.2 Synchronní časovací obvody

U synchronních časovacích obvodů je časování docíleno odpočítáváním synchronizačních impulzů pomocí čítačů nebo posuvných registrů. Impulzy generované synchronními obvody mají délku či zpoždění vyjádřenou v počtu taktů synchronizačních impulzů a jejich přesnost závisí jen na přesnosti kmitočtu generátoru synchronizačních impulzů. Je-li řízen krystalem, je velmi přesný. Tolerance rezonančního kmitočtu běžného krystalového rezonátoru je lepší než 10^{-4} – do ní se počítá nepřesnost nastavení i teplotní závislost kmitočtu

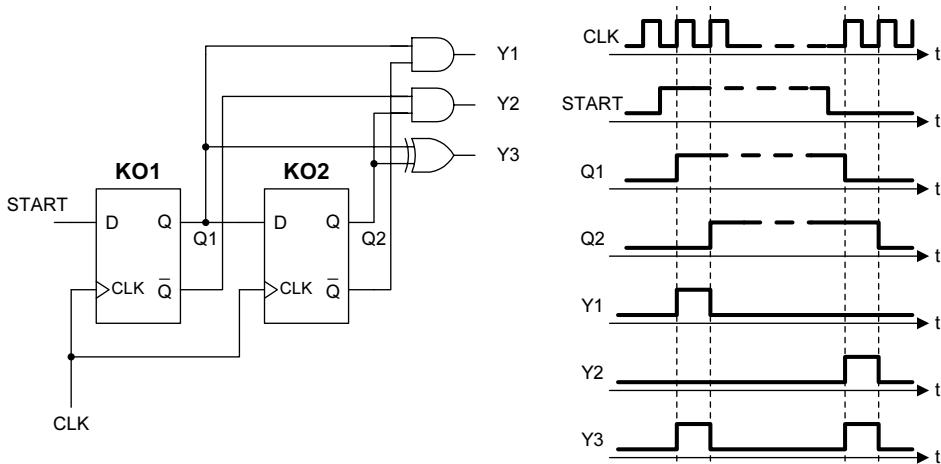


Obr. 11.9 Obvod pro generaci jednoho impulzu

v běžném rozsahu teplot (-20 až $+70$ °C). Existují i precizní rezonátory s mnohem menší tolerancí. Další výhodou časovacích obvodů v synchronní verzi je to, že výstupní impulzy, které dodávají dalším částem číslicového systému, mají přesně definované časování vůči synchronizačním impulzům, rozváděným po celém systému z jednoho centrálního generátoru. Odpadají pak problémy plynoucí z nedodržení podmínek správného časování u klopných obvodů nebo registrů.

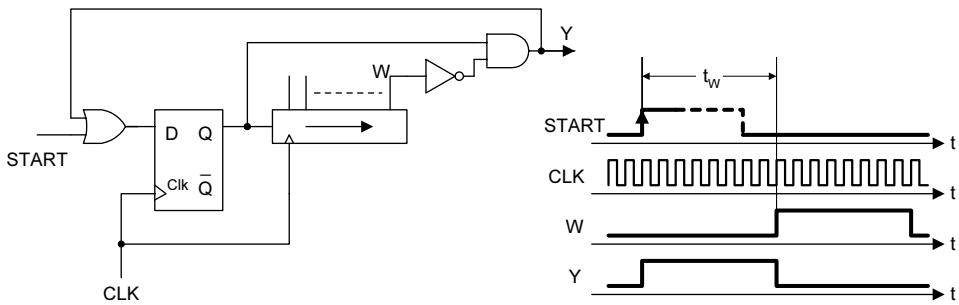
Obr. 11.9 ukazuje obvod pro **generaci jednoho impulzu**, vždy o délce rovné **délce synchronizačního impulzu**. Základem je synchronní spínač, známý z kapitoly o klopných obvozech. Náběžnou hranou na vstupu *START* se KO_1 překlopí do stavu 1 a následně synchronní spínač propustí jeden impulz *CLK*. Tím je ale KO_1 ihned vynulován. Signál *START* se může vrátit do stavu 0 kdykoliv. Impulz *X* tedy může být jakkoliv dlouhý, počínaje minimální délkou pro spolehlivé překlopení KO_1 .

Další užitečný obvod generuje jeden impulz o délce **jedné periody *CLK*** – viz *obr. 11.10*. Stav na vstupu *START* se s prvním impulzem *CLK* objeví na výstupu *Q1*. Při dalším *CLK* se stav *Q1* překopíruje do *Q2*. Přímé i negované výstupy KO_1 a KO_2 pak jsou jednoduše zpracovány. Po změně stavu *START* z 0 na 1 je generován impulz na výstupu *Y1*, po změně z 1 na 0 je generován impulz na výstupu *Y2*, a po jakékoli změně je generován impulz na výstupu *Y3*. Obvod pracuje jako **synchronní detektor hran** – náběžných na výstupu *Y1*, doběžných na výstupu *Y2*, a obou na výstupu *Y3*. Obvod s výstupem *Y3* lze nazvat jako **synchronní detektor změn**.



Obr. 11.10 Synchronní detektor hran

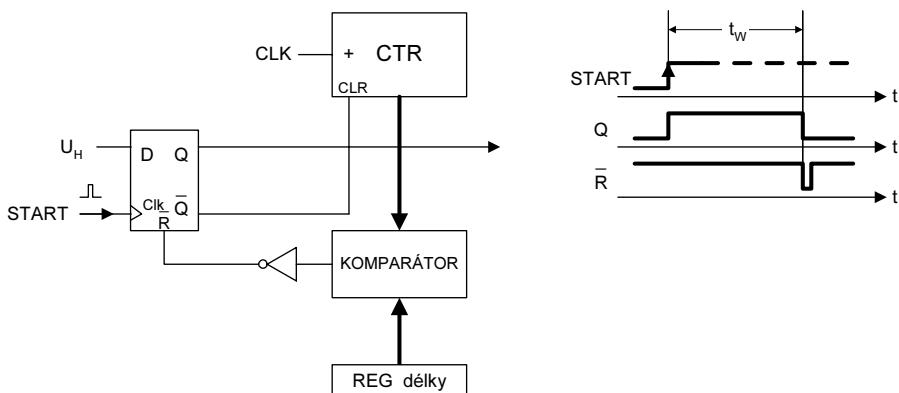
Další obvod na *obr. 11.11* slouží ke generaci impulzu o délce dané násobkem periody *CLK*. Je variantou předchozího zapojení s výstupem *Y₁*, ale na místě KO_2 je nyní posuvný registr. Jeho délka určuje i délku výstupního impulzu. Logický součet na vstupu a vazba z výstupu *Y* zajišťuje stav 1 klopného obvodu po celou dobu trvání výstupního impulzu bez ohledu na stav na vstupu *START*. Jedná se tedy o **synchronní variantu MKO-NR**.



Obr. 11.11 Obvod pro generaci jednoho impulzu o větší délce

Změna délky impulzu je v nejjednodušším případě možná výběrem odbočky na posuvném registru. Pro impulzy o délce mnoha period CLK je však tato realizace neúsporná a místo posuvného registru se využívá čítač. S posuvným registrem s N klopnými obvody lze totiž odpočítat až N hodinových impulzů, zatím co s čítačem až 2^N impulzů.

Princip ukazuje obr. 11.12. Klopny obvod je ve stavu 0, čítač CTR je trvale nulován a nepočítá. Spouštěcím impulzem se KO překlopí, nulování skončí a čítač počítá impulzy CLK . V okamžiku shody čísla v čítači s číslem, které bylo předem zapsáno do registru délky, je KO vynulován a tím je vynulován i čítač. Obvod je připraven ke generaci dalšího impulzu. Jelikož klopny obvod zůstává překlopen do stavu 1 po celou dobu trvání výstupního impulzu, nemají během této doby žádný vliv případné opakování spouštěcích impulzů. Jedná se tedy opět o **synchronní variantu MKO-NR**.

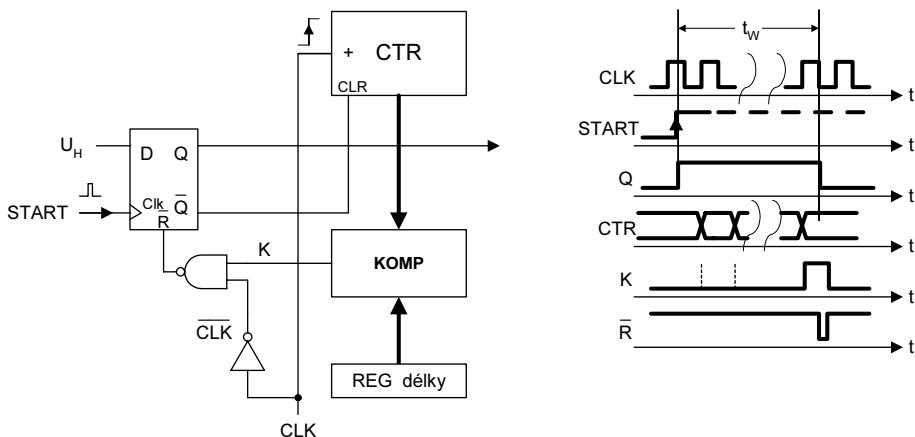


Obr. 11.12 Obvod pro generaci jednoho impulzu o velké délce

Uvedené zapojení jen ilustruje princip, prakticky by však pracovalo nespolehlivě. Důvodem jsou **falešné impulzy** na výstupu komparátoru vlivem **hazardních stavů**, ke kterým dochází při změnách stavu na vstupech komparátoru. I když tyto falešné impulzy jsou jen krátké, klopny obvod jimi může být vynulován v nesprávném okamžiku.

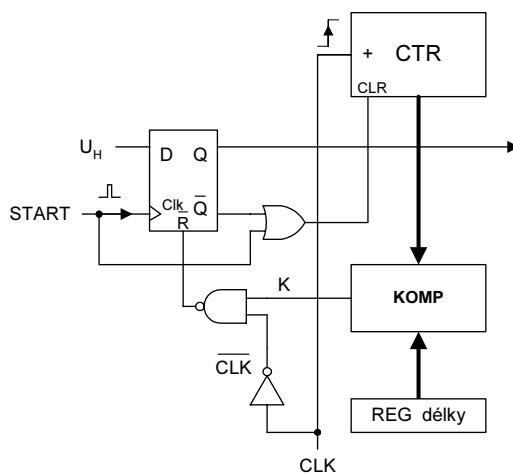
Další obr. 11.13 ukazuje úpravu obvodů tak, aby byl **vyloučen vliv hazardů**. V časovém diagramu na obrázku vpravo jsou falešné impulzy na výstupu komparátoru K vyznačeny

jako slabé čárkované čáry. Signál pro vynulování KO je vytvořen jako součin signálu K s negovaným signálem CLK – proto může být aktivní jen ve druhé polovině taktu CLK , když falešné impulzy odezněly.



Obr. 11.13 Vyloučení vlivu hazardů v komparátoru

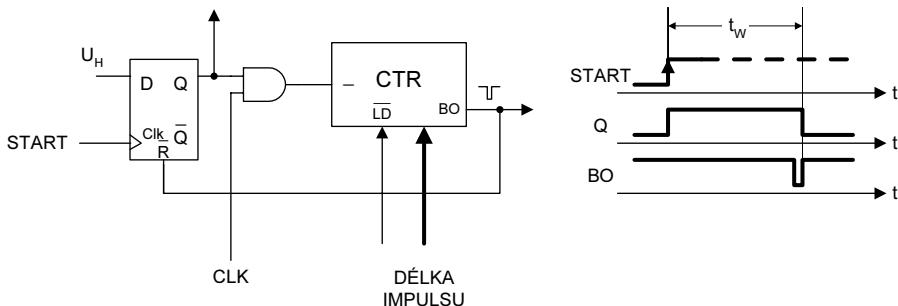
Obr. 11.14 ukazuje princip synchronní varianty MKO-R (znovu spustitelného). K tomu postačí malá změna předchozího obvodu. Spouštěcí impulzy zde mohou vynulovat čítač kdykoliv, tedy i během trvání výstupního impulzu. Tím začne odpočítávání period CLK vždy znova od nuly.



Obr. 11.14 Obvod pro generaci jednoho impulzu – znovu spustitelný

Jiné zapojení synchronního MKO s čítačem je na obr. 11.15. Zde není využíván komparátor. Čítač v tomto případě počítá dolů od přednastavené hodnoty. Při dopočítání do nuly vznikne na jeho výstupu (BO) impulz, který vynuluje KO. Před generací dalšího impulzu

musí být čítač **znovu přednastaven** na hodnotu „DÉLKA IMPULSU“ signálem \overline{LD} . Z tohoto důvodu zapojení není vhodné pro samostatnou funkci, je ale velmi často využíváno v periferických (časovacích) obvodech mikropočítačů. Čítač má svoji adresu a počáteční hodnota se do něj zapisuje programově, z datové sběrnice procesoru. Je tedy **softwarově** nastavován, ale **hardwarem** spouštěn vnějším signálem *START*. Na výstupu BO dává krátký impulz na konci intervalu, na výstupu Q pak dává impulz po celou dobu intervalu.



Obr. 11.15 Obvod pro generaci jednoho impulzu s přednastavitele čítačem

Výhodou synchronních MKO je přesnost, velký rozsah časování a možnost programového nastavení délky impulzu. Generátory hodinových impulzů jsou zpravidla řízeny kryštalem a jsou velmi přesné. Spouštěcí impulz však není synchronizován a jeho hrana se může vyskytnout v libovolném okamžiku vzhledem k taktům *CLK*. Proto synchronní MKO nemůže **současně** splnit dva požadavky:

- výstupní impulz má začínat současně s hranou spouštěcího impulzu,
- délka výstupního impulzu má být přesně rovna násobku periody *CLK*.

Vhodnou úpravou obvodů MKO lze vyhovět buď jedné, nebo druhé podmínce, ale vždy s jistou chybou. V závislosti na okamžiku výskytu hrany spouštěcího impulzu se absolutní chyba v časování pohybuje v mezích **jedné periody *CLK***. Relativní chyba pak bude záležet na nastaveném počtu period *CLK* a s jejich větším počtem se úměrně snižuje.

11.3 Impulzní šířkový modulátor

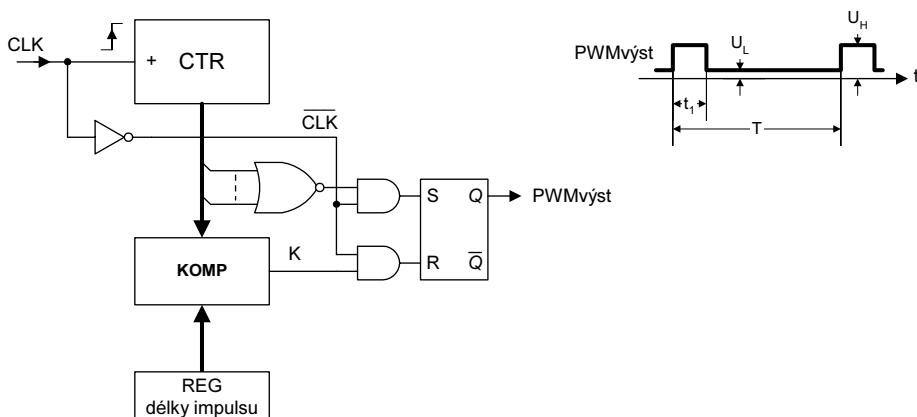
Impulzní šířková modulace **PWM** (angl. Pulse Width Modulation) je jednoduchý způsob převodu čísla na analogový signál (střední hodnotu napětí). Periodicky generované impulzy mají periodu *T* a délku (šířku) t_1 , takže v ideálním případě je střední hodnota napětí:

$$U_{\text{stř}} = U_1 \cdot \frac{t_1}{T},$$

kde U_1 je amplituda impulzu. Vytvoření střední hodnoty napětí není záležitostí modulátoru, nýbrž až demodulátoru na místě určení signálu. Jako demodulátor je používán filtr typu dolnofrekvenční propusti, který impulzní průběh vyhladí. Mnoho zařízení, řízených signálem PWM, však má samo od sebe dostatečnou setrvačnost, takže filtr ani není zapotřebí

(elektromagnety, elektromotory, žárovky, elektrotepelná zařízení, atd.). Filtr či samotná seřvačnost zařízení je velmi účinným opatřením pro potlačení rušení v přenášeném PWM signálu.

Jednoduchý synchronní modulátor ukazuje obr. 11.16.



Obr. 11.16 Impulzní šířkový modulátor

Čítač CTR s **modulem M** nepřetržitě čítá impulzy a ve stavu 0 překládí klopný obvod RS do stavu 1. Po dosažení stavu shodného s **číslem N** v registru délky impulzu je komparátorem klopný obvod vynulován – tím je dána délka impulzu $t_1 = N \cdot T_{CLK}$ (T_{CLK} je perioda hodinových impulzů). Čítač však počítá dál, po posledním stavu v jeho M-stavovém cyklu následuje stav 0 a tím je KO opět překlopen do stavu 1. Platí tedy, že $T = M \cdot T_{CLK}$. Střední hodnota napětí je proto úměrná poměru N/M .

Počtem stavů čítače v periodě T , neboli modulem čítače, je dána **relativní chyba** poměru t_1/T . Běžně se používá 8bitový čítač s modulem 256, takže relativní chyba činí přibližně 0,4 %. Každý další bit čítače snižuje chybu na polovinu.

Konečný počet stavů čítače není jediným zdrojem nepřesnosti. Jelikož výstupním signálem je číslicový signál s úrovněmi U_H a U_L , je přesný vztah pro $U_{stř}$:

$$U_{stř} = (U_H - U_L) \cdot \frac{N}{M} + U_L$$

Jelikož napětí U_H a U_L nepředstavují konstanty, ale celá pásma hodnot, je $U_{stř}$ zřejmě vyjádřena značně nepřesně. Pro dosažení vyšší přesnosti proto demodulátor musí převádět číslicový signál PWM na signál s přesněji definovanými úrovněmi prostřednictvím přesných spínačů.

426410



PAMĚTI

Účelem paměti je uložení velkého množství dat. Existuje mnoho fyzikálních jevů, na jejichž základě lze (někdy zatím jen teoreticky) zkonztruovat paměť. Významného rozšíření se však zatím dočkaly jen paměti polovodičové v technologii **CMOS** a paměti magnetické. V tomto textu budou rozebrány různé typy polovodičových pamětí. Vývoj paměti vždy probíhal velmi intenzivně. Všechny jejich podstatné parametry, jako je vybavovací doba a kapacita, se rychle mění. Aktuální hodnoty je třeba hledat ve firemních materiálech.

12.1 Rozdelení paměti

Paměti podléhají intenzivnímu vývoji a je jich mnoho typů. Nové typy mají stálé rostoucí kapacitu a jsou rychlejší. Současně se zpravidla komplikuje jejich ovládání. Bude proto účelné provést na začátku rozdelení paměti podle různých hledisek.

Podle způsobu adresace:

- Paměť s **adresovým výběrem** – data jsou uložena na stabilních adresách a dodání adresy je nutné pro operaci čtení i zápisu.
- Paměti **bezadresové** – zásobníková paměť LIFO (angl. *Last In – First Out*), nebo paměť fronty FIFO (angl. *First In – First Out*). Adresy jsou generovány automaticky vnitřními obvody v **pevně stanoveném pořadí**, buď v opačném než zápis (LIFO), nebo ve stejném jako zápis (FIFO). Proto vnější adresa neexistuje.

Podle trvanlivosti dat po vypnutí napájení:

- Paměť **energeticky závislá** (angl. *volatile*) – data se ztrácí po vypnutí napájení.
- Paměť **energeticky nezávislá** (angl. *non-volatile*) – data se neztrácí po vypnutí napájení. Toto hledisko se vztahuje na **samotné** paměťové obvody. Celý paměťový systém totiž lze vybavit záložní baterií a i paměti energeticky závislé pak neztrácejí data.

Podle snadnosti a rychlosti zápisu:

- Paměť s možností čtení i zápisu – **RWM** (angl. *Read-Write Memory*) – zápis i čtení jsou stejně snadné a rychlé. Paměť RWM je též (dokonce častěji) označována zkratkou **RAM** (z angl. *Random Access Memory*), což doslova znamená „paměť s libovolným přístupem“. Tato zkratka je významově dosti posunutá, avšak všeobecně se ujala.
- Paměti **permanentní** – ROM (angl. *Read Only Memory*) – původně jen pro čtení. Postupně s rozvojem technologie byly vyvinuty typy, umožňující více nebo méně snadný zápis, vždy však podstatně pomalejší než čtení. Paměť je energeticky nezávislá.

12.2 Důležité parametry paměti

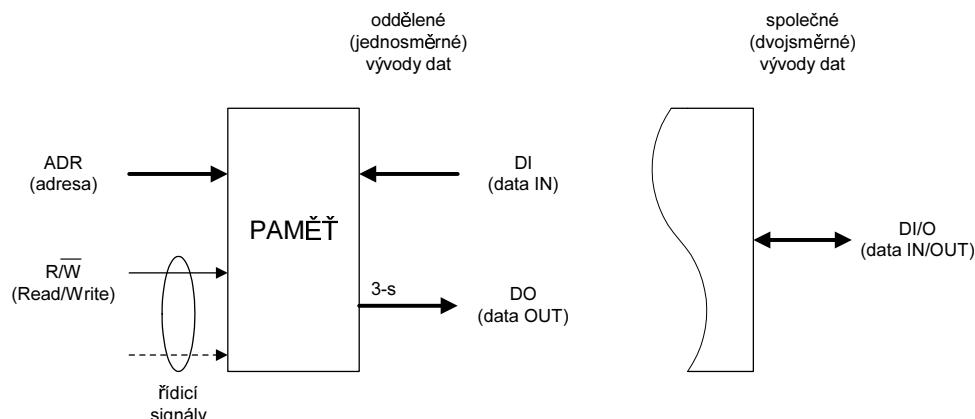
Důležitým údajem, charakterizujícím rozsah paměti, je její **kapacita**. Udává se v celkovém počtu bitů. Jednotkou je kilobit Kb (zde $1\text{ K} = 2^{10} = 1024$), megabit Mb (zde $1\text{ M} = 2^{20} = 1048576$), nebo gigabit Gb (zde $1\text{ G} = 2^{30}$). Kapacitu paměti lze udávat i v počtu slov, např. 256 KB (zde „B“ znamená „byte“). Důležitá je i **organizace paměti**, tj. počet adres

a délka slova – zpravidla 1 nebo 2 byte. Tak např. údaj $1M \times 8$ znamená paměť o 2^{20} adresách, a na každou adresu je slovo o délce 1 byte. Celková kapacita by byla 8 Mb, případně 1 MB.

Důležitým údajem, charakterizujícím rychlosť paměti, je **vybavovací doba** (značena t_{AC}). Uvádí se v nanosekundách ($1\text{ns} = 10^{-9}\text{ s}$). Je to doba od začátku čtecího cyklu do vydání platných dat. Začátkem cyklu může být změna adresy, ale podle typu paměti i změna jiných signálů. Přesnou definici vybavovací doby je třeba nalézt v údajích výrobce. V závislosti na typu paměti existuje řada dalších dynamických parametrů.

12.3 Paměti paralelní a sériové

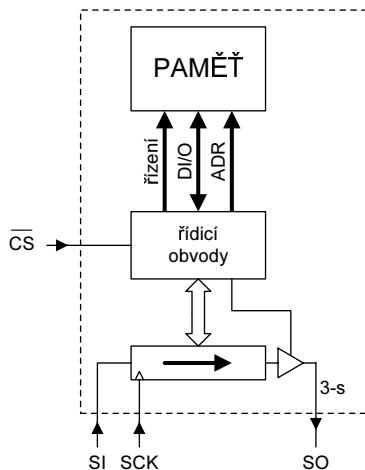
Ke spolupráci s dalšími obvody v číslicovém systému se u pamětí využívají typické soustavy signálů. *Obr. 12.1* ukazuje paralelní přístup, kdy adresa (ADR) i data jsou vícebitové. Vstupní data (DI) a výstupní data (DO) mohou být rozdělena – jednosměrná, nebo častěji sdružená – dvojsměrná (DI/O). Soustava řídicích signálů určuje činnost paměti (čtení–zápis, atd.), řídí směr přenosu dat, atd. Podle typu paměti se soustavy řídicích signálů mohou vzájemně lišit.



Obr. 12.1 Paměť s paralelním přístupem

Vedle paralelního přístupu existuje i přístup sériový. Sériové paměti jsou využívány tam, kde je vhodné snížit počet spojů (zpravidla z ekonomických důvodů) a pomalejší přenos není na závadu. Typickou podobu sériové paměti ukazuje *obr. 12.2*.

Jedná se o tzv. **třívodičové spojení** (výběrový signál \overline{CS} se nepočítá), zavedené pro počítačové periferie SPI. Do vnitřního posuvného registru se ze vstupu SI bit za bitem přenese povel, kterým je dána následující operace paměti (čtení, zápis, atd.). Pak se přenese adresa. Jedná-li se o zápis dat, přenesou se ještě zapisovaná data. Řídicí obvody provedou požadovanou operaci s vnitřní pamětí. Jedná-li se o operaci čtení, jsou ihned po zadání adresy vydávána sériová data na výstupu SO. Výstupní třístavový obvod je ovládán řídicími obvody tak, že kromě operace čtení je vždy ve vysokoimpedančním stavu. Hodinové impulzy pro posuvný



Obr. 12.2 Paměť se sériovým přístupem

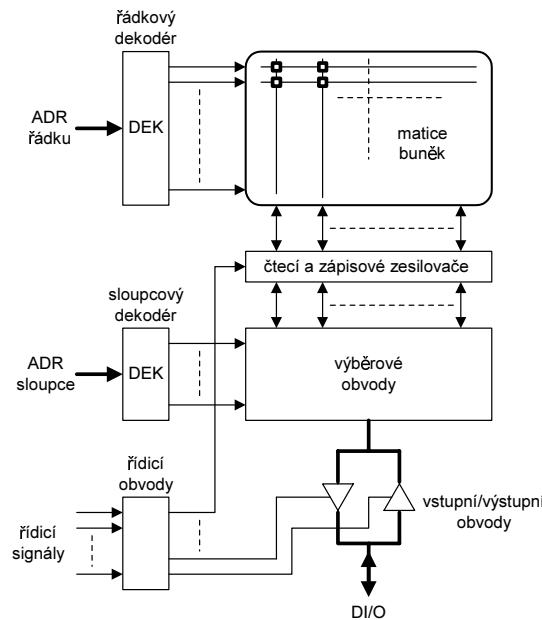
registr musí být generovány vnějšími obvody. Sériová paměť má pouzdro s malým počtem vývodů. Nevýhodou je však zpoždění způsobené přenosy po jednotlivých bitech. Vedle třívodičového spojení existuje i spojení **dvouvodičové**, zavedené pro počítačové periferie IIC. U obvodů tohoto typu se nepoužívá výběrový signál a sériová data jsou dvojsměrná.

12.4 Vnitřní uspořádání paměti s adresovým výběrem

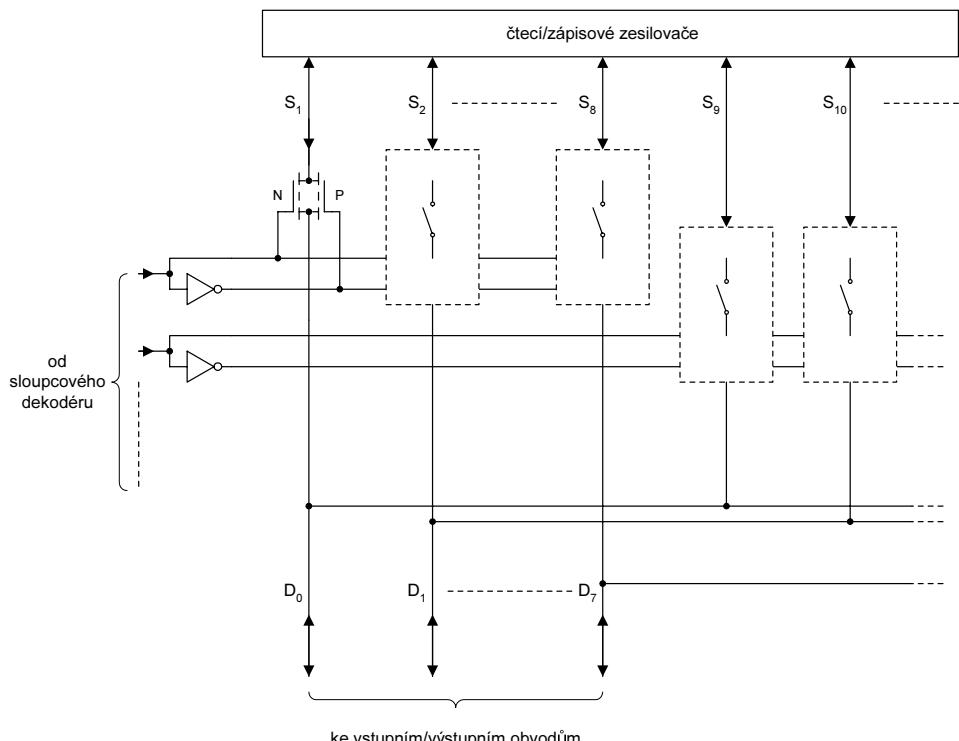
Obr. 12.3 ukazuje vnitřní obvody paměti bez detailů. Jádrem je **matice paměťových buněk**, umístěných v křížení rádkových a sloupcových vodičů. Jedna část bitů adresy vybírá prostřednictvím **rádkového dekodéru** vždy jeden z rádků maticy. Tím jsou vybrány všechny buňky daného rádku a připojeny na sloupcové vodiče. Druhá část bitů adresy je zavedena do **sloupcového dekodéru**. Ten ovládá **výběrové obvody**, což je skupina CMOS spínačů zapojených jako multiplexer/demultiplexer. Na výběrové obvody pak navazují **vstupní a výstupní obvody**. Čtecí a zápisové **zesilovače** jsou nutné jen u některých typů pamětí. Jejich úlohou je všeobecně zesílení signálu z paměťových buněk, a u dynamických pamětí i obnovování obsahu buněk.

Rídicí signály určují pracovní cyklus paměti (čtení–zápis, apod.), ovládají třístavové výstupní obvody, vyvolávají úsporný režim sníženého příkonu, atd. Soustava rídicích signálů je charakteristická vždy pro daný typ paměti a bude vysvětlena v dalším textu.

Spínače ve výběrových obvodech jsou tak seskupeny, že sloupcový dekodér vybírá celé skupiny sloupců – tomu odpovídá délka slova paměti. Obr. 12.4 ukazuje princip výběrových obvodů pro 8bitové slovo. Funkce spínačů je zcela symetrická a výběrové obvody proto mohou sloužit pro oba směry přesunu dat, tj. pro čtení i pro zápis. Jedná se vlastně o skupinový multiplexer/demultiplexer.



Obr. 12.3 Vnitřní uspořádání paměti s adresovým výběrem

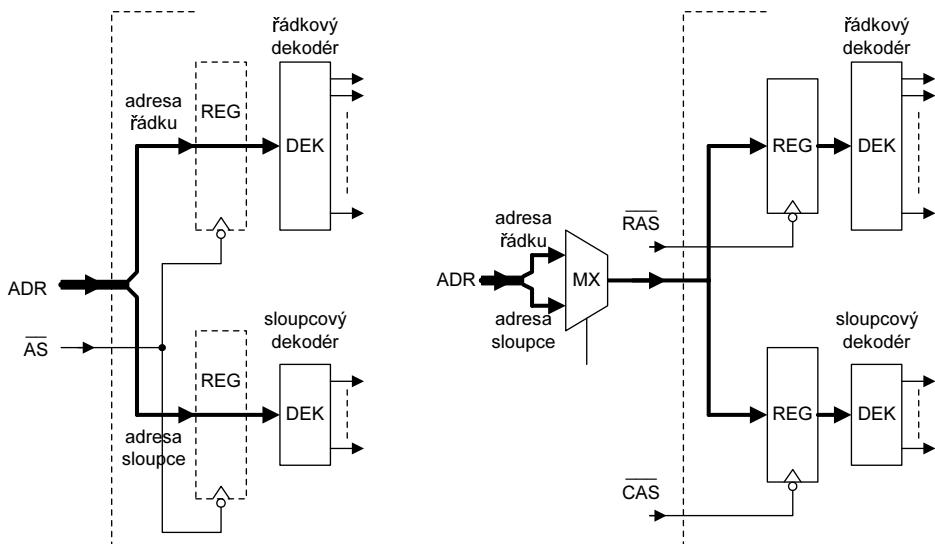


Obr. 12.4 Princip výběrových obvodů sloupců

Zavedení adres do řádkového a sloupcového dekodéru je v obr. 12.3 jen naznačeno a u konkrétních typů paměti se liší. V podstatě se využívají tři možnosti – viz obr. 12.5. Na obrázku vlevo je nejjednodušší a nejčastější možnost – do paměti (její hranice je naznačena čárkovaně) je přiváděna **současně celá adresa** a až uvnitř se rozvětuje na dvě složky, které jsou zavedeny do příslušných dekodérů. To zajišťuje nejmenší zpoždění. Typicky je tento způsob zavádění adresy využíván **u statických pamětí SRAM** a **u permanentních pamětí** (viz další text). Jistou nevýhodou tohoto řešení je velký počet adresových vývodů pouzdra u paměti větší kapacity.

Jako druhá možnost a **jen výjimečně** jsou před dekodéry předřazeny registry (v obr. 12.5 vlevo čárkovaně vyznačeny), do kterých se adresa zapíše impulzem \overline{AS} (Address Strobe). Některí výrobci tak zjednoduší konstrukci paměťového systému v počítačích s multiplexní sběrnicí.

V pravé polovině obr. 12.5 je vyznačena třetí možnost, kdy adresa je do paměti zavedena **postupně ve dvou krocích**. Paměť je vybavena dvěma registry, ovládanými individuálně. Nejprve se prvá část adresy (adresa řádku) zapíše do řádkového registru impulzem RAS (Row Address Strobe), pak se druhá část (adresa sloupce) zapíše do sloupcového registru impulzem CAS (Column Address Strobe). Obě složky adresy je nutné ve vhodném čase přepínat v obvodech mimo vlastní paměť (v adresovém multiplexeru). Celkové zpoždění, vzniklé postupným zápisem do adresových registrů, je přípustné jen tehdy, když zavádění druhé části adresy (tj. adresy sloupce) probíhá v době, kdy v paměti probíhají takové vnitřní děje, které ještě adresu sloupce nepotřebují.



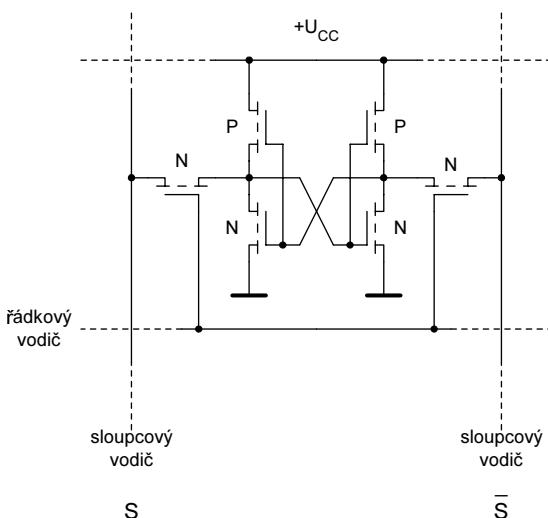
Obr. 12.5 Možnosti zavedení adresy do paměti – vlevo jako celek, vpravo postupně

Typicky je tento způsob zavádění adresy využíván **u dynamických pamětí DRAM** o velké kapacitě (viz další text). Pro zavedení adresy ve dvou krocích je zapotřebí jen poloviční počet vývodů pouzdra, což snižuje cenu integrovaných paměťových obvodů a vyvažuje komplikace spočívající v potřebě vnějších podpůrných obvodů.

12.5 Statické paměti RAM

Ve statických pamětích RAM (SRAM) se k zapamatování číslicových (dvojstavových) signálů využívá kladná zpětná vazba. Nejjednodušším obvodem s kladnou zpětnou vazbou je **bistabilní klopny obvod**. Ten lze vnějším signálem během zápisu ovládat (překlápet) a naopak jeho výstupní signál lze během čtení dalšími obvody zpracovávat. Takto zapamatovaný jeden bit informace zůstane v klopňém obvodu uložen do té doby, než je eventuálně přepsán, nebo do vypnutí napájení.

Celkové uspořádání odpovídá obr. 12.3, obr. 12.4 a obr. 12.5, adresa je do paměti zaváděna jako celek a zpravidla bez registrů. Tím je minimalizováno zpoždění. Paměťová matic je složena ze šestitransistorových buněk dle obr. 12.6. Klopny obvod je implementován nejjednodušším způsobem jako dva invertory. Jeho výstupy jsou dalšími dvěma tranzistory, řízenými řádkovým vodičem, připojovány na dva sloupce. Rozdíl napětí mezi oběma sloupcy je při čtení vyhodnocen rozdílovým čtecím zesilovačem. Při zápisu je naopak přes dvojici sloupcových vodičů a otevřené spínače klopny obvod překlopen do patřičného stavu.



Obr. 12.6 Buňka statické paměti RAM

Soustava řídicích signálů SRAM je jednoduchá:

- **výběrový signál \overline{CS}** (Chip Select) stavem 1 blokuje zápis do paměti, uvádí výstupy do vysokoimpedančního stavu, a omezuje příkon paměti tím, že vyvolá snížení vnitřního napájecího napětí matice na hodnotu, při které ještě klopny obvody spolehlivě udrží zapsaná data. Při stavu 0 se naopak paměť uvede do provozu – to však vyžaduje relativně dlouhou dobu. Pro rychlé připojování a odpojování výstupů se tento signál nehodí.
- signál \overline{OE} (Output Enable) působí jen na výstupní obvody – stavem 1 je uvádí do vysokoimpedančního stavu a stavem 0 je připojuje. V obou případech jen v relativně krátké době. Lze jej přibližně charakterizovat jako „čtecí impulz“.

- signál \overline{WR} (Write) nebo též \overline{WE} (Write Enable) rozhoduje o operaci zápisu (stavem 0) nebo čtení (stavem 1). Lze jej přibližně charakterizovat jako „**zápisový impulz**“. Při stavu 0 jsou výstupní obvody blokovány bez ohledu na \overline{OE} .

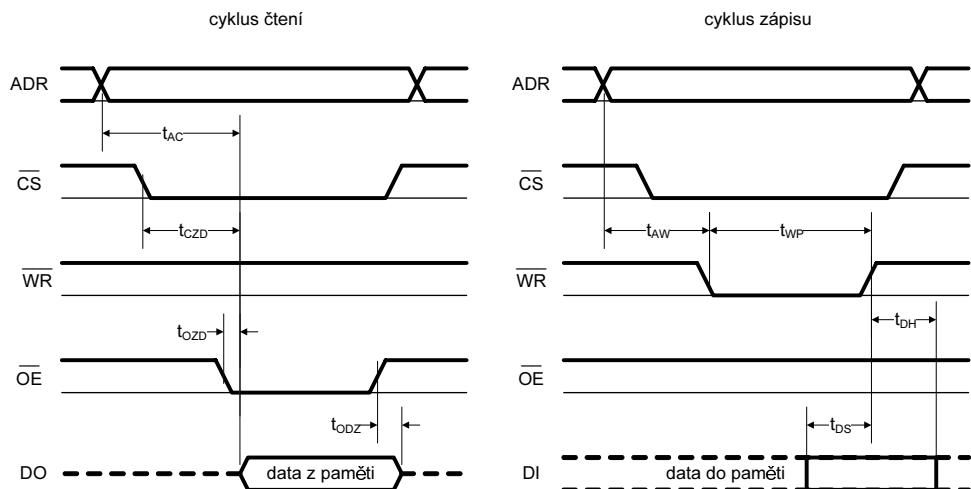
Tab. 12.1 shrnuje funkci uvedených řídicích signálů (– značí libovolný stav).

Tab. 12.1 Význam řídicích signálů SRAM

CS	\overline{OE}	\overline{WR}	operace	DI/O	napájecí proud
H	–	–	žádná	vysoká impedance	snížený
L	L	H	čtení	DO	plný
L	–	L	zápis	DI	plný
L	H	H	žádná	vysoká impedance	plný

Časový diagram a hlavní dynamické parametry ukazuje *obr. 12.7*. V cyklu čtení je podstatná zvláště **vybavovací doba** (t_{AC}), doba odblokování od \overline{CS} (t_{CZD}), a podstatně kratší doba odblokování od \overline{OE} (t_{OZD}) a zablokování od \overline{OE} (t_{ODZ}). V cyklu zápisu je podstatná doba odstupu zápisového impulzu od změny adresy (t_{AW}), délka zápisového impulzu (t_{WP}), a předstih a přesah zapisovaných dat kolem konce zápisového impulzu (t_{DS} a t_{DH}). Interval, po který musí být zapisovaná data stabilní, je orámován tlustou čarou.

Signál \overline{CS} se nemusí měnit v každém cyklu – může být trvale ve stavu 0. Činnost SRAM se tím urychlí, neboť odpadá zpoždění t_{CZD} . Paměť pak ale má větší spotřebu, neboť odpadají doby blokování se sníženým příkonem.



Obr. 12.7 Průběh cyklu čtení (vlevo) a cyklu zápisu (vpravo) paměti SRAM

Paměti SRAM jsou velmi rychlé, vybavovací doby pod 10 ns nejsou výjimečné.

Následující výpis kódu ukazuje realizaci asynchronní paměti RAM s organizací $64k \times 8$ bitů v jazyce VHDL. Vlastní paměť je realizována dvouozměrným polem typu `std_logic_vector`.

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

entity ram_async is
    port (
        adr : in std_logic_vector (15 downto 0);
        di : in std_logic_vector (7 downto 0);
        cs : in std_logic;
        oe : in std_logic;
        wr : in std_logic;
        do : out std_logic_vector (7 downto 0)
    );
end ram_async;

architecture behav of ram_async is
    type mem is array (0 to 65535) of std_logic_vector (7 downto 0);
    signal ram_block : mem;
begin
    -- realizace zapisu do ram procesem
    -- proces se spustí při každé změně wr nebo di
    process (wr, di)
    begin
        -- zapis do paměti probíhá pouze při aktivním cs a wr
        if cs = '0' and wr = '0' then
            ram_block(to_integer(unsigned(adr))) <= di;
        end if;
    end process;

    -- čtení probíhá při aktivním cs a oe, jinak dava na vysokou impedanci
    do <= ram_block(to_integer(unsigned(adr))) when (cs = '0' and oe = '0')
        else (others => 'Z');
    end behav;
```

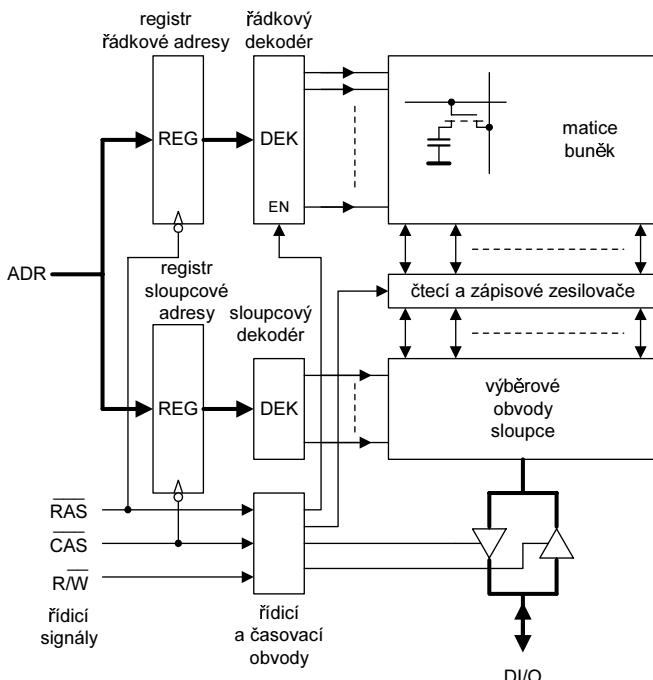
Výpis 12.1 Asynchronní paměť RAM

12.6 Dynamické paměti RAM

V dynamických pamětích RAM (DRAM) se k zapamatování číslicových (dvojstavových) signálů využívá **náboj na kondenzátoru**. Kondenzátor, nabity či vybitý přes jednoduchý spínač (MOSFET) během zápisu, udrží náboj po jistou dobu. Napětí na kondenzátoru lze během čtení porovnat s referenčním napětím a rozlišit tak stav 0 a 1. Tím je realizována paměť na jeden bit informace. Nabity paměťový kondenzátor se časem vybije. Aby nedošlo ke ztrátě informace, je nutno náboj dostatečně často **obnovovat**.

Celkové uspořádání paměti DRAM ukazuje obr. 12.8. Adresa se do paměti zavádí ve dvou krocích. Nejprve se adresa řádku zapíše do řádkového registru impulzem \overline{RAS} , pak se adresa sloupce zapíše do sloupcového registru impulzem \overline{CAS} . Paměťová matici obsahuje

buňky v nejjednodušší podobě. Na stejné ploše čipu proto má DRAM větší kapacitu, než by měla SRAM se složitější šestitransistorovou buňkou. Tomu odpovídá i **nižší cena za jeden bit** u paměti DRAM.

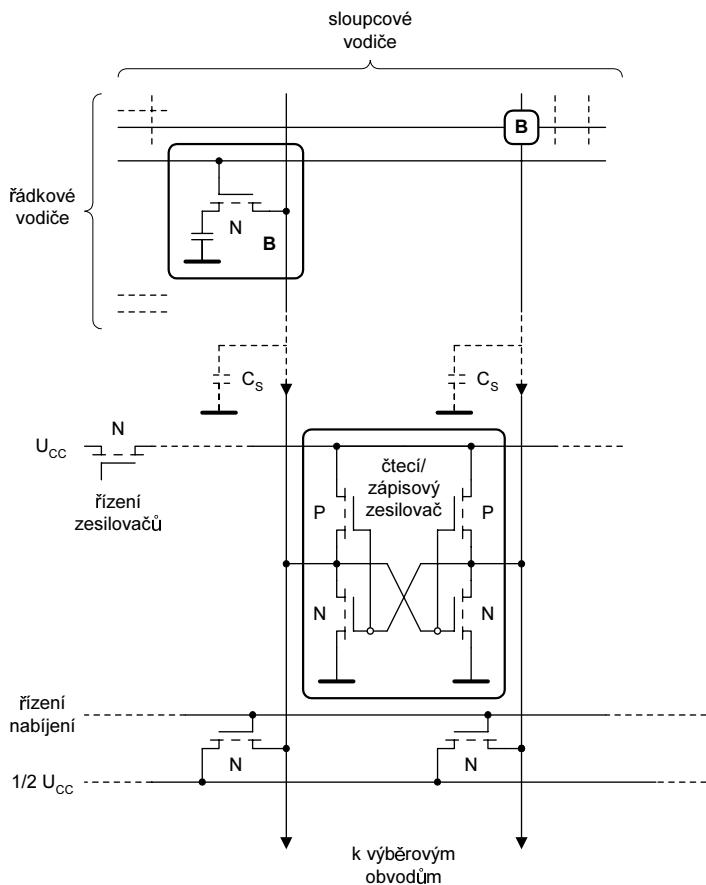


Obr. 12.8 Vnitřní uspořádání paměti DRAM

Problém však představuje čtení stavu buňky. Paměťový kondenzátor totiž má velmi malou kapacitu, řádově desítky FF (femtofarad, 10^{-15} F). Naproti tomu kapacita sloupcového vodiče proti zemi je mnohem větší. Připojením paměťového kondenzátoru na sloupcový vodič tak vzniká kapacitní dělič, který sníží napěťový skok úměrně poměru kapacit – tedy velmi podstatně.

Výrobci DRAM vyvinuli několik vtipných řešení čtecích zesilovačů. Velmi rozšířená varianta je na obr. 12.9. Čtecí zesilovač je konstruován jako klopny obvod, uvedený ve vhodném okamžiku do rovnovážného stavu (tj. shodného napětí na obou výstupech). Shodné napětí se zajistí nabítím sloupcových vodičů o kapacitě C_S na polovinu napájecího napětí U_{CC} přes řízený spínače – tzv. **přípravné nabíjení** (angl. precharge). Pak se na čtecí zesilovače přivede napájecí napětí (U_{CC}). V důsledku kladné zpětné vazby překlopení KO závisí i na nepatrné nesymetrii obou napětí. Nesymetrie je vyvolána tak, že se na jeden z obou sloupcových vodičů připojí paměťová buňka se svým paměťovým kondenzátorem, zatím co na druhý ne. Ten vlastně slouží v daném okamžiku jako zdroj referenčního napětí. Po úplném překlopení je klopny obvod již zdrojem napětí o maximální amplitudě ($U_L = 0$, $U_H = U_{CC}$) pro sloupcový vodič. Přes stále ještě otevřený spínač v buňce se její paměťový kondenzátor nabije (či vybije) na krajní hodnoty napětí. Informace v buňce je tak obnovena. Buňka na referenčním sloupci je zdánlivě nevyužitá – bude však využita při jiné řádkové adrese, kdy

se funkce sloupců obráti. Střídavé využívání sloupců jako paměťových a jako referenčních je zajištěno vhodným spojením buněk s rádky a sloupcí matic – buňky na lichých rádcích jsou spojeny s lichými sloupcí a buňky na sudých rádcích se sudými sloupcí. Čtecí zesilovače obsluhují vždy páry sloupců „sudý-lichý“.

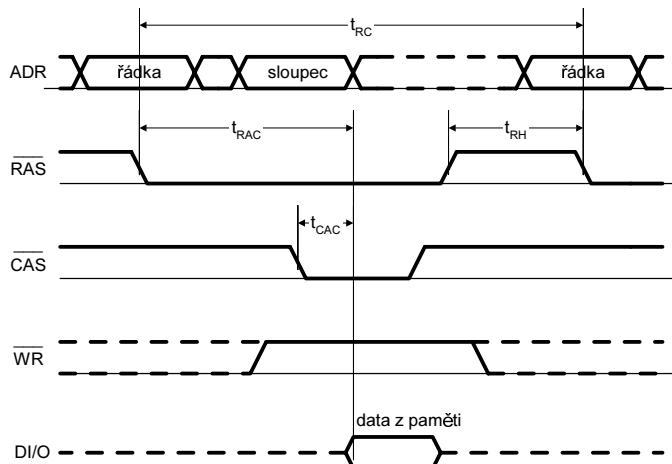


Obr. 12.9 Funkce čtecích zesilovačů DRAM

Uvedené usporádání umožňuje čtení a současně i obnovení informace vždy **na všech buňkách rádku**. Výsledek čtení zůstává uložen ve čtecích zesilovačích. Na ně navazují výběrové obvody a vstupní/výstupní obvody podle obr. 12.3 a obr. 12.4.

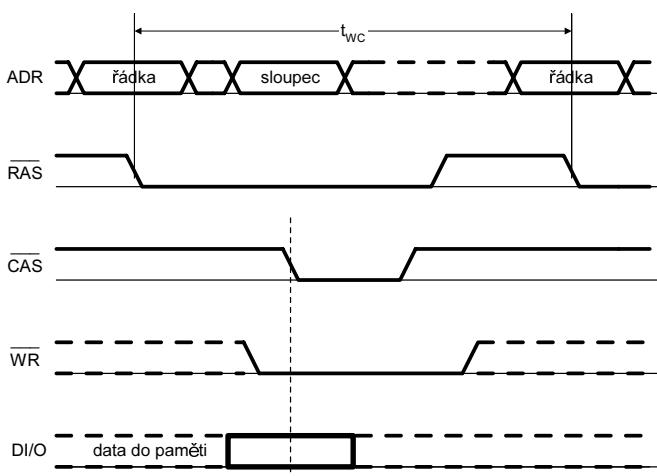
Dění v paměti DRAM je podstatně složitější, než u paměti SRAM. Čtecí cyklus je zahájen hranou (doběžnou) signálu RAS – viz obr. 12.8. Tím je zapsána rádková adresa do rádkového registru a současně je ukončeno nabíjení sloupců. Do čtecích zesilovačů je zavedeno napájecí napětí. Následně je rádkovým dekódérem vybrána řádková adresa a hranou (doběžnou) signálu CAS je zapsána do sloupcového registru. Sloupcový dekódér s výběrovými obvody vybere skupinu čtecích zesilovačů (slovo z paměti). Následně jsou odblokovány výstupní třístatové obvody a zapamatovaná informace se dostává

na výstupy paměti. S návratem signálu \overline{CAS} do stavu 1 se výstupní obvody blokují (uvedou se do vysokoimpedančního stavu). S návratem signálu RAS do stavu 1 se blokuje řádkový dekodér a začnou se nabíjet sloupcové vodiče, takže paměť je připravena k dalšímu cyklu. Posloupnost vhodné časovaných řídicích signálů je generována vnitřními řídicími a časovači obvodů. Celý průběh čtecího cyklu ukazuje obr. 12.10.



Obr. 12.10 Průběh čtecího cyklu DRAM

Obrázek ukazuje důležité doby, jako je t_{RAC} (vybavovací doba od hrany \overline{RAS}), t_{CAC} (vybavovací doba od hrany CAS), t_{RH} (doba nabíjení sloupců), a t_{RC} (doba čtecího cyklu). Zápisový povel WR musí být neaktivní, tj. ve stavu 1, kolem doběžné hrany CAS . Čtená data jsou přítomna na výstupech s malým zpožděním po dobu, kdy $\overline{CAS} = 0$, tedy jen po zlomek doby cyklu. Rovněž celá vybavovací doba t_{RAC} je jen částí doby cyklu. Běžně se t_{RAC} pohybuje kolem 50 ns, t_{RC} pak je přes 100 ns.



Obr. 12.11 Průběh zápisového cyklu DRAM

Průběh zápisového cyklu ukazuje obr. 12.11. Zápisový povel \overline{WR} musí být aktivní (ve stavu 0) kolem doběžné hrany \overline{CAS} . Interval, po který musí být zapisovaná data stabilní, je orámován tlustou čarou.

Z principu paměťové buňky dynamické paměti vyplývá nutnost obnovování informace (angl. *refresh*). Interval obnovování, tj. doba udržení neporušených dat záleží na teplotě čipu, na typu paměti a na technologii výroby. Při teplotě kolem 20°C to mohou být i jednotky sekund – ale bez záruky. Katalogové údaje jsou podstatně opatrnější. K obnovení informace na celé řadce dojde automaticky při čtení či zápisu s adresou dotyčné rádky. Během intervalu obnovování je však nutné obnovit informaci na všech rádkách matice. Do normální činnosti paměti se proto vkládají obnovovací cykly. Jsou to zkrácené cykly, omezené jen na vložení adresy rádku, doprovázené impulzem RAS . Tím je vyvolána činnost čtecích zesilovačů a tím i obnovení obsahu buněk daného rádku. Vkládání adresy sloupce je zbytečné, neboť žádný výstup dat z paměti není v obnovovacím cyklu předpokládán.

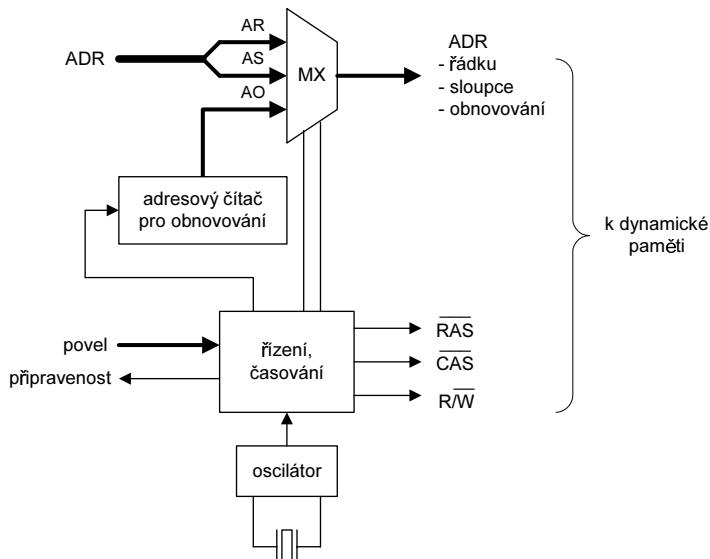
Obnovovací cykly lze rozložit v čase různým způsobem:

- Před uplynutím intervalu obnovování se provedou postupně – jeden za druhým – obnovovací cykly na všech adresách rádku. Toto **blokové (dávkové) obnovování** (angl. *burst refresh*) vyřazuje paměť z normální činnosti vždy na poměrně dlouhou dobu a v některých případech může být na závadu např. přesného časového průběhu programu v počítači.
- Obnovovací cykly jsou pravidelně rozloženy tak, že všechny proběhnou během intervalu obnovování. Toto **rozložené obnovování** (angl. *time-distributed refresh*) sice probíhá často, ale nikdy nevyřazuje paměť na dlouhou dobu.
- Obnovovací cykly jsou prokládány mezi čtecí a zápisové cykly v době, kdy paměť by nebyla využívána. Toto **skryté obnovování** (angl. *transparent refresh*) je typicky řízeno samotným procesorem v počítači.
- V některých případech může obnovování **zcela odpadnout**. Tato situace nastává u systémů, kde jsou data z paměti stále opakovaně čtena (např. u videopaměti).

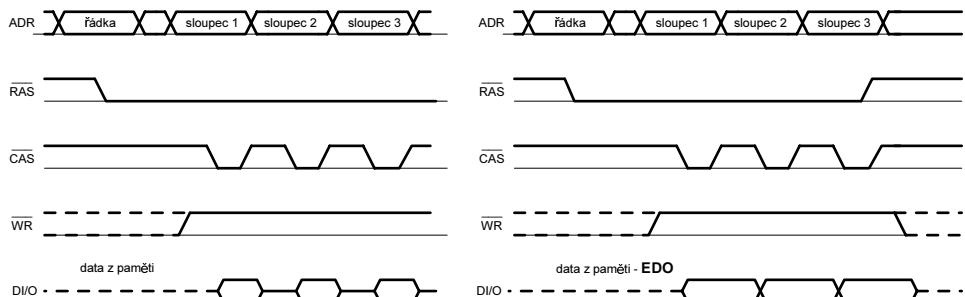
Pro postupné dodávání rádkových adres pro obnovování je vždy využíván čítač adres. Spolu s adresovým multiplexerem a řídícími obvody tvoří **řadič dynamické paměti** – viz obr. 12.12.

Řadič usnadňuje manipulaci s pamětí. Systém, využívající paměť, zadává řadiči typ paměťové operace (povel). Řadič vytvoří potřebnou sekvenci impulzů pro paměť. Přesné časování impulzů, které je pro plné využití dynamické paměti kritické, zajišťuje krystalový oscilátor. Řadič zajišťuje i obnovování obsahu paměti některou z výše uvedených metod.

Z obr. 12.10 a obr. 12.11 je vidět rozdíl mezi vybavovací dobou paměti a dobou cyklu. Doba cyklu omezuje frekvenci opakovaných operací a vybavovací doba je špatně využita. Dynamické paměti umožňují za jistých podmínek podstatné zrychlení opakovaných operací. Jednou z metod je **čtení po stránkách**. Využívá se skutečnost, že po zadání adresy rádky a impulzu RAS se do čtecích zesilovačů uloží obsah celé rádky. Pak stačí opakovaně přivádět postupně zvyšované adresy sloupce a doprovázet je jen impulzy CAS – viz obr. 12.13 vlevo. Ještě příznivějšího průběhu cyklů stránkového čtení je dosaženo u pamětí **DRAM-EDO** (angl. *Extended Data Output*) – viz obr. 12.13 vpravo. Trvání dat na výstupu je prodlouženo vždy až do doběžné hrany CAS a systém navazující na paměť tak má data přístupná po delší čas, aniž by se doba cyklu prodlužovala. Výstup se uvede do vysokoimpedančního stavu po návratu signálů RAS a CAS do stavu 1.



Obr. 12.12 Řadič dynamické paměti



Obr. 12.13 Čtení po stránkách a výstup u DRAM typu EDO

Podstatného zrychlení činnosti dynamických pamětí se však dosáhlo až zavedením synchronních dynamických pamětí.

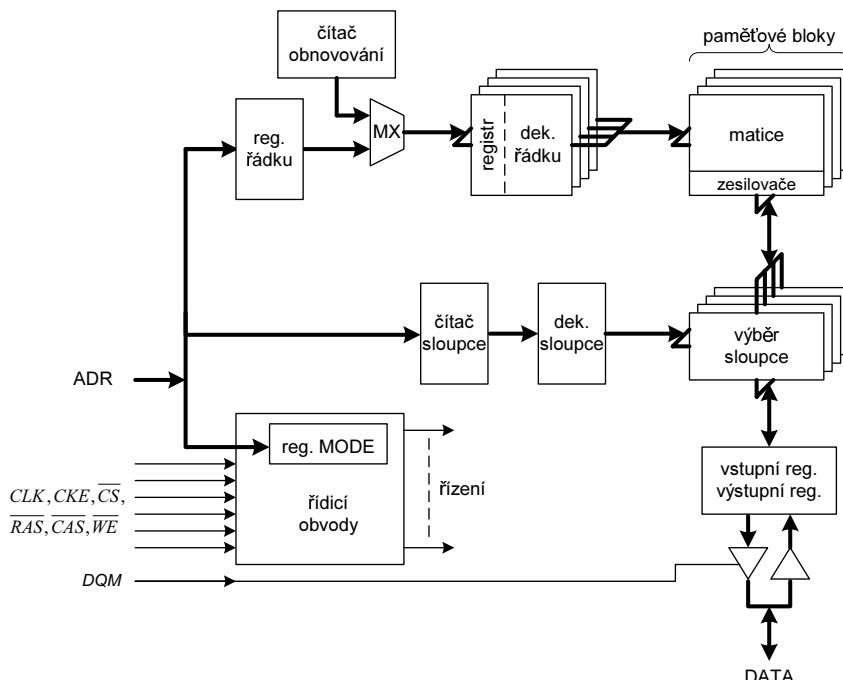
12.7 Synchronní dynamické paměti RAM

Synchronní dynamické paměti RAM, zkratkou SDRAM, využívají stejný princip paměťových buněk, čtecích zesilovačů a dekódování adres, jako běžné DRAM. Jsou však vybaveny dalšími vnitřními obvody, které podstatně urychlují čtení a zápis dat a zjednoduší časování signálů pro paměť.

Podstatnou změnou je zavedení **synchronizačních** (hodinových) **impulzů**. Všechny vstupní signály – adresy, řídicí signály, zapisovaná data – jsou náběžnou hranou hodinového impulzu vzorkovány, tj. zapsány do vnitřních registrů. Vnitřní signály se tak mění v přesně

definovaných okamžicích a naopak na časování vnějších signálů jsou kladený podstatně mírnější nároky, než u jednoduché DRAM. I výstupní data z paměti se mění v závislosti na hodinových impulzech. Celá činnost SDRAM tak může být synchronizována s dalšími obvody v systému (např. s procesorem v počítači).

Zrychlení činnosti SDRAM je dosaženo obdobným principem, jako u výše popsané paměti EDO. Po vložení adresy řádku jsou postupně vybírány sloupce, ale na rozdíl od EDO je u SDRAM sloupcová adresa generována **vnitřním čítačem**, inkrementovaným hodinovými impulzy. Odpadá tak i zadávání sloupcové adresy a data mohou být čtena (zapisována) v rytmu hodinových impulzů, s periodou jen několika nanosekund. Tato rychlosť je využitelná jen u čtení či zápisu celé skupiny dat, neboli v **dávkovém čtení či zápisu** (angl. *burst read, burst write*). Délka dávky a počáteční adresa čítače se do paměti naprogramuje. *Obr. 12.14 ukazuje vnitřní uspořádání SDRAM.*



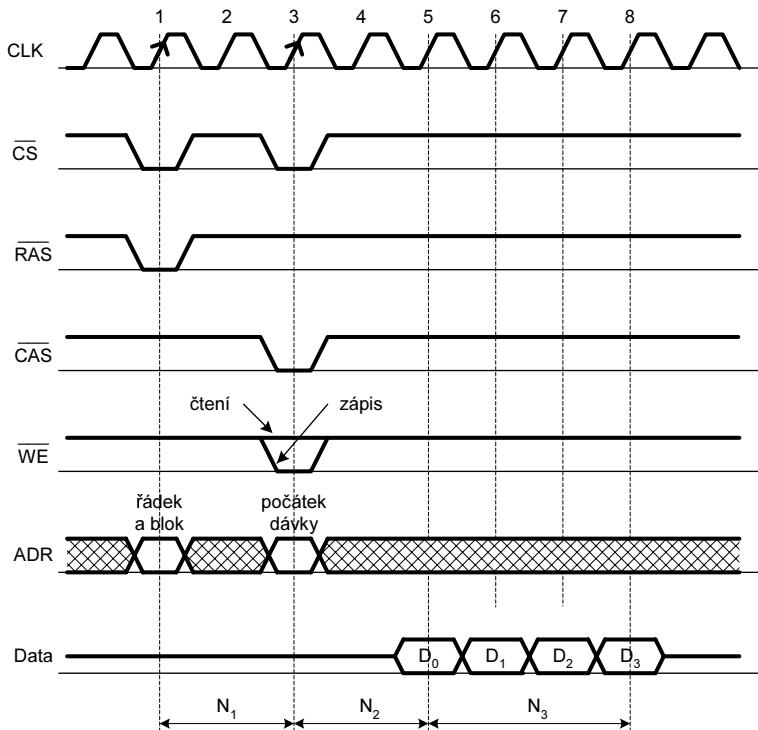
Obr. 12.14 Vnitřní uspořádání paměti SDRAM

Ve srovnání s pamětí DRAM (viz obr. 12.8) je vidět shodu v matici buněk, dekodéru řádků a sloupců, výběrových obvodech a vstupních a výstupních obvodech. Navíc je zde čítač adresy sloupce, čítač pro obnovování s multiplexerem, a registr režimu (MODE). Matice buněk s přilehlými obvody tvoří tzv. **paměťový blok** (angl. *memory bank*). Paměťových bloků je několik – typicky dva nebo čtyři. To přispívá ke zrychlení činnosti paměti, neboť zatím co v jednom bloku probíhá operace čtení nebo zápisu, v jiném může probíhat přípravné nabíjení. Následující čtení řádku může být vyvoláno z tohoto právě připraveného bloku a tím se ušetří čas. Blok se vybírá nejvyšším bitem (bity) řádkové adresy.

Na adresové vstupy paměti se přivádí postupně adresa řádku a adresa pro přednastavení sloupcového čítače. Na rozdíl od paměti DRAM, do které se přivádí adresa sloupce pro sloupcový registr, se u paměti SDRAM adresa sloupce vytváří až ve sloupcovém čítači, takže může být vnitřními impulzy rychle inkrementována. Přečte či zapiše se tak celá dávka dat (angl. *data burst*), počínaje přednastavenou adresou. Lze naprogramovat různé délky dávky – typicky 1, 2, 4 a 8 slov, nebo plný počet daný počtem adres sloupců.

Obnovování informace nevyžaduje u paměti SDRAM žádné vnější obvody. V patřičném okamžiku lze přepnout vnitřní multiplexer a postupně generovat adresy řádků pomocí vnitřního čítače pro obnovování.

Soustava řídicích signálů a hlavně jejich funkce se od DRAM značně liší. Základní cyklus čtení či zápisu ukazuje obr. 12.15.



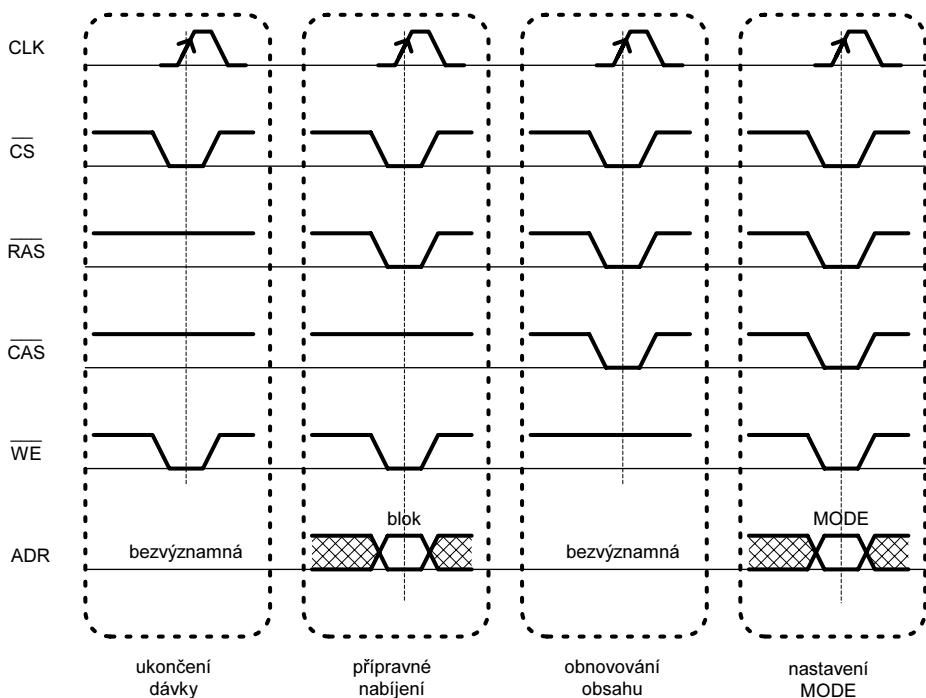
Obr. 12.15 Průběh čtecího a zápisového cyklu paměti SDRAM

Aktivní hranou CLK je hrana náběžná. Stav všech vstupních signálů je podstatný jen v okolí této hrany. Při hraně, označené v obrázku jako 1, se přivádí adresa bloku a řádku, doprovázená impulzem RAS. O dva takty později se přivádí adresa pro přednastavení sloupcového čítače, doprovázená impulzem CAS. Současně stav signálu WE rozhoduje o tom, zda bude následovat čtení či zápis. O jistý počet taktů dále jsou při čtení vydána prvá data z přednastavené adresy sloupce, následovaná takt za taktem daty z postupně zvyšovaných sloupcových adres. Toto zpoždění výdeje prvních dat za CAS se nazývá **doba latence** paměti. Počet vydaných slov, neboli **délka dávky**, je naprogramován v řídicích obvodech paměti.

Při zápisu dat musí být naopak data ve stejných okamžicích dodávána – o automatickém zvyšování sloupcové adresy a o délce dávky platí totéž. Veškeré časové parametry, uváděné u DRAM, jsou v případě SDRAM definovány velmi jednoduše, jen v počtu taktů CLK. Jsou to:

- maximální kmitočet CLK
- odstup \overline{CAS} po (číslo N_1 v obr. 12.15)
- doba latence po \overline{CAS} (číslo N_2 v obr. 12.15)
- délka dávky (číslo N_3 v obr. 12.15) – je v jistých mezích programovatelná
- doba přípravného nabíjení

Vedle čtení a zápisu vyžaduje SDRAM ještě další operace, které vyplývají z její konstrukce. Příkazy pro vyvolání těchto operací ukazuje obr. 12.16. Jsou kódovány kombinacemi hodnot řídicích signálů \overline{CS} , \overline{RAS} , \overline{CAS} , \overline{WE} při náběžné hraně CLK.



Obr. 12.16 Průběh cyklů pro vkládání příkazů do paměti SDRAM

Ukončení dávky (angl. *burst terminate*) umožňuje předčasně ukončit dlouhou dávku dat při čtení či zápisu. Naprogramování délky dávky je totiž omezené – typicky na 1, 2, 4, 8 a celou stránku. Chceme-li např. přenést 30 slov, musíme naprogramovat plnou délku, např. 1024. Po 30. slovu by bylo nutné čekat, až celá dávka doběhne. Proto je vhodné umožnit její přerušení.

optimálně řídit. Před přenosem dávky u jednoho paměťového bloku lze vyvolat přípravné nabíjení u jiného bloku, který tak bude po doběhnutí dávky ihned schopen činnosti. Tato doba se počítá rovněž v počtu impulzů CLK . Blok, u kterého má proběhnout nabíjení, se určí v tomto příkazu nejvyššími bity adresy.

Automatické obnovování obsahu (angl. *self refresh*) využívá vnitřní čítač řádků, takže odpadá vnější řadič paměti. Automaticky se obnovuje obsah celého paměťového bloku.

Nastavení MODE souvisí s naprogramováním řídících obvodů. Do řídícího registru MODE se vloží informace o délce dávky a latenci. Informace se předává po adresových signálech.

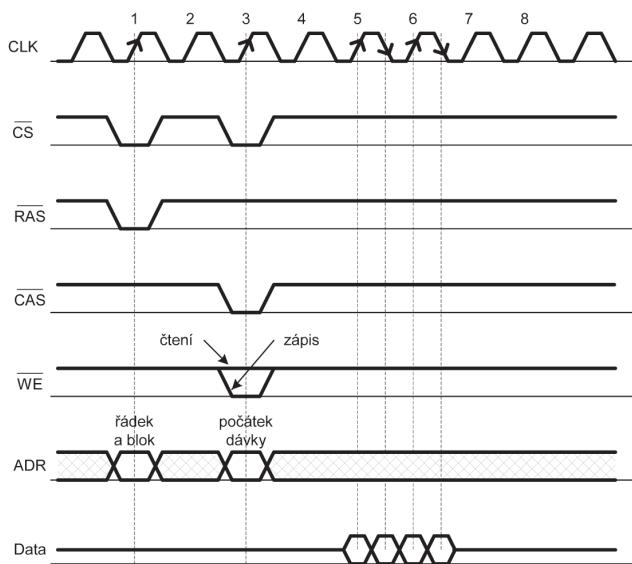
Dynamické parametry SDRAM se specifikují jako trojice čísel v pořadí:

latence – odstup \overline{CAS} za \overline{RAS} – doba přípravného nabíjení

Dalším parametrem je maximální kmitočet CLK . Příkladem specifikace může být paměť: 256 Mb, 32 M \times 8, 133 MHz, 3-2-2.

12.8 Synchronní dynamické paměti DDR

Jednoduchá úprava umožnuje ještě dál zrychlit činnost paměti SDRAM. Spočívá hlavně v úpravě čítače sloupců. Během čtení či zápisu dávky dat je inkrementován jak náběžnou, tak i dóběžnou hranou CLK . Tím se dosáhne zdvojnásobení přenosové rychlosti při přenosu dávky, takže např. při kmitočtu $f_{CLK} = 200$ MHz jsou data vydávána v intervalech 2,5 ns. Průběh čtecího či zápisového cyklu ukazuje obr. 12.17. Zkratka DDR je odvozena z označení **dvojnásobná přenosová rychlosť** (angl. *Double Data Rate*).



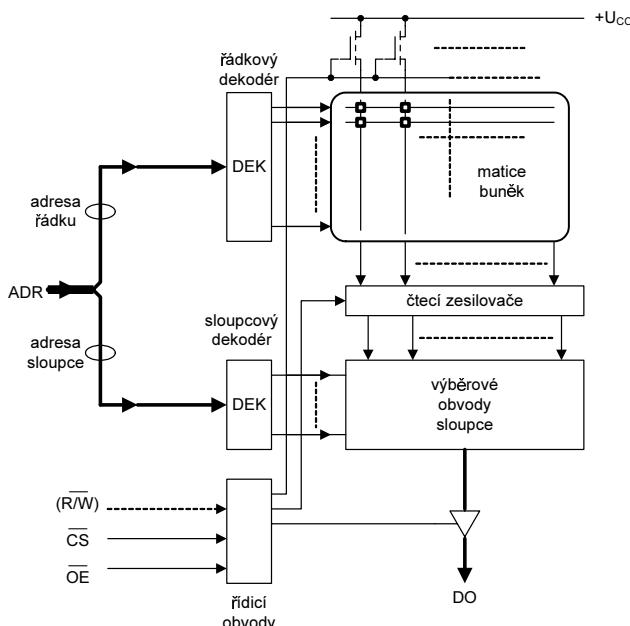
Obr. 12.17 Průběh čtecího a zápisového cyklu paměti DDR SDRAM

12.9 Permanentní paměti

Podstatnou vlastností každé permanentní paměti je to, že je energeticky nezávislá – uložená data se neztrácejí po vypnutí napájení. Existuje několik typů permanentních pamětí:

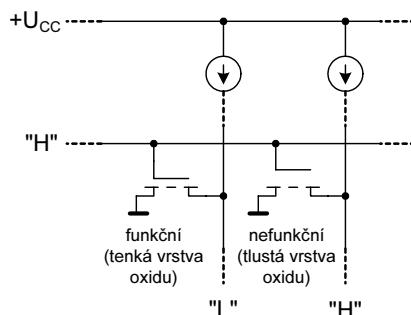
- Paměť **ROM** (angl. *Read-Only Memory*) – obsah je dán při výrobě a nelze jej již změnit.
- Paměť programovatelná, **PROM** (angl. *Programmable ROM*) – obsah může uživatel zapsat (naprogramovat), ale naprogramování je nevratné.
- Paměť **EPROM** (angl. *Erasable PROM*) – paměť programovatelná a vymazatelná. Je programována elektricky. Před programováním musí být vymazána ultrafialovým zářením.
- Paměť **EEPROM** (angl. *Electrically Erasable PROM*) – paměť programovatelná, programována elektricky na libovolné adrese bez předchozího vymazání. Spíše než o programování a mazání zde lze mluvit o **zápisu dat** na adresu, obdobně jako u paměti RAM. Zápis je však mnohem pomalejší než čtení.
- paměť **FLASH** – paměť programovatelná a vymazatelná. Je programována elektricky na libovolné adrese, ale elektricky mazána jako celek (nebo po velkých částech – sektořech či blocích). Před programováním musí být vymazána.

Vnitřní uspořádání permanentní paměti menšího rozsahu ukazuje obr. 12.18. Ve srovnání s obr. 12.3, který znázorňuje hlavní bloky paměti s adresovým výběrem, jsou vyznačeny cesty dat jen pro čtení. To odpovídá funkci permanentní paměti. Vnitřní signály, potřebné pro naprogramování obsahu, se však běžně neuvádějí. Pro vložení dat se vždy využívají vývody DO, pro jejich zápis do matice buněk slouží patřičné kombinace řídicích signálů.



Obr. 12.18 Vnitřní uspořádání permanentní paměti

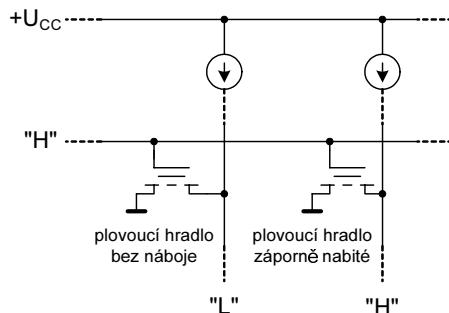
Provedení buněk paměťové matice odpovídá typu paměti. U paměti ROM má buňka nejjednodušší podobu jako tranzistor MOS, zapojený mezi sloupec a zem. V každé buňce existuje tranzistor, ale jen některé z nich jsou funkční. Naprogramováním se ovlivní funkčnost tranzistoru tak, že u funkčních tranzistorů je mezi řídicí elektrodou (hradlem) a polovodičem jen tenká vrstva oxidu křemíku, zatímco u tranzistorů nefunkčních je tato vrstva tlustá. Po vybuzení řádkového vodiče dojde u funkčních tranzistorů k vytvoření indukovaného kanálu a tím ke zkratování sloupce na zem (úroveň L), zatímco u tranzistorů s tlustou vrstvou oxidu je vzdálenost hradla od polovodiče tak velká, že nedojde k inverzi vrstvy pod hradlem a k vytvoření vodivého kanálu. Napětí na sloupci tedy zůstane vysoké (na úrovni H). Princip je ilustrován na obr. 12.19. Individuální naprogramování spočívá v proleptání tlusté vrstvy oxidu na stanovených místech před dalšími – již standardními – technologickými operacemi. Takto zapsaná data jsou samozřejmě neměnná. Maska pro litografii v této fázi výrobního procesu je individuální podle zadání zákazníka. Paměť tohoto typu se nazývá **maskou programovaná**.



Obr. 12.19 Buňky paměti ROM maskou programované

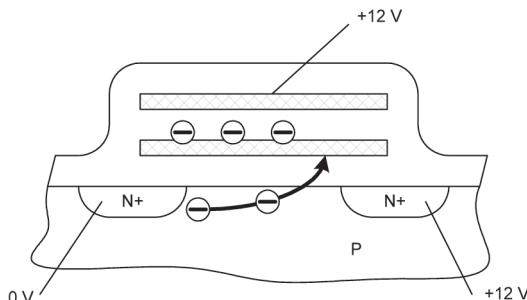
Programovatelné **paměti PROM** byly v minulosti realizovány pomocí buněk s přepalovacími spojkami. Tato technologie vyžadovala velké proudy pro přepálení spojek a byla realizovatelná jen v bipolární technologii. Zanikla po nástupu technologie CMOS.

V technologii CMOS se využívají buňky, u kterých se mezi řídicím hradlem a polovodičem vytvoří další vodivá vrstva, totálně odizolovaná oxidem křemíku – tzv. **plovoucí hradlo** (angl. *floating gate*). Na plovoucí hradlo lze dopravit elektrony a nabít je tak záporně. Po vybuzení řádkového vodiče a stavu bez náboje se vytvoří indukovaný kanál, sloupec je zkratován na zem a je na něm úroveň L. Pokud je ale plovoucí hradlo záporně nabité, je vliv kladného napětí na řádkovém vodiči nedostatečný k vytvoření kanálu a na sloupci zůstane úroveň H. Jinými slovy, náboj na plovoucím hradle ovlivňuje prahové napětí MOS tranzistoru. Princip je ilustrován obr. 12.20.



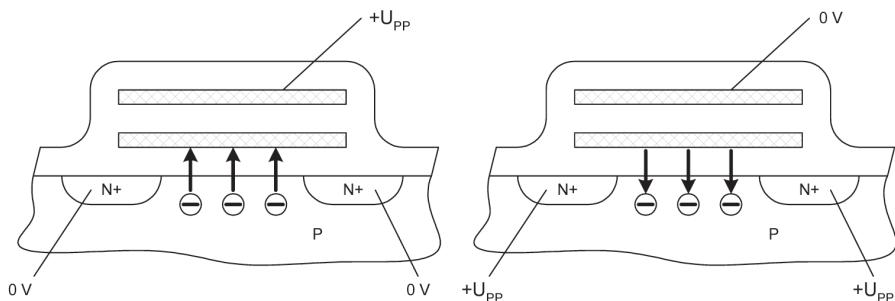
Obr. 12.20 Buňky paměti programované nábojem

Způsob přivedení náboje na plovoucí hradlo a jeho odvedení je v podstatě dvojí. Z něj pak vyplývá i způsob naprogramování (případně mazání) buněk v matici. Za prvé je to **lavinová injekce** elektronů. Při ní jsou elektrony ve vodivém kanálu urychleny elektrickým polem mezi konci kanálu a je jim dodána energie dostatečná k vytvoření omezeného lavinového průrazu. Současně je jejich dráha vychýlena vlivem kladného napětí na řídícím hradle. Urychlené elektrony („hot electrons“) nabudou dostatečné energie k průletu tenkou vrstvou oxidu pod plovoucím hradlem. Tento mechanismus injekce „horkých“ elektronů umožňuje nabít hradlo záporně během krátké doby, řádově mikrosekund. Technologie umožňující tuto funkci má název FAMOS (z angl. *Floating-Gate Avalanche-Injection MOS*). Obr. 12.21 ukazuje řez tranzistorem FAMOS. Tlustá šipka ukazuje dráhu elektronů.



Obr. 12.21 Řez tranzistorem FAMOS

Druhý způsob využívá kvantový jev – přemístění elektronu přes vrstvu izolantu při velmi vysoké intenzitě elektrického pole, známý jako Fowler-Nordheimův mechanismus **tunelování elektronů**. Aby nebylo nutné používat vysoké napětí, musí být vrstva izolantu velmi tenká. Při vrstvě oxidu křemičku o tloušťce kolem 10 nm postačuje napětí do 20 V. Opět se využívají dvě hradla – řídící a plovoucí (totálně odizolované) – viz obr. 12.22. Programovací napětí je v něm označeno jako $+U_{pp}$. Obrázek vlevo ukazuje uspořádání pro nabít plovoucího hradla záporným napětím, obrázek vpravo pak vybití plovoucího hradla. Manipulací s napětím tedy lze měnit stav paměťové buňky. Celý mechanismus však je relativně pomalý – změna stavu buňky trvá řádově milisekundy.



Obr. 12.22 Uspořádání pro nabítí a vybití plovoucího hradla

Oba výše uvedené mechanismy jsou základem pamětí typu EPROM, EEPROM a FLASH.

Paměť EPROM je nejstarší a využívá buňku FAMOS. Paměťová matici má jednoduchou strukturu. Naprogramování injekcí horkých elektronů zablokuje tranzistor v buňce. V cestě signálu od sloupcového vodiče na výstup dat je zpravidla jedna negace, takže naprogramovaná buňka se projeví jako stav „0“ na výstupu. K programování slouží přístroj – programátor EPROM – který dodává potřebná napětí a přesně časované signály. Při programování je nutné na jeden z vývodů pouzdra přivést zvýšené programovací napětí U_{PP} (zpravidla kolem 12 V). Programuje se vždy celé slovo najednou, data k programování se přivádí na vývody DO. Programátor je řízen počítačem, který také dodává celý soubor s celým obsahem paměti. Existuje mnoho výrobců a mnoho typů EPROM, a proto programátor musí být schopen přizpůsobit velikost napětí a časování signálů podle konkrétní paměti. Typ se musí programátoru sdělit prostřednictvím počítače (výběrem z menu), nebo u dokonalejších pamětí lze informaci o typu přímo vyčíst z programované paměti. Trvanlivost dat ve správně naprogramované EPROM záleží hlavně na teplotě. Je pro běžné účely dostatečná, rádově desetiletí.

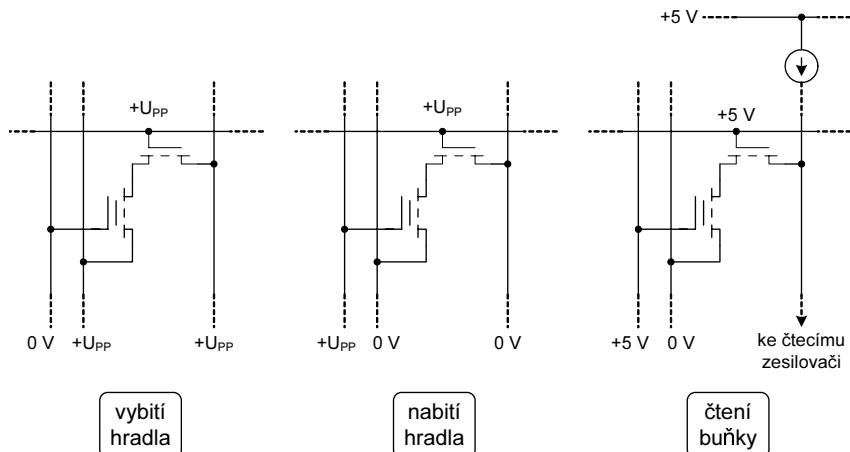
K mazání EPROM slouží mazací přístroj. Vymazání buňky se provádí UV zářením o vlnové délce 254 nm, generovaným nízkotlakou rtuťovou výbojkou po dobu asi 15 minut. Záření dodá elektronům energii k opětnému překonání bariéry oxidu křemíku směrem do polovodiče. Pouzdro integrované paměti samozřejmě musí dovolit průchod UV záření a je proto opatřeno víckem z křemenného skla. Vymazaná paměť dává na všech adresách a ve všech bitech dat stav „1“.

Programování i mazání lze opakovat, u pamětí EPROM je však počet opakování malý – rádově desítky. Podstatnou nevýhodou je nutnost vyjmutí obvodu z desky s plošným spojem při mazání i programování. To vyžaduje velmi kvalitní (a drahou) objímkou pro integrovaný obvod, tak aby dlouhodobá spolehlivost nebyla ohrožena korozí kontaktů. Celkově lze paměť EPROM charakterizovat jako nevhodnou pro aplikace vyžadující časté změny obsahu a v dnešní době jako celkově zastaralou.

Paměti EPROM se vyrábějí i ve verzi **OTP** (angl. *One-Time Programmable* – jednou programovatelná). Jedná se o paměť EPROM s pouzdrem bez okénka, tedy s nemožností vymazání obsahu. To je řešení relativně levné, paměť nevyžaduje objímkou (nebude se přeprogramovávat) a může být v plošném spoji zapájena.

Paměť EEPROM využívá tunelování elektronů v obou směrech. Buňka matici je složena ze dvou tranzistorů – jeden výběrový a jeden paměťový. Paměťový tranzistor má plovoucí hradlo a řídící hradlo. To se uplatňuje jen při zápisu dat a běžně není ve zjednodušených

schématech kresleno. Princip ukazuje obr. 12.23 s vepsanými hodnotami napětí pro vybití hradla, nabítí hradla a čtení buňky. Programovací napětí je v něm označeno jako $+U_{PP}$.



Obr. 12.23 Pracovní režimy buňky EEPROM

Buňka EEPROM je poměrně složitá, umožňuje však záporné nabítí či vybití plovoucího hradla na jakékoli adresě. Spiše než „programování“ či „vymazání“ má smysl zde hovořit o **zápisu** hodnoty „0“ či „1“. Tím připomíná paměť RAM, zápis je však mnohem pomalejší. Počet zápisů na jedné adrese je omezený. Různí výrobci uvádějí počty desítek až stovek tisíc.

Paměť FLASH je založena na buňce, využívající kombinace principu lavinové injekce elektronů pro záporné nabítí hradla (naprogramování buňky), a principu tunelování elektronů pro vybití hradla (vymazání buňky). Buňky jsou jednotranzistorové, jednodušší než u EEPROM. Jednoduchá konstrukce paměti však neumožňuje selektivní vymazání na jednotlivých adresách, nýbrž jen **mazání velkých bloků** buněk (sektorů). Sektor je vybírána nejvyššími byty adresy. Programování je rychlé (odtud název „flash“), srovnatelné s programováním paměti EPROM. Mazání je pomalé, řádově do jedné sekundy. Maže se však najednou celá velká oblast. Postup při změně obsahu je takový, že nejprve se vymaže patřičný sektor a pak se celý znova naprogramuje novým obsahem. Paměť FLASH se svojí jednoduchou buňkou umožňuje velmi vysokou integraci. Dělení paměťové matice na sektory je nutné pro úsporu času při malých změnách, omezených jen na malou oblast buněk. Pak totiž postačí manipulace jen s jedním sektorem. Počet zápisů na jedné adrese je omezený a garantované počty jsou přibližně stejné jako u EEPROM. Je nutné si uvědomit, že se jedná o počty zápisů **na jedné adrese**, nikoliv u celé paměti – počet zápisů pro celou paměť je mnohem větší. Při vhodném uspořádání obsahu lze zatížení buněk opakoványmi zápisy rozložit přibližně rovnoměrně. To je zvláště důležité u takových aplikací, jako je nahrazena pevného disku („flash disk“) v počítačích.

Programování a mazání EEPROM a FLASH paměti může probíhat v programátoru. Tak se běžně postupuje při vývoji a ověřování systému, kdy paměť je třeba přeprogramovat mnohokrát. Po dokonalém ověření funkce se pak naprogramuje nová paměť a do systému se může zapájet. Paměti EEPROM a FLASH však umožňují i programování a mazání přímo v systému, takže paměť může být zapájena. Pro manipulaci s pamětí musí některé signály

být přepínány dodatečnými obvody (multiplexery), nebo musí být pro programování přímo rezervovány. Paměti EEPROM a FLASH jsou často součástí obvodů velmi velké integrace, jako jsou jednočipové mikropočítače. Pak jsou samozřejmě programovány v systému.

Programování moderních typů EEPROM a FLASH je velmi usnadněno tím, že přímo na čipu je vedle vlastní paměti integrován i sekvenční obvod – **řadič programování** – který řídí potřebné vnitřní operace. Rovněž zvýšené programovací napětí je generováno na čipu integrovanou **nábojovou pumpou**, takže zvenku je paměť napájena jen běžným provozním napětím U_{CC} . Čtení z pamětí EPROM, EEPROM a FLASH je velmi jednoduché a podobné čtení ze statické RAM. Průběh čtecího cyklu odpovídá obr. 12.7. Zápis, mazání a další operace se však liší typ od typu podle výrobce.

Průběh zápisu je však mnohem pomalejší, než u paměti RAM. Významného zrychlení bylo dosaženo u pamětí, vybavených vnitřním posuvným registrem na obsah stránky. Data se neprogramují přímo do jednotlivých buněk, ale postupně (a rychle) se zavedou do registru. Po jeho zaplnění se jeho jednotlivé vývody propojí se sloupovými vodiči a data se tak přesunou do buněk najednou.

Vybavovací doba pamětí EPROM, EEPROM a FLASH je poměrně dlouhá, ve srovnání s rychlou SRAM až desetinásobná. Tuto nevýhodu se podařilo obejít u **synchronních** EEPROM a FLASH podobným trikem jako u SDRAM – paměť je vybavena vnitřním čítačem sloupcové adresy a data jsou vydávána v rytmu hodinových impulzů.

Vývoj v permanentních pamětech se soustředil na EEPROM a hlavně na FLASH. Objevilo se nové uspořádání matice buněk, tzv. **NAND FLASH** a další. Paměť s klasickým uspořádáním buněk je nazývána **NOR FLASH** pro jistou podobnost s logickým členem, u kterého vzniká paralelní zapojení tranzistorů na sloupcový vodič logická funkce NOR. Toto uspořádání však vyžaduje připojení každé buňky jednak na sloupcový vodič, jednak na zem. Při sériovém spojení buněk u paměti NAND FLASH je propojení podstatně jednodušší, což vede na vyšší hustotu integrace. Paměť NAND FLASH je však pomalejší než NOR FLASH.

Následující výpis kódu ukazuje realizaci paměti ROM s organizací 8×8 bitů v jazyce VHDL.

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

entity rom_async is
    port (
        adr : in std_logic_vector (3 downto 0);
        data : out std_logic_vector (7 downto 0)
    );
end rom_async;

architecture a_rom_async of rom_async is
    type mem is array (0 to 15) of std_logic_vector (7 downto 0);
    signal rom_block : mem := (X"F1", X"12", X"19", X"72", X"E2", X"D9", X"19",
                               X"76", X"26", X"C7", X"2B", X"A3", X"11", X"F1", X"2F", X"D6");
begin
    data <= rom_block(to_integer(unsigned(adr)));
end a_rom_async;
```

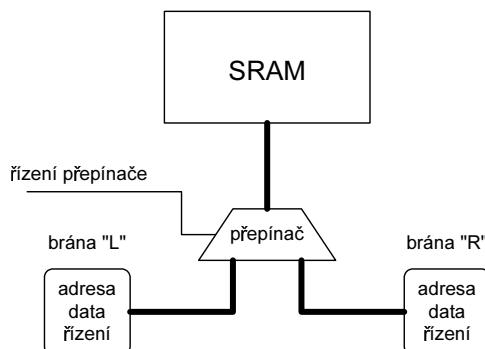
Výpis 12.2 Asynchronní paměť ROM

12.10 Dvojbránová paměť

Dvojbránová paměť DPM (angl. *Dual-Port Memory*) je statická RAM, která umožňuje přístup k datům ze dvou bran. Každá brána zahrnuje adresu, data a řídicí signály. Dvojbránová paměť je využívána k předávání dat mezi dvěma částmi systému – „uživateli společné paměti“ – které nepracují v přísném synchronismu. Velmi často je např. vložena mezi komunikační procesor a řídicí počítač. V paměti jsou na stanovených místech uloženy zprávy k odeslání a zprávy přijaté, určené ke zpracování. Komunikační procesor pracuje svým rytmem, odesílá a přijímá zprávy. Řídicí počítač pracuje též svým rytmem, zpracuje přijaté zprávy a ukládá do paměti zprávy určené vysílání. Při přímém propojení, bez DPM jako spojovacího článku, by docházelo k časovým ztrátám, k čekání jedné části systému na druhou.

Někdy, i když méně často, je třeba zavést spojovací článek pro více než dva uživatele. Existují proto i paměti se třemi nebo čtyřmi branami. Jejich konstrukce je jen rozšířením principů dvojbránových pamětí, uvedených v dalším textu. Paměti s více než dvěma branami proto nebudou již zvláště popisovány.

Existuje několik způsobů, jak ze dvou stran dosáhnout na buňky ve společné paměťové matici. Nejjednodušší je vzít za základ standardní paměť SRAM a doplnit ji přepínačem, který přepíná signály ze dvou bran na vstupní a výstupní vývody SRAM – viz obr. 12.24. Je zřejmé, že není možný **současný** přístup z obou bran – brány se musí vhodným způsobem střídat.



Obr. 12.24 Vytvoření dvojbránové paměti z SRAM

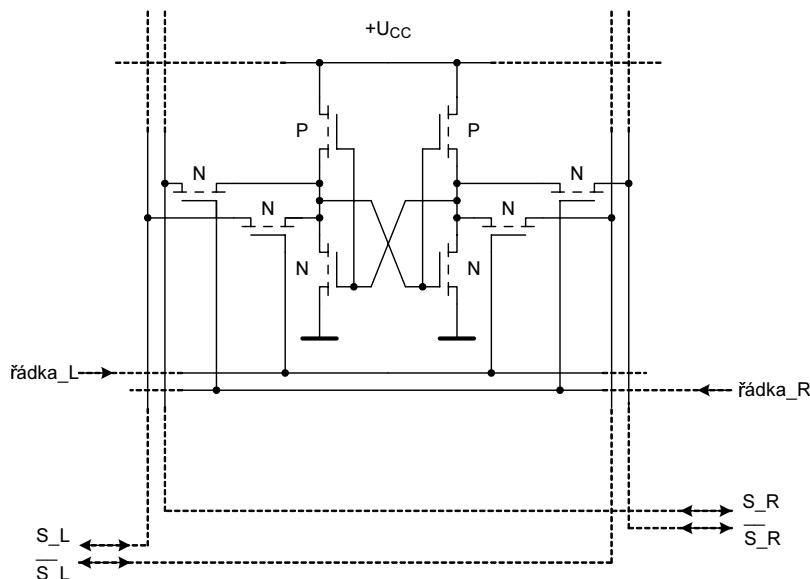
Někdy je možné zařídit střídání přístupu v pravidelném rytmu. Každý uživatel má svoji bránu k dispozici po přesně určenou dobu časového okna. To je možné bez časových ztrát, pokud každý uživatel pracuje v **pravidelném rytmu** „práce s pamětí“ – „zpracování dat“. Funkce obou uživatelů a přepínače jsou vzájemně vázány a centrálně řízeny. Pak se mohou tyto dvě fáze činnosti vzájemně překrývat takto:

uživatel L (levý): ... práce s pamětí → zpracování dat → práce s pamětí →
 uživatel R (pravý): ... zpracování dat → práce s pamětí → zpracování dat →
 přepínáč: ... L R T

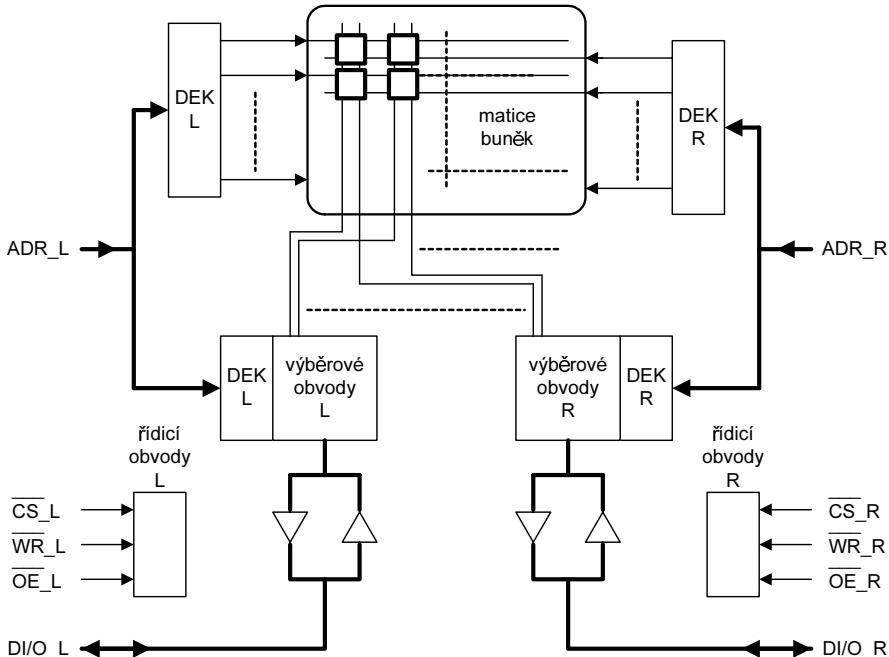
Většinou takováto pravidelnost není možná a požadavky na přístup k paměti jsou náhodně rozložené v čase, občas i současné z obou stran. Pro náročnější aplikace byl vyvinut nový typ paměťové buňky, přístupné ze dvou stran – viz obr. 12.25.

Princip je obdobný jako u buňky běžné SRAM podle obr. 12.6. Buňka je však napojena na dvě dvojice sloupců – dvojice sloupců „levých“ (S_L, \bar{S}_L) a dvojice sloupců „pravých“ (S_R, \bar{S}_R). Připojení na „levé“ sloupce je řízeno „levým“ řádkovým vodičem, připojení na „pravé“ sloupce „pravým“ řádkovým vodičem. S buňkou tak lze pracovat ze dvou stran. Jen jedna operace je zakázaná, a to **současný zápis** z obou stran. Pokud náhodou nejsou takto zapisovaná data z obou stran shodná, nebude výsledek definován. Uspořádání DPM s využitím matice buněk se dvěma přístupy ukazuje obr. 12.26.

Obrázek ukazuje shodu s obrázkem běžné SRAM (obr. 12.3), všechny obvody kromě vlastní matice jsou však zdvojeny a jsou navzájem nezávislé. Vstupní a výstupní signály jsou uspořádány do dvou skupin, tj. bran. Nezávislost obvodů dovoluje současné operace čtení či zápisu na různých adresách. Na shodných adresách je však třeba vyloučit současné operace zápisu – jiné kombinace operací nejsou na závadu. Obvody podle obr. 12.26. jsou proto ještě doplněny o **arbitrážní obvod** – „arbiter“.



Obr. 12.25 Statická buňka přístupná ze dvou stran

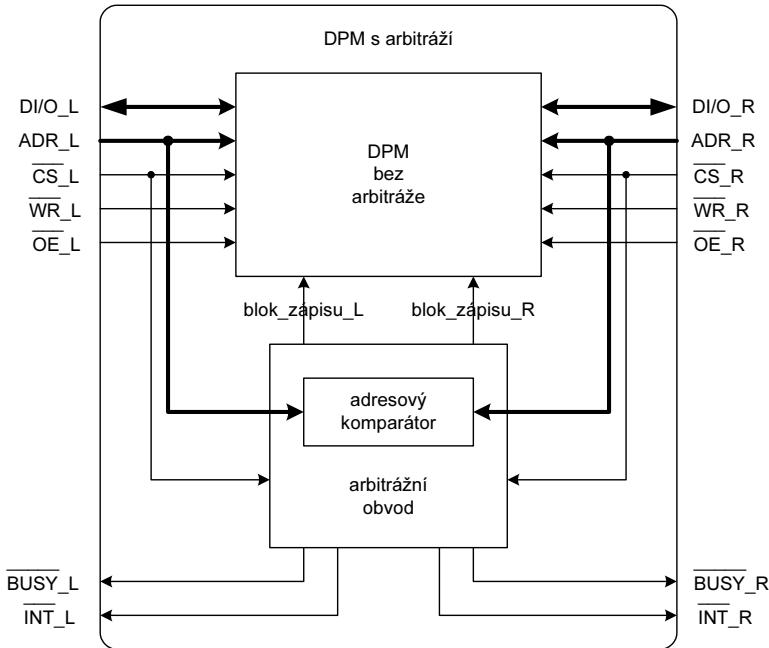


Obr. 12.26 Přístup k matici buněk ze dvou bran

Úkolem arbitrážních obvodů je řešení střetu dvou (obecně libovolného počtu) požadavků na přístup do jednoho místa. Nejčastěji se u DPM jedná o arbitráž symetrickou, s rovnocennými prioritami požadavků. Funkci arbitráže lze popsat dvěma pravidly:

1. operace je povolena tomu uživateli, který generoval požadavek jako první
2. při současném výskytu více než jednoho požadavku je operace povolena uživateli náhodně zvolenému

Na obr. 12.27 je znázorněno začlenění arbitrážního obvodu do paměti DPM. Požadavek na přístup k paměti je odvozen od výběrového signálu \overline{CS} . Jediná operace, která podléhá arbitráži, je současný zápis z obou stran na stejných adresách. Arbiter proto obsahuje adresový komparátor. Pohotovost brány k operaci je označena uživateli signálem $BUSY$. Stavem „1“ je signalizována pohotovost k operaci, stavem „0“ dočasná nemožnost provedení operace. Uživatel brány paměti tedy musí tento signál testovat před provedením operace a při jeho stavu „0“ počkat.



Obr. 12.27 Dvojbránová paměť s arbitráží

Při předávání dat mezi uživateli přes DPM je velmi často nutné informovat druhého uživatele o **nové události**, např. o zaplnění oblasti paměti novými daty. K okamžitému informování slouží výstupy *INT*. Jejich smyslem je přerušení počítače, který je uživatelem brány. Tato funkce je zařízena tak, že při zápisu z jedné brány do vyhrazené oblasti adres (několik nejvyšších adres) je generován signál přerušení ve druhé bráně. Ten trvá do té doby, než se z brány, ve které bylo generováno přerušení, přečtu data z vyhrazené oblasti adres.

Trend ke zvyšování rychlosti se nevynul ani dvojbránovým pamětem. Vzniklo tak několik typů **synchronních DPM**. Princip synchronního provozu s čítačem sloupců inkrementovaným hodinovými pulzy je obdobný jako u synchronních dynamických pamětí.

Následující výpis VHDL kódu ukazuje realizaci synchronní dvojbránové paměti RAM s jednou bránou pro zápis a druhou bránou pro čtení s organizací 512×8 bitů v jazyce VHDL. Zápis a čtení z paměti je realizován pomocí dvou nezávislých procesů. Příklad umožňuje navzájem nezávislé operace zápisu a čtení (mohou probíhat i současně).

```

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

entity ram_dual is
  port (
    clock_wr, clock_rd : in std_logic;
    data_in : in std_logic_vector (7 downto 0);
    write_address, read_address : in std_logic_vector (8 downto 0);
    we : in std_logic;
    data_out : out std_logic_vector (7 downto 0)
  );
end entity;

```

```

    );
end ram_dual;

architecture a_ram_dual of ram_dual is
begin
    -- zapis do RAM
    process (clock_wr)
    begin
        -- zapis do RAM na vzestupnou hranu clock_wr
        if clock_wr'event and clock_wr = '1' then
            if we = '1' then
                ram_block(to_integer(unsigned(write_address))) <= data_in;
            end if;
        end if;
    end process;

    -- cteni z RAM
    process (clock_rd)
    begin
        -- zapis ctene adresy do registru na vzestupnou hranu clock_rd
        if clock_rd'event and clock_rd = '1' then
            read_address_reg <= read_address;
        end if;
    end process;

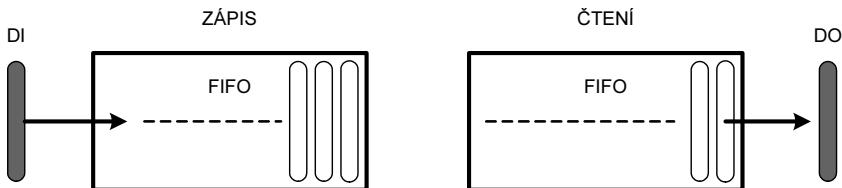
    -- data z ram jsou ctena z adresy ulozene v registru
    data_out <= ram_block(to_integer(unsigned(read_address_reg)));
end a_ram_dual;

```

Výpis 12.3 Dvojbránová synchronní paměť RAM s jednou bránou pro zápis a druhou pro čtení

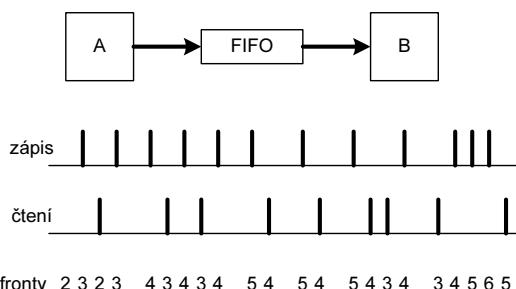
12.11 Paměť fronty

Paměť fronty FIFO (angl. *First In – First Out*) patří mezi paměti **bezadresové**. Adresa není zapotřebí proto, že pořadí zápisu a čtení dat je pevně dán. Při zápisu dat se jednotlivá slova řadí do fronty a při čtení se vybírají v tom pořadí, v jakém byla zapsána. Operace zápisu a čtení jsou přitom vzájemně nezávislé, takže paměť FIFO může být zaplněna do různé míry – od zcela vyprázdněné až po zcela zaplněnou. *Obr. 12.28* názorně ilustruje činnost při zápisu a čtení. Posunování dat, znázorněné na obrázku, je jen zdánlivé. U moderních konstrukcí paměti FIFO jsou data naopak na stabilních pozicích a posunují se ukazatele na ně – viz další text.



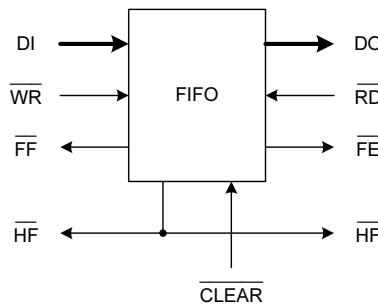
Obr. 12.28 Činnost paměti FIFO

Paměť FIFO se používá jako vyrovnávací člen mezi dvěma částmi systému. Předpokládejme, že jedna část (A) předává data druhé části (B). Při přímém propojení musí A i B pracovat ve shodném rytmu – jinak vždy jedna část musí čekat na druhou, aby nedocházelo ke ztrátě dat. Při vložení vyrovnávací paměti se mohou rychlosti zpracování dat u obou částí **krátkodobě** rozcházet, aniž by docházelo k časovým ztrátám. Je-li momentálně A rychlejší než B, bude se FIFO postupně zaplňovat. Po nějaké době se situace změní a B je rychlejší, takže se FIFO postupně vyprazdňuje – viz příklad na obr. 12.29.



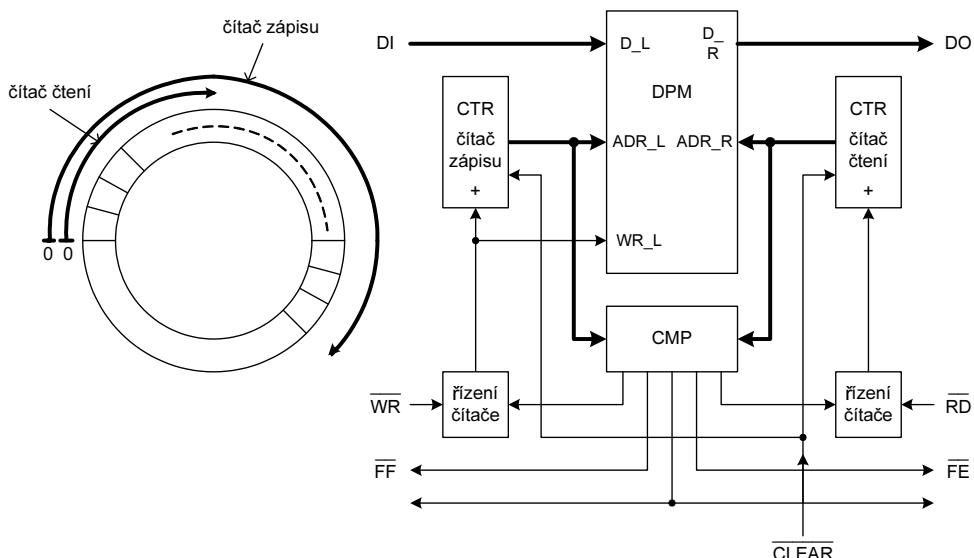
Obr. 12.29 FIFO jako vyrovnávací paměť

V praxi není vyloučeno vyprázdnění či přeplnění FIFO. Aby při tom nedocházelo ke ztrátám dat či čtení falešných dat, je třeba oba uživatele paměti informovat o pohotovosti paměti. FIFO proto generuje signály FF , EF , HF – Full FIFO (plné FIFO), Empty FIFO (prázdné FIFO), Half Full (zpola plné). Signál FF je nutný pro ochranu před přeplněním FIFO a ztrátou dat. Před zápisem je testován. Je-li ve stavu „1“, je možné zápis provést, jinak je třeba zápis pozdržet a dále signál testovat. Obdobně se zachází se signálem EF při čtení dat. Signál HF , který není generován u všech typů FIFO, je užitečný jako předběžné varování před přeplněním při zápisu nebo úplným vyprázdněním při čtení. Soustava vstupních a výstupních signálů je vyznačena na obr. 12.30.



Obr. 12.30 Vstupní a výstupní signály paměti FIFO

Konstrukci paměti FIFO ukazuje obr. 12.31. Je založena na zjednodušené dvojbránové paměti a dvou **ukazatelích na adresy**, realizovaných čítači. Zjednodušení DPM spočívá v omezení funkce u obou bran – jedna dovoluje jen zápis, druhá jen čtení. Po každém zápisu dat se posune čítač zápisu a po každém čtení se posune čítač čtení. Vzhledem k funkci čítačů, kdy po nejvyšší adrese následuje nula, lze adresy v paměti znázornit jakoby navinuté na kružnici.



Obr. 12.31 Konstrukce paměti FIFO se dvěma ukazateli adres

Čítač zápisu obsahuje většinou vyšší číslo než čítač čtení. To odpovídá částečnému zaplnění FIFO. Jakmile by čítač čtení „doběhl“ čítač zápisu, odpovídalo by to úplnému vyprázdnění FIFO. Naopak čítač zápisu nesmí „předběhnout“ čítač čtení, což by odpovídalo úplnému zaplnění FIFO. Obě adresy jsou proto srovnávány v komparátoru, který v případě zaplnění nebo vyprázdnění FIFO zablokuje patřičný čítač. Komparátor též generuje signály FF, EF, HF. Signálem CLEAR lze vynulovat oba čítače, což odpovídá vyprázdnění FIFO.

Paměť FIFO podle výše uvedené konstrukce je **asynchronní**. Dovoluje navzájem nezávislé operace zápisu a čtení – mohou probíhat i současně. Zpoždění, které zavádí paměť FIFO vložená mezi dvě části systému, samozřejmě zpožďuje celý systém. Výrobci proto používají ty nejrychlejší technologie. Další zrychlení se dosahuje u **synchronních** pamětí FIFO. Kromě signálů z obr. 12.30 jsou do nich ještě zavedeny hodinové impulzy, nezávisle pro čtení a pro zápis. Princip je obdobný jako u paměti SDRAM.

Následující výpis kódu ukazuje realizaci synchronní paměti FIFO s organizací 256×8 bitů v jazyce VHDL. Paměť FIFO je realizována pomocí dvou nezávislých procesů implementujících čítače zápisové a čtecí adresy. Příklad umožňuje navzájem nezávislé operace zápisu a čtení (mohou probíhat i současně). Z důvodu rozsahu výpisu kódu nerealizuje příklad signály EF, HF, FF indikující stav zaplnění FIFO paměti.

```

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

entity fifo is
  port (
    reset : in std_logic;
    wr_en : in std_logic;
    wr_clk : in std_logic;
    rd_en : in std_logic;
    rd_clk : in std_logic;
    din : in std_logic_vector (7 downto 0);
    words : out std_logic_vector (7 downto 0);
    dout : out std_logic_vector (7 downto 0)
  );
end fifo;

architecture model of fifo is
  type mem is array(0 to 255) of std_logic_vector (7 downto 0);
  signal fifo_mem : mem;
  signal fifo_wr_ptr, fifo_rd_ptr : unsigned (7 downto 0);
begin
  -- zapis do FIFO
  process (wr_clk, reset)
  begin
    -- asynchronni reset
    if reset = '1' then
      fifo_wr_ptr <= (others => '0');
    -- zapis do fifo na vzestupnou hranu wr_clk
    elsif wr_clk'event and wr_clk = '1' then
      -- zapis je proveden pouze kdyz je povolen
      if wr_en = '1' then
        fifo_mem(to_integer(fifo_wr_ptr)) <= din;
        fifo_wr_ptr <= fifo_wr_ptr + 1;
      end if;
    end if;
  end process;
  -- pocet slov ve FIFO
  words <= std_logic_vector (fifo_wr_ptr - fifo_rd_ptr);

  -- cteni z FIFO

```

```

process (rd_clk, reset)
begin
    -- asynchronni reset
    if reset = '1' then
        fifo_rd_ptr <= (others => '0');
    -- cteni z fifo na vzestupnou hranu rd_clk
    elsif rd_clk'event and rd_clk = '1' then
        -- cteni je provedeno pouze kdyz je povoleno
        if rd_en = '1' then
            dout(7 downto 0) <= fifo_mem(to_integer(fifo_rd_ptr));
            fifo_rd_ptr <= fifo_rd_ptr + 1;
        end if;
    end if;
    end process;
end model;

```

*Výpis 12.4 Synchronní FIFO s organizací 256×8
 (signály EF, HF, FF nejsou implementovány)*

426410



PROGRAMOVATELNÉ LOGICKÉ OBVODY

V této kapitole jsou popsány tři typy obecných architektur programovatelných logických obvodů (angl. *Programmable Logic Device – PLD*). První architekturou jsou obvody typu **SPLD** (angl. *Simple Programmable Logic Device*), druhou pak obvody typu **CPLD** (angl. *Complex Programmable Logic Device*) a poslední obvody typu **FPGA** (angl. *Field Programmable Gate Array*). Výrobou programovatelných logických obvodů se dnes zabývá mnoho světových polovodičových firem. Jmenujme zde alespoň ty největší výrobce, mezi něž patří firmy Xilinx, Altera, Lattice Semiconductor, Actel, Atmel, Cypress Semiconductor a QuickLogic. Podíváme-li se na údaje z roku 2001 (zdroj Paltek Corporation), vyrábělo v té době prvních pět největších výrobců (Xilinx, Altera, Lattice Semiconductor, Actel a Atmel) kolem 96 % všech vyrobených PLD obvodů. Z toho první tři největší výrobci (Xilinx, Altera a Lattice Semiconductor) vyráběli přes 87 % všech vyrobených PLD obvodů. Konkurence na poli programovatelných obvodů tedy byla a je vysoká. Tento aspekt a dále také rozšíření používání PLD obvodů v odvětvích jako jsou telekomunikace (mezi velké odběratele PLD obvodů patří např. Cisco, AT&T, Motorola, Alcatel, Siemens, atd.) způsobil, že PLD obvody zažily v posledních 10 letech poměrně velký vývoj. Tempo tohoto vývoje stále narůstá a dnešní největší programovatelné obvody typu FPGA dosahují hustoty několika desítek miliónů **ekvivalentních hradel**. Počet ekvivalentních hradel je používán jako měřítko velikosti obvodu. Stanovuje počet dvouvstupových hradel NAND nebo NOR, které by bylo možné daným programovatelným obvodem nahradit. Je to zřejmě měřítko jen velmi přibližné a nebude v úvahu to, zda daná architektura PLD je více nebo méně vhodná pro realizaci daného číslicového systému. Lepší měřítko však zatím není.

V jediném FPGA obvodu lze dnes běžně realizovat např. několik 32bitových procesorů včetně dalších podpůrných bloků pro číslicové zpracování obrazu v reálném čase, atp. Funkci některých FPGA obvodů lze dnes navíc po částech měnit přímo za běhu obvodu. Současně se dnes začínají objevovat programovatelné analogové obvody, pomocí nichž lze realizovat programovatelná analogová zapojení (např. operační sítě, analogové filtry, atd.). Tyto obvody jsou zatím oproti číslicovým PLD obvodům malé, vývoj však pokračuje velmi rychle. Analogové programovatelné obvody vyrábí např. firma Anadigm.

13.1 Historie programovatelných logických obvodů

Za vůbec první programovatelný logický obvod lze považovat paměť PROM. Paměť PROM lze využít pro realizaci kombinačních logických funkcí tak, že paměť využijeme jako tzv. vyhledávací tabulku **LUT** (angl. *Lookup Table*). V tomto případě přivádíme na adresové vodiče PROM paměti vstupní signály (proměnné). Obsah paměti PROM vytvoříme tak, že na adresy, jejichž hodnota je tvořena vektorem hodnot vstupních proměnných, uložíme hodnoty, které jsou tvořeny vektory požadovaných výstupních hodnot. Výstupní datové signály paměti PROM pak reprezentují výstupy kombinační logiky. Tímto způsobem můžeme např. paměti PROM o velikosti 2 Kb s organizací 256×8 bitů (8 adresových vodičů, 8 datových vodičů) vytvořit programovatelný obvod, kterým lze realizovat 8 kombinačních funkcí s 8 vstupními signály (proměnnými). Výhodou takovéto realizace je, že všechny realizované funkce mají stejné zpoždění ze vstupu na výstup a to pro všechny možné kombinace vstupních hodnot. Na principu generátorů logických funkcí pomocí paměti (LUT) je založena funkce obvodů **FPGA**.

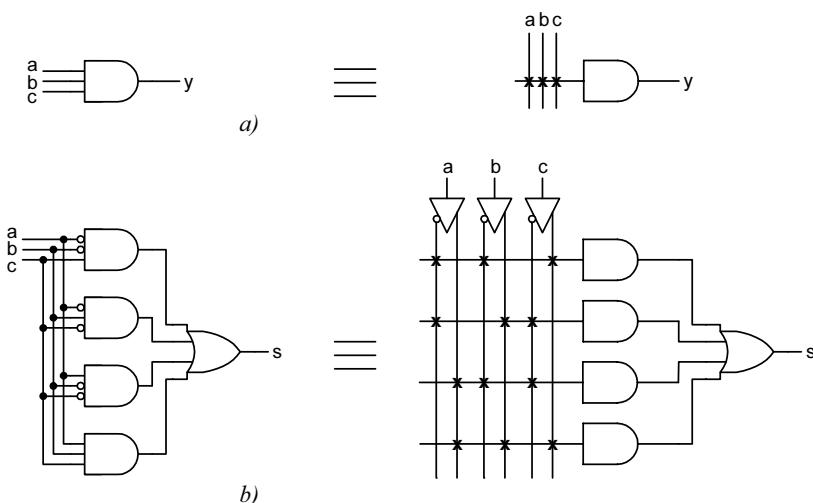
Mezi první programovatelné logické obvody lze zařadit zřejmě obvody **PLA** (angl. *Programmable Logic Array*) od firmy National Semiconductor. Obvody PLA obsahovaly pole hradel **AND** následované polem hradel **OR**. Logická funkce tedy u nich vznikala v disjunktní formě, tj. jako součet součinů. Tento způsob tvorby logické funkce se uchytíl a na tomto principu je založena funkce všech dnešních obvodů architektur **SPLD** a **CPLD**. Obvody PLA byly programované maskou (tj. již při výrobě obvodu), takže tyto obvody nemohl programovat sám koncový uživatel. V důsledku toho se tyto obvody na trhu příliš neuvaly.

V roce 1975 přišla na trh firma Signetics Corporation s obvody nazvanými **FPLA** (angl. *Field Programmable Logic Array*), konkrétně se jednalo o obvod 82S100. Po převzetí firmy Signetics firmou Philips byl tento obvod označován také jako PLS100. Obvody FPLA tvořilo programovatelné pole hradel AND následované programovatelným polem hradel OR. Tyto obvody však měly poměrně dlouhou dobu přenosu signálu ze vstupu na výstup. Toto zpoždění bylo dáno hlavně připojením výstupu hradel AND k programovatelnému poli hradel OR. Pro návrh obvodů neexistoval tehdy žádný jazyk, a tak musel návrhář nastavovat přímo hodnoty jednotlivých programovatelných buněk. Tyto nevýhody spolu s poměrně vysokou cenou způsobily malé rozšíření těchto obvodů.

Další významným pokrokem bylo uvedení obvodů **PAL**. Tyto obvody navrhla firma MMI (Monolithic Memories, Inc). Obvody PAL vycházely z obvodů FPLA a obsahovaly programovatelné pole hradel AND, které bylo následováno pevným neprogramovatelným polem hradel OR. Ke každému hradlu OR tak bylo možno připojit pouze omezený počet výstupů hradel AND (součinů). Díky tomuto zjednodušení došlo ke snížení doby přenosu signálu ze vstupu na výstup. Počet součinů, které byly připojeny na vstup hradla OR, byl na základě praktických zkušeností stanoven na osm. Velkou výhodou těchto obvodů bylo, že se daly programovat v tehdy již běžných programátorech paměti PROM. Mezi první obvody řady PAL patří dodnes dobře známé obvody PAL16L8 (kombinační výstupy) a PAL16R8 (výstupy s registry). Firma MMI dále napsala pro tyto obvody návrhový software, který umožňoval popsat číslicový systém pomocí velmi jednoduchého jazyka ve formě booleovských rovnic. Tento popis umožňoval návrhový software transformovat do výstupu, jímž již šlo v programátoru programovat obvody PAL. Tím došlo k významnému zjednodušení vlastního návrhu obsahu těchto obvodů. Tento software se jmenoval **PALASM** (PAL Assembler) a firma MMI ho zveřejnila ve formě zdrojového kódu napsaného v jazyce Fortran. Program PALASM umožňoval dokonce softwarovou simulaci navrženého obvodu. Díky tomu bylo možné ověřit funkci návrhu ještě před fyzickým naprogramováním obvodu. Díky funkcím návrhu a simulace lze PALSAM označit za první návrhový systém pro PLD obvody. Tento software pak byl mnoha dalšími firmami dále rozvíjen. Všechny zmíněné obvody dnes řadíme do první generace PLD obvodů. Za zmínu ještě stojí, že firma Signetics Corporation a MMI již mezi dnešními výrobcí programovatelných obvodů nenajdeme.

Pro kreslení schémat obvodů PAL s hradly AND o mnoha vstupech bylo přijato zvláštní zakreslování těchto hradel. *Obr. 13.1a* vlevo ukazuje standardní 3-vstupové hradlo AND, realizující logickou funkci $y = abc$. Obrázek vpravo pak ukazuje ekvivalentní zakreslení, které se používá pro obvody PAL. Vodorovný vodič na vstupu hradla reprezentuje všechny jeho vstupy (v tomto případě 3). Svislé vodiče pak reprezentují signály a , b a c . Křížky, případně hvězdičky v místě křížení vodorovného vodiče a svislých vodičů reprezentují naprogramované spojení vstupních signálů na vstupy hradla. V tomto případě jsou všechny tři signály připojeny na vstupy hradla. Tento způsob kreslení schématu se používá i u dalších

PLD obvodů (např. GAL, SPLD, CPLD, atd). Naprogramované spojení je realizováno fyzicky použitou technologií výroby obvodu, tj. například EPROM buňkou u obvodů PAL a EEPROM buňkou u obvodů typu GAL (SPLD).



Obr. 13.1 a) Standardní hradlo AND a jeho ekvivalent ve schématu obvodu PAL
b) Standardní logické schéma a jeho ekvivalent ve schématu PLD obvodi

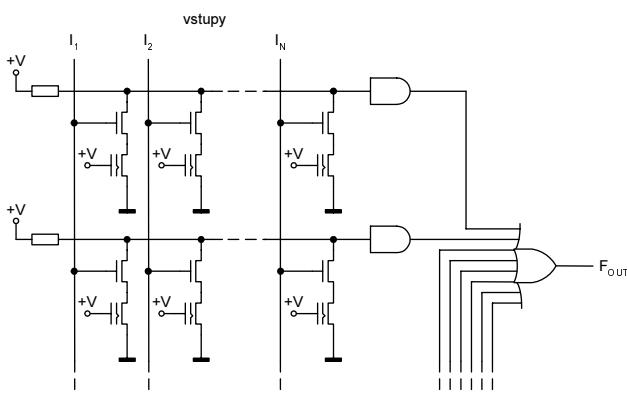
Obr. 13.1b vlevo ukazuje standardní logické schéma, které reprezentuje logickou funkci $s = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc$. Obrázek vpravo pak ukazuje ekvivalentní zakreslení téže funkce, které se používá pro PLD obvody.

Vývoj v oblasti PLD obvodů šel nezadržitelně dál. Postupně se začaly objevovat nové PLD obvody, které řadíme již do druhé generace PLD obvodů. V roce 1983 uvedla firma AMD (*Advanced Micro Devices*) obvod PAL22V10. Tento obvod byl založen na obvodech PAL popsaných v předchozím odstavci, přinesl však jedno významné vylepšení, a to tzv. **výstupní makrobuňku** (OLMC – angl. *Output Logic Macro Cell*). Tyto obvody bývají označovány jako obvody PAL s makrobuňkou. Výstupní makrobuňka byla umístěna na každém výstupu obvodu. Každou makrobuňku bylo možné naprogramovat buď jako kombinační nebo jako registrový výstup. Dále bylo možné u každé makrobuňky programovat, zda má být výstup v přímé nebo negované formě. Výstup makrobuňky byl třístavový, ovládaný jedním logickým součinem, takže bylo možné také například přepnout makrobuňku z výstupního režimu do funkce vstupu. Tento typ obvodu vyrábělo svého času nakonec kromě firmy AMD mnoho dalších firem, jmenujme např. Cypress Semiconductor, Lattice Semiconductor a Texas Instruments. Všechny dosud zmíněné obvody však měly jednu nevýhodu – byly programovatelné pouze jednou (OTP – angl. *One Time Programmable*). Díky rozvoji technologie u paměti EPROM se dostala tato technologie i do oblasti PLD obvodů a objevily se na trhu PLD obvody, jejichž obsah bylo možné smazat pomocí ultrafialového záření. Proto již bylo možné tyto obvody opakováně mazat a znova naprogramovat.

Obr. 13.2 ukazuje, jak je logická funkce ve tvaru součtu součinů vytvářena u PLD obvodů realizovaných technologií EEPROM. Každá EEPROM buňka sestává ze dvou tranzistorů.

Horní tranzistor spíná na základě vstupního signálu, druhý tranzistor je programovatelný a určuje, zda bude horní tranzistor připojen na nulový potenciál (tj. zda může sepnout). Pokud žádná buňka součinového vodiče nevede, je díky rezistoru připojenému na $+V$ součinový vodič ve stavu H. Pokud alespoň jedna buňka vede, je součinový vodič ve stavu L. Skupina EEPROM buněk, připojených na jeden součinový vodič, tak tvoří funkci logického součinu. Celé zapojení se nazývá **součinová matic** nebo **AND matic**.

Jednotlivé součiny (výstupy matic) jsou pak spojeny do součtu a tak je vytvořena logická funkce ve tvaru součtu součinů signálů I_1 , I_2 až I_N . Součtová forma kombinační funkce vyžaduje, aby vstupní proměnné byly dostupné v přímém i negovaném tvaru. Signály I_1 , I_2 až I_N jsou vytvářeny ve vstupních obvodech, a to jako přímé i negované vstupní signály (viz proměnné a , b , c v obr. 13.1b vpravo). Tomu by odpovídalo až $I_1 = a$, $I_2 = \bar{a}$, $I_3 = b$, $I_4 = \bar{b}$, atd. Druhou úlohou vstupních obvodů je odlehčení vnějších vstupů obvodu od zátěže, způsobené velkým počtem tranzistorů ve sloupcích matic.



Obr. 13.2 Tvorba logické funkce v součtovém tvaru pomocí buněk EEPROM

V roce 1984 vstoupila na trh firma DATA I/O se svým návrhovým systémem **ABEL**. Tento návrhový systém disponoval jazykem vyšší úrovně, určeným pro popis číslicových systémů (HDL – *Hardware Description Language*), který byl nazván stejně jako návrhový systém, tj. ABEL (*Advanced Boolean Expression Language*). Jazyk vycházel z tehdy již velmi rozšířeného jazyka PALASM. Jazykem ABEL lze popsat číslicový systém pomocí booleovských rovnic, pravdivostních tabulek a stavových automatů, přičemž tyto způsoby je možné kombinovat. Práva na jazyk ABEL získala po několika akvizicích firma Xilinx. Tento jazyk se dnes vyskytuje již v sedmé verzi a dodnes ho některé současné návrhové systémy podporují (např. od firmy Xilinx a Lattice Semiconductor). Podrobnější popis jazyka ABEL lze v češtině najít např. v literatuře [Kol00]. Pro návrh nových číslicových systémů založených na PLD obvodech se však doporučuje používat některý z novějších HDL jazyků, např. jazyk VHDL (viz kapitola 3) nebo jazyk Verilog.

Další vývoj PLD obvodů pokračoval s nástupem technologie paměti EEPROM a jejím využití v PLD obvodech. Této nové technologie bylo využito zejména u PLD obvodů označovaných jako **GAL** (*Generic Array Logic*). Obvody GAL lze zařadit již do třetí generace PLD obvodů. Obvody typu GAL jsou také zařazovány do třídy jednoduchých programova-

telných obvodů (SPLD). Obvodům GAL se podrobněji věnuje následující podkapitola. Na konci osmdesátých let minulého století nastává v oblasti PLD obvodů bouřlivý vývoj. Vývojem a výrobou PLD obvodů se na konci osmdesátých a začátkem devadesátých let již zabývá mnoho firem a vývoj PLD obvodů již nelze od této doby přehledně rozdělit ani stručně popsat. V průběhu tohoto období vznikají nové řady CPLD obvodů, jmennuji např. alespoň obvody MACH firmy AMD a dále vznik první řady obvodů MAX, kterou společně vyvinula firma Altera a Cypress Semiconductor. Nové obvody v této době na trh uvádí také firma Xilinx (řady XC7200 a XC7300), QuickLogic, Lattice Semiconductor, a tak by bylo možné pokračovat dál a dál. Z uvedeného je vidět, že cesta vývoje PLD obvodů nebyla a není ani dnes nijak přímočará a byla navíc od svých počátků provázená soudními sporami firem o patentová práva, a tato situace trvá dodnes. Lze však říci, že od začátku devadesátých let vyvíjí většina firem dvě od sebe velmi odlišné architektury PLD obvodů. První je architektura CPLD obvodů, která je založená na programovatelné matici hradel AND, hradlech OR a makrobuňkách (vychází tedy z původní koncepce obvodů PAL) a na programovatelných místech používá buňky EEPROM nebo FLASH.

Druhou architekturou je pak architektura FPGA obvodů, která je založená na principu generátorů logických funkcí pomocí pamětí. Nejčastěji je použita technologie SRAM (statická RAM), případně Anti-Fuse (programování průrazem izolantu v místě křížení dvouvodičů).

Historický vývoj PLD obvodů lze přehledně shrnout do tab. 13.1.

Tab. 13.1 Historie vývoje PLD obvodů

Generace	Typ PLD	Technologie
I. 1975–1983	PROM PLA, FPLA PAL	bipolární TTL CMOS
II. 1983–1985	PAL s makrobuňkou	EPROM
III. 1985–1988	GAL	EEPROM CMOS EEPROM
1988–	MACH MAX CPLD FPGA	Advanced CMOS EEPROM FLASH Fast FLASH SRAM, Anti-Fuse

Při programování PLD se do obvodu přivádějí tzv. **konfigurační data**, která definují stav vnitřních programovatelných míst. Zpravidla se přivádějí sériově, bit za bitem. K programování funkce PLD se využívají všeobecně tyto technologie:

- Buňky EEPROM nebo FLASH. Tak je tomu např. u programovatelné matice AND z obr. 13.2. Obdobně tak lze realizovat i propojovací matice (či pole) a programovatelné multiplexory.
- Buňky SRAM v podobě klopných obvodů CMOS. Každý klopný obvod ovládá spínací tranzistor. Tím lze realizovat programovatelné spojení mezi vodiči. Jedna nebo několik buněk může řídit programovatelné multiplexory. Paměť lze také realizovat např. generátory logických funkcí.

- Buňky vytvořené v křížení vodičů. Jeden vodič je vytvořen difúzí v základním materiálu integrovaného obvodu, druhý je na povrchu, vytvořen z polykrystalického křemíku. Izolantem mezi vodiči je kombinace oxidu a nitridu křemíku. Nitrid má vysokou hodnotu dielektrické konstanty, což snižuje napětí potřebné pro průraz. Průrazem dojde k vodivému propojení obou vrstev. Buňka je naprogramována nevratně. Tato technologie má název Anti-Fuse (tzv. „obrácená pojistka“).

Každá z těchto technologií má svoje **výhody i nevýhody**:

- Technologie EEPROM (FLASH) zaručuje činnost obvodu ihned po náběhu napájení, přitom obvod lze přaprogramovat. Má však největší spotřebu napájecího proudu.
- Technologie SRAM zaručuje velmi nízkou spotřebu (CMOS), ale konfigurační data se musí do obvodu přenášet při každém náběhu napájení. K tomu se využívají bud' vnější sériové paměti EEPROM (FLASH), nebo se data generují počítačem (typické při ověřování a ladění systému). Po přenesení konfiguračních dat je již vnější paměť nepotřebná a pro úsporu celkového příkonu se může odpojit od napájení.
- Technologie Anti-Fuse potřebuje nejmenší plochu na jednu programovatelnou buňku a má malý příkon. Naprogramování je velmi spolehlivé. Obvod však nelze přaprogramovat a nehodí se tedy pro experimentování.

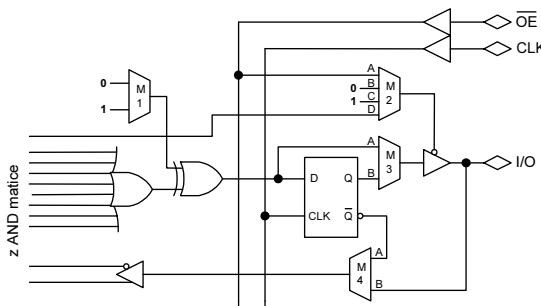
13.2 Jednoduché programovatelné logické obvody (SPLD)

Jak již bylo řečeno v předchozí kapitole, patří do této skupiny zejména obvody typu **GAL**. Obvody typu GAL patří také do skupiny elektricky reprogramovatelných PLD obvodů (angl. *Electrically Erasable Programmable Logic Device* – EEPLD, někdy také označované E²PLD). Obvody typu GAL byly a jsou dodnes poměrně oblíbeným průmyslovým standardem a vyrábí je stále několik výrobců (zejména např. firma Lattice Semiconductor a Atmel). Tyto obvody jsou typické svou programovatelnou maticí AND následovanou hradly OR a výstupními makrobuňkami. Obvody GAL se dnes vyrábí v několika velikostech, označovaných jako 16V8, 18V10, 20V8, 20V10, 22V10 a 26V12. Toto označení se používalo již u obvodů PAL a zachovalo se i u obvodů GAL, jež z obvodů řady PAL vznikly. První číslo znamená celkový počet vstupních a výstupních vývodů, druhé číslo pak znamená počet makrobuňek, přičemž každá makrobuňka může fungovat jako vstup i výstup. Vezmeme-li například obvod 16V8, disponuje tento obvod osmi vývody, které mohou fungovat jako vstup nebo výstup, a osmi vývody, které jsou pouze vstupní. U obvodů GAL se většinou používá technologie EEPROM. Jejich konfiguraci tedy lze smazat a znova je naprogramovat. V důsledku použití technologie EEPROM mají obvody GAL omezený počet programovacích cyklů. Obvyklý počet programovacích cyklů garantovaných výrobcem se u obvodů GAL pohybuje v rozsahu 100 až 1 000 (dle výrobce). Většina výrobců obvodů GAL garantuje dobu udržení konfiguračních dat 20 let.

Architektura obvodů SPLD, do níž obvody GAL patří, je tvořena následujícími programovatelnými prvky:

- velká společná programovatelná matici AND
- makrobuňky sestávající z:
 - programovatelného vícevstupového hradla OR
 - programovatelného hradla XOR
 - programovatelného klopného obvodu
 - programovatelného vstupního / výstupního obvodu

Zjednodušené schéma typické výstupní makrobuňky ukazuje obr. 13.3.



Obr. 13.3 Výstupní makrobuňka SPLD obvodu (řada GAL)

Vstupní signály do makrobuňky přicházejí z programovatelné AND matice, do které jsou zavedeny signály v přímé i negované formě, ze všech vstupních pinů. Po průchodu programovatelnou AND maticí vznikají na jednotlivých výstupech AND hradel logické součiny zvolených vstupních signálů (angl. *product term*). Tyto součiny jsou pak přiváděny na vstupy hradla OR příslušné makrobuňky. Na výstupu hradla OR je tak formována požadovaná logická funkce ve tvaru součtu součinů. Takto vytvořený signál dále prochází programovatelným hradlem XOR. Toto hradlo slouží jako řízená negace (dle nastavení multiplexoru M1). Po průchodu hradlem XOR je signál přiveden na vstup klopného obvodu. Jako hodinový signál tohoto klopného obvodu slouží globální hodinový signál *CLK*.

Makrobuňka obvodu GAL může pracovat celkem ve třech režimech. První režim je tzv. jednoduchý (angl. *simple mode*). V tomto režimu lze přivést do makrobuňky až 8 součinů a makrobuňka funguje buď jako vstup (M2 nastaven na vstup C, M4 je nastaven na vstup B), nebo jako jednoduchý kombinační výstup s případnou zpětnou vazbou do AND matice (M2 nastaven na vstup B, M3 je nastaven na vstup A, M4 je nastaven na vstup B). Druhý režim je o trochu složitější (angl. *complex mode*). V tomto režimu lze přivést do makrobuňky až 7 součinů a makrobuňka v tomto režimu funguje jako kombinační třístavový výstup, případně jako kombinační obousměrný výstup (M2 nastaven na vstup A nebo D, M3 je nastaven na vstup A, M4 je nastaven na vstup B). Poslední (třetí) režim, je tzv. registrový (angl. *registered mode*). V tomto režimu lze přivést do makrobuňky až 8 součinů a makrobuňka funguje jako registrový výstup (M2 nastaven na vstup A nebo B, M3 je nastaven na vstup B, M4 je nastaven na vstup A).

Firmy v průběhu let obvody GAL neustále technologicky vylepšovaly a dnes dosahují obvody GAL celkem vysokých hodinových kmitočtů f_{\max} a nízkého zpoždění t_{pd} (zpoždění ze vstupu na výstup v režimu kombinačního obvodu). Například nejrychlejší obvody GAL firmy Lattice Semiconductor dnes (rok 2006) dosahují $f_{\max} = 250$ MHz a $t_{pd} = 3,5$ ns. Firma Lattice Semiconductor obvody GAL vylepšila i po stránce programování a vyrábí dnes řadu obvodů označenou jako ispGAL. Tyto obvody jsou rychlejší než klasické obvody GAL (některé dosahují $f_{\max} = 455$ MHz a $t_{pd} = 2,3$ ns). Obvody řady ispGAL lze navíc programovat již osazené na desce plošných spojů pomocí sériového rozhraní. Takovéto programování označuje většina výrobců zkratkou ISP (*In-System Programmable*). Tento typ programování je dnes dostupný prakticky u všech v současné době vyráběných PLD obvodů. Programování obvodu v systému (ISP) probíhá nejčastěji pomocí rozhraní JTAG, které je definováno standardem *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE Std 1149.1)*, případně pomocí novějšího standardu *IEEE Standard for In-System Configuration of Programmable Devices (IEEE Std 1532-2002)*. Rozhraní JTAG používá pro komunikaci s obvody celkem 5 signálových vodičů (*TCK*, *TDI*, *TDO*, *TMS* a *TRST*). Většina výrobců však signál *TRST* nepoužívá a programování tak probíhá pomocí 4 signálových vodičů *TCK*, *TDI*, *TDO* a *TMS*. Více obvodů lze řadit do tzv. JTAG řetězce a tak např. všechny obvody na desce programovat pomocí jednoho JTAG rozhraní. Vlastní programování obvodů řady ispGAL trvá několik sekund, takže testování navrženého obvodu může probíhat přímo v cílové aplikaci. U obvodů řady ispGAL garantuje výrobce vyšší počet programovacích cyklů (až 10 000) při zachování stejné doby udržení konfiguračních dat (tj. 20 let). Obvody architektury SPLD dosahují velikosti od 8 do 12 makrobuňek a vyrábějí se nejčastěji v pouzdrech DIP, PLCC a SSOP. Napájecí napětí těchto obvodů bývá nejčastěji 5 V nebo 3,3 V. Vyskytuje se však i obvody určené pro napájecí napětí 2,5 V nebo 1,8 V.

13.3 Komplexní programovatelné logické obvody (CPLD)

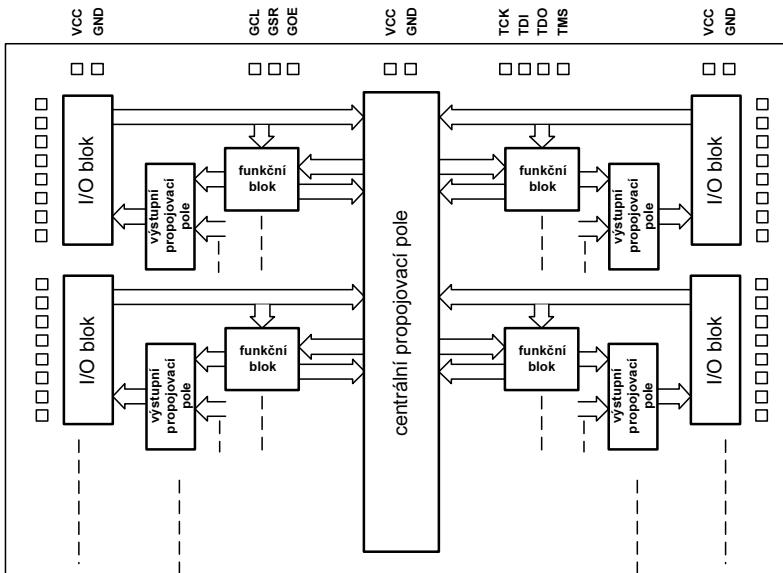
Obvody typu CPLD patří podobně jako obvody GAL do skupiny elektricky reprogramovatelných PLD obvodů (EEPLD). Většina CPLD obvodů je programovatelná přímo v cílovém systému, nesou tedy i označení ISP. Tyto obvody jsou typické, podobně jako obvody GAL, svou programovatelnou maticí hradel AND následovanou hradlem OR a makrobuňkou. Na výstupu hradla OR je tak stejně jako u obvodů GAL formována požadovaná logická funkce ve tvaru součtu součinů. Od obvodů GAL se však obvody CPLD liší hlavně velkým centrálním propojovacím polem. Mezi starší obvody CPLD tak lze zařadit například obvody řady EP a řadu obvodů MAX5000 od firmy Altera, řady XC7200/7300 firmy Xilinx, atd.

Makrobuňky jsou sdružovány do větších skupin a tvoří tzv. **funkční bloky**. Pro architekturu obvodů CPLD jsou charakteristické tyto čtyři programovatelné prvky:

1. velké centrální propojovací pole
2. programovatelné funkční bloky, uspořádané kolem propojovacího pole, sestávající z:
 - programovatelné matice AND
 - několika (8) makrobuňek s alokátory součinů
3. výstupní propojovací pole
4. vstupní/výstupní bloky (I/O bloky)

Všechny výše uvedené stavební prvky mají u různých výrobců různá označení, jejich význam a funkce je však velmi podobná nebo dokonce shodná.

Zjednodušené blokové schéma obecného CPLD obvodu je na obr. 13.4:



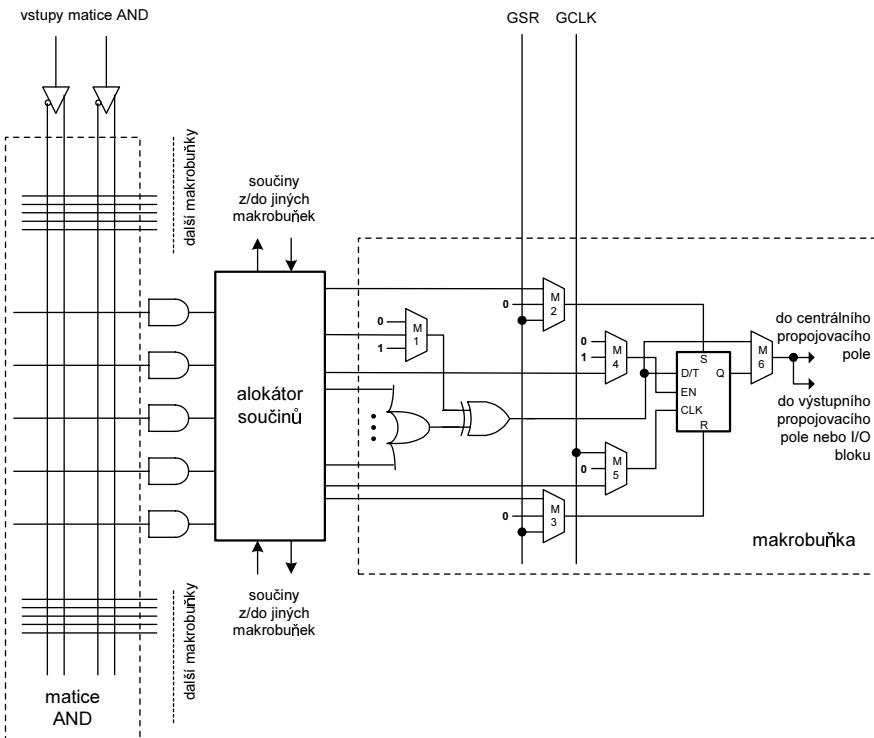
Obr. 13.4 Blokové schéma obvodu CPLD

Pomocí makrobuňek lze realizovat různě složité kombinační a sekvenční logické či paměťové funkce. Přes programovatelné **vstupní/výstupní bloky** lze přivádět vstupní signály z vývodů obvodu nebo naopak vyvádět výstupní signály. Na rozdíl od jednodušších SPLD, kde vstupní/výstupní obvody jsou přímo spojeny s makrobuňkou, jsou však u CPLD zásadně od makrobuněk odděleny a tvoří samostatný I/O blok, do kterého mohou výstupní signály z makrobuněk vstupovat přes programovatelné **výstupní propojovací pole**. Tím se všeobecně zlepší využití jak makrobuněk, tak výstupních obvodů.

Všechny vstupní/výstupní bloky a všechny makrobuňky lze spolu vzájemně propojovat pomocí **centrálního programovatelného propojovacího pole**.

Všechny uvedené programovatelné prvky udržují svoji konfiguraci většinou v buňkách typu EEPROM nebo FLASH. Obvody v důsledku toho mají omezený počet programovacích cyklů. Obvyklý počet programovacích cyklů garantovaných výrobci CPLD obvodů se pohybuje v rozsahu 1 000 až 10 000 cyklů (dle výrobce). Programování obvodu probíhá většinou pomocí sériového rozhraní. Nejčastěji obvody podporují pro programování rozhraní standardu JTAG (IEEE Std 1149.1), případně novější standard IEEE Std 1532-2002. Díky tomu jsou tyto obvody snadno programovatelné již přímo v cílovém systému (ISP). Vlastní programování obvodu trvá několik sekund, takže testování navrženého číslicového systému tak může probíhat přímo v cílové aplikaci. Některé CPLD obvody podporují i testování pomocí standardu JTAG BST (Boundary-Scan Testing). Díky tomu lze u takových obvodů provádět např. testy správného osazení obvodu v desce, zjišťovat hodnoty na pinech obvodu při plné funkci obvodu, atd.

Hlavním blokem, ve kterém se v obvodech typu CPLD vytváří kombinační nebo sekvenční logické funkce, je makrobuňka. Zjednodušené zobecněné schéma makrobuňky obvodu typu CPLD uvádí obr. 13.5. Makrobuňky CPLD obvodů konkrétních výrobců se od této uvedené makrobuňky budou v detailech lišit, pro pochopení funkce však toto zjednodušené zobecněné schéma postačí.



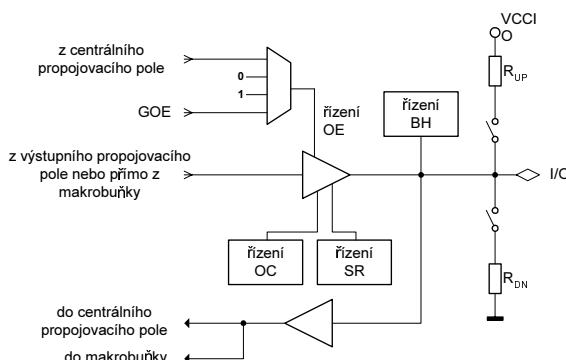
Obr. 13.5 Makrobuňka obvodu CPLD a její místo ve funkčním bloku

Do matic AND jednotlivých funkčních bloků lze zpravidla zavést několik desítek vstupních signálů (u většiny výrobců se dnes toto číslo pohybuje kolem 36 vstupních signálů). Vstupní signály jsou dodávány do programovatelné AND matice v přímé i negované formě. Po průchodu AND maticí vznikají na jejích jednotlivých výstupech logické součiny. **Alokátor součinů** umožňuje předávání jednotlivých součinů nebo i jejich součtu mezi sousedními makrobuňkami a tak umožňuje vytvářet podstatně složitější kombinační funkce, než jednotlivé makrobuňky – ty totiž mají omezený počet součinů, vyvedených z AND matice. Toto uspořádání podstatně rozšiřuje možnosti obvodů CPLD oproti obvodům SPLD.

Součiny z AND matice jsou též přivedeny na hradlo OR v makrobuňce a tak je formována, podobně jako u obvodů SPLD, požadovaná logická funkce ve tvaru součtu součinů. Takto vytvořený signál dále prochází programovatelným hradlem XOR. Toto hradlo slouží, podobně jako u obvodů SPLD, jako řízená negace (funkci určuje nastavení multiplexoru M1). Po průchodu hradlem XOR je signál přiveden na vstup programovatelného klopného obvodu. Tento klopný obvod lze většinou konfigurovat tak, aby realizoval funkci klopného

obvodu typu D, T, JK nebo RS. Jako hodinový signál tohoto klopného obvodu lze přivést buď některý ze součinů z matici, nebo některý globální hodinový signál *GCLK* (toto určuje nastavení multiplexoru M5). Podobně lze přivést jako řídící signály *R* a *S* některý ze součinů, nebo některý globální signál *GSR* (toto určuje nastavení multiplexorů M2 a M3). Funkci klopného obvodu lze blokovat pomocí jeho vstupu *EN*, na nějž lze přivádět opět signál z některého ze součinů (toto určuje nastavení multiplexoru M4). Pokud je třeba, aby makrobuňka realizovala pouze kombinační funkci, může signál pomocí multiplexoru M6 klopný obvod obejít (tzv. *bypass*). Výsledný signál pak lze vést pomocí výstupního propojovacího pole na některý z I/O bloků, tj. na vývod (pin) obvodu, a také zpět do centrálního propojovacího pole, nebo v některých případech také do sousedních makrobuněk.

Výstupní budiče CPLD obvodů nejsou součástí makrobuňky, jako tomu je v případě obvodů SPLD (GAL), ale vstupní/výstupních bloků (I/O bloků). I/O blok umožňuje vstup a výstup signálu z/do obvodu. Zjednodušené zobecněné schéma typického I/O bloku obvodu typu CPLD uvádí obr. 13.6. Připomínáme opět, že se I/O bloky CPLD obvodů konkrétních výrobců mohou od tohoto uvedeného I/O bloku v detailech lišit, pro pochopení funkce však toto zjednodušené zobecněné schéma postačí.



Obr. 13.6 I/O blok obvodu CPLD

Signál z makrobuňky přichází přes výstupní propojovací pole do I/O bloku, který obsahuje výstupní budiče. U některých CPLD obvodů se nemusí výstupní propojovací pole vyskytovat, signál pak přichází z makrobuňky rovnou do I/O bloku. Výstupní **třístavové budiče** v CPLD obvodech jsou řízeny signálem *OE* (angl. *output enable*). Multiplexorem lze vybrat způsob řízení – výstup dvojstavový (L a H), výstup ve stavu Z, řízení signálem přiváděným z centrálního propojovacího pole, nebo řízení globálním signálem *GOE*, přiváděným do celého CPLD zvenku. Výstupní budiče většinou mohou pracovat i ve funkci **otevřeného kolektoru** (OC). Je možné ovládat i **rychlost přeběhu** (angl. *slew rate – SR*) výstupního napětí. Řízení rychlosti přeběhu je většinou výrobců realizováno ve dvou stupních (velká nebo malá rychlosť přeběhu). Přepnutím výstupů obvodu do režimu malé rychlosti přeběhu lze snížit příkon obvodu a také snížit jeho elektromagnetické vyzařování (samozřejmě za cenu, že na těchto výstupech je pak omezena maximální dosažitelná frekvence).

Pokud je I/O blok využíván jako vstup, může se uplatnit obvod s funkcí „bus hold“, známý také pod pojmem **aktivní terminátor** (viz podkapitola 5.2). U většiny CPLD obvodů lze u I/O bloku dále konfigurovat, zda má být vývod integrovaného obvodu (pin) připojen

vnitřním rezistorem na napájecí napětí V_{CCIO} (angl. *pull-up*), nebo na zem (angl. *pull-down*). Tyto rezistory jsou realizovány jako tranzistory MOS s odporem kanálu několik desítek kiloohmů. Vstupní signál je dále zesílen ve vstupním zesilovači a odtud je veden do centrálního propojovacího pole, případně u některých CPLD obvodů také přímo do některé makrobuňky.

Dnešní CPLD obvody dosahují velikosti zhruba od 32 do 1024 makrobuňek a vyrábějí se v různých pouzdrách od typu PLCC, přes pouzdra typu QFP až k různým pouzdrům typu BGA. Přepočteme-li velikost CPLD obvodu v makrobuňkách na počet ekvivalentních hradel, dostáváme se k velikostem zhruba od 600 do 300 000 ekvivalentních hradel. Většina současných CPLD obvodů má několik napájecích vývodů (minimálně dva). Jedno napájecí napětí je určeno pro napájení I/O bloků, druhé napájecí napětí je určeno pro jádro obvodu. Většina výrobců sdružuje I/O bloky do skupin, přičemž každá skupina I/O bloků má svůj vlastní napájecí přívod. Díky tomu může být každá skupina I/O bloků napájena jiným napětím a tak může pracovat, pokud to obvod podporuje, s jinými úrovněmi L a H. Tak např. jedna skupina I/O bloků může splňovat napěťový standard LVCMOS 3,3 V a jiná např. LVCMOS 2,5 V. Díky této funkci mohou dnešní CPLD obvody fungovat, mimo jiné, také jako převodníky napěťových logických úrovní. Napájecí napětí jádra dnešních CPLD obvodů bývá nejčastěji 3,3 V. V širší míře se dnes objevují i obvody s napájecím napětím jádra 2,5 V nebo 1,8 V. V menší míře se dnes vyskytují CPLD obvody určené pro napájecí napětí 5 V. Většina CPLD obvodů s napájecím napětím 3,3 V je však 5 V-tolerantní, takže není problém připojovat tyto obvody k 5 V obvodům.

Mezi největší a nejnámější výrobce CPLD obvodů patří firmy Lattice Semiconductor (např. řady ispMACH4A5, ispMACH4000 a ispXPLD5000), dále Altera (např. řady MAX7000B/AE/S a MAX3000A) a Xilinx (např. řady XC9500/XL/XV, CoolRunner XPLA3 a CoolRunner-II).

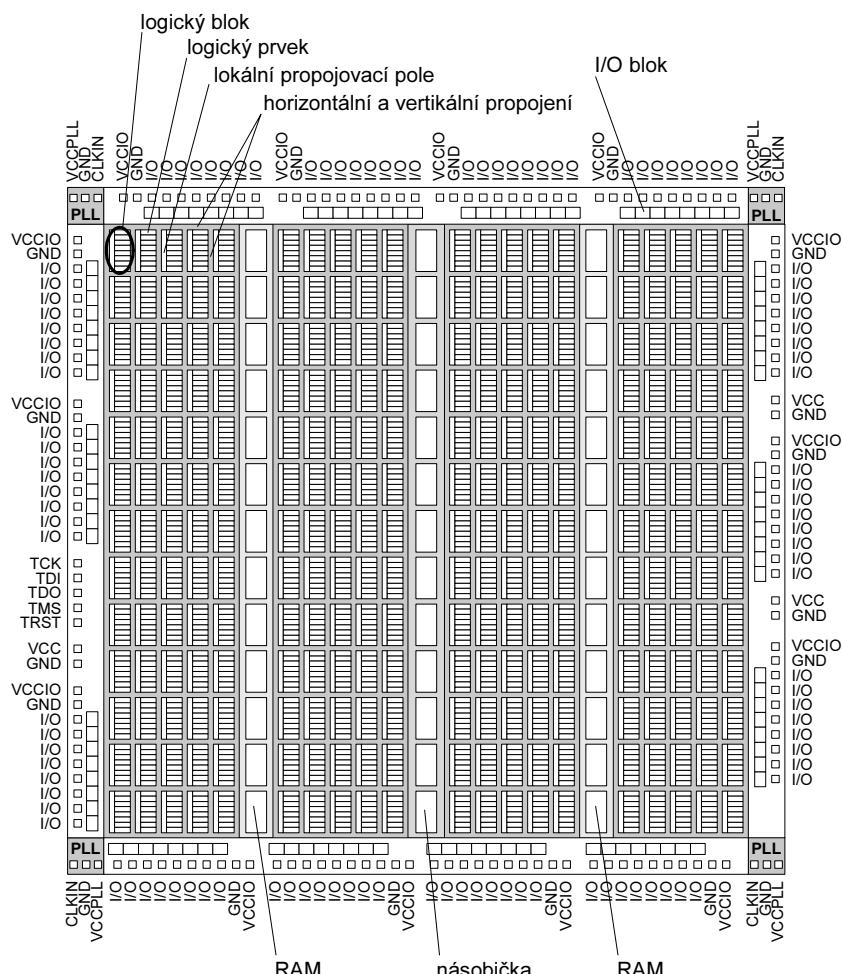
V poslední době se začala objevovat nová kategorie obvodů, kterou výrobci označují jako CPLD, avšak o CPLD obvody v pravém slova smyslu se nejedná. Tyto nové řady „CPLD“ obvodů totiž nejsou založeny na tvorbě logické funkce pomocí matic AND, hradlech OR, makrobuňkách a velkém centrálním propojovacím poli, ale jsou založeny na malých generátorech logických funkcí s paměťmi typu SRAM (LUT tabulkách), klopných obvodech, horizontálních a vertikálních propojeních. Z tohoto důvodu mají tyto obvody mnohem blíže k obvodům FPGA, z jejichž principu tyto obvody vzešly. Co si však tyto nové řady obvodů převzaly z architektury CPLD je, že svoji konfiguraci udržují ve vnitřní paměti FLASH případně EEPROM. Při zapnutí napájení pak obvody tuto konfiguraci zkopírují do SRAM buňek. Díky tomu jsou tyto obvody schopny udržet svoji konfiguraci i po vypnutí napájení a nemusí mít připojenu externí konfigurační paměť jako obvody typu FPGA, které jsou založeny pouze na buňkách SRAM. Představiteli těchto nových obvodů je např. řada obvodů MAX II firmy Altera nebo řada MachXO firmy Lattice Semiconductor.

13.4 Programovatelná logická pole (FPGA)

Obvody architektury FPGA patří, podobně jako předchozí architektury obvodů SPLD a CPLD, do skupiny elektricky reprogramovatelných PLD obvodů (EEPLD) a prakticky všechny FPGA obvody jsou programovatelné přímo v cílovém systému, nesou tedy i označení ISP. Výše uvedené architektury SPLD a CPLD obvodů vycházely z koncepce implementace logic-

kých rovnic pomocí matice AND, hradla OR a makrobuňky. Obvody architektury FPGA jsou založeny na malých generátorech logických funkcí s pamětími (LUT tabulkách), klopných obvodech a mnoha horizontálních a vertikálních propojeních. Největší současné FPGA obvody obsahují několik stovek tisíc LUT tabulek a klopných obvodů. Nejčastěji se u architektury FPGA používá technologie SRAM (např. obvody firmy Altera, Xilinx a Lattice Semiconductor). Někteří výrobci pak používají technologii průrazu izolantu Anti-Fuse (např. firma Actel).

Obvody architektury FPGA kombinují výhody PLD obvodů s výhodami plně zákaznických VLSI obvodů a umožňují implementaci rozsáhlých číslicových systémů. Díky podobnosti s plně zákaznickými obvody se dnes používají pro návrh a simulaci jak FPGA obvodů, tak i obvodů plně zákaznických velmi podobné návrhové systémy. Velkou výhodou FPGA obvodů je pak i poměrně snadný přechod s návrhem k plně zákaznickým obvodům, jakmile byl prototyp ověřen v obvodu FPGA. Zjednodušené blokové schéma obecného FPGA obvodu je na obr. 13.7.



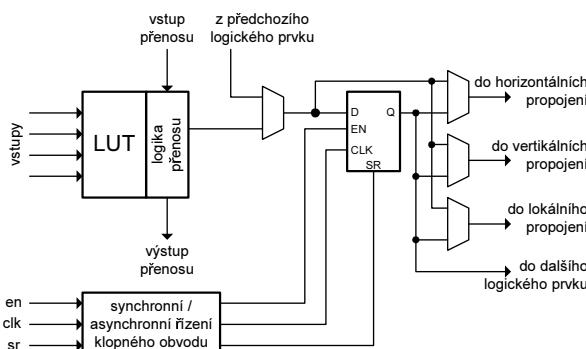
Obr. 13.7 Blokové schéma obvodu FPGA

Základem obvodů FPGA jsou tři stavební prvky:

1. programovatelné **logické bloky** tvořené:
 - logickými prvky, které obsahují:
 - » generátor logické funkce (vyhledávací tabulka LUT)
 - » klopný obvod
 - lokálním propojovacím polem
2. programovatelné horizontální a vertikální **propojení**
3. programovatelné **vstupní/výstupní bloky** (I/O bloky)

K těmto třem základním stavebním prvkům přibyly v poslední době tzv. specializované bloky, mezi něž patří např. násobičky, paměti, bloky pro úpravu hodinových signálů, nebo dokonce celé procesory.

Výše jmenované stavební prvky mají u různých výrobců různá označení, jejich funkce je však podobná. Logické bloky obsahují vždy několik logických prvků (typicky osm logických prvků) a lokální propojovací pole, které tyto logické prvky může propojovat mezi sebou a dále umožňuje také jejich propojení s logickými prvky v nejbližších logických blocích. V jednotlivých logických prvcích lze realizovat jednoduché kombinační nebo sekvenční obvody. Velmi zjednodušené blokové schéma logického prvku je na obr. 13.8.



Obr. 13.8 Zjednodušené blokové schéma logického prvku

Logický prvek obsahuje generátor logické funkce pomocí paměti (LUT tabulku). Dále obsahuje podpůrnou logiku pro vstup a výstup aritmetického přenosu. Tato logika je využívána v případech, že je logický prvek použit pro realizaci čítače, scítačky, odčítáčky, a podobně. Logický prvek dále obsahuje klopný obvod a blok asynchronního či synchronního řízení tohoto klopného obvodu. Prostřednictvím multiplexorů jimiž lze vytvořenou kombinační nebo sekvenční funkci propojovat na další prvky, spoje, atd. Dlouhou dobu byly používány LUT tabulky, které měly 4 vstupy a jeden výstup (tj. organizaci paměti 16×1). V současné době používají některé nejnovější FPGA obvody LUT tabulky mající až šest vstupů a až dva výstupy. U některých výrobců lze navíc tyto nové LUT tabulky nastavovat do různých konfigurací, např. jako několik LUT tabulek o třech až šesti vstupech.

Horizontální a vertikální propojení slouží k propojení vzdálenějších logických bloků a k propojení s I/O bloky. Z obr. 13.7 je zřejmé, že architektura FPGA obvodů nedisponuje, na rozdíl od architektury CPLD, žádným velkým centrálním propojovacím polem. Naopak propojení jsou rozmištěna rovnoměrně po celé ploše obvodu.

I/O bloky v FPGA obvodech mají podobné vlastnosti jako v obvodech CPLD. To se týká řízení rychlosti přeběhu výstupního napětí, funkce „bus hold“, řízení třístavového výstupu, a programovatelných rezistorů. Řízení rychlosti přeběhu je většinou realizováno ve dvou stupních (velká nebo malá rychlosť přeběhu).

V posledních letech k výše popsaným základním stavebním prvkům ještě postupně přibyly specializované bloky:

- vložené bloky paměti
- fázové závěsy PLL
- vložené bloky násobiček

Jako první přibyly do FPGA obvodů vložené **bloky paměti**. Tyto paměťové bloky jsou umístěny v FPGA obvodu ve sloupci (např. namísto sloupce logických bloků). Blok vložené paměti lze zkonfigurovat do různých paměťových funkcí, jako např. ROM, jedno- i dvoubránové RAM, FIFO, a nebo jako velký posuvný registr s odbočkami (což je vhodné např. pro realizaci číslicových filtrů, apod.). Velikost paměťového bloku je u jednotlivých řad FPGA obvodů i výrobců různá a pohybuje se zhruba od 0,5 Kb do 512 Kb. Organizaci paměťových bloků lze nastavovat, takže např. pro 4 Kb blok lze nastavit organizaci od $4K \times 1$, $2K \times 2$, $1K \times 4$ až po 128×32 . Paměťové bloky lze spojovat dohromady jak do šířky, tak do hloubky. Díky tomu lze vytvořit paměťový blok s organizací vyhovující individuálním požadavkům. Některé řady FPGA obvodů mají dokonce několik druhů různě velikých paměťových bloků (např. řada Stratix/II od firmy Altera má paměťové bloky o velikostech 0,5 Kb, 4 Kb a 512 Kb).

Fázové závěsy (PLL – angl. *Phase Locked Loop*) jsou prvním analogovým blokem, který se v obvodech FPGA objevil. Pomocí fázového závěsu lze v FPGA obvodech vytvořit z jednoho vstupního kmitočtu několik různých kmitočtů s různým fázovým posunem. Fázové závěsy v FPGA obvodech obsahují napětím řízený oscilátor, několik děliček, fázový detektor, nábojovou pumpu, filtr typu dolní propust, detekci zavěšení, obvody pro řízení střídy a obvody pro výběr požadovaného fázového posunu výstupů. Minimální vstupní kmitočet fázových závěsů v obvodech FPGA bývá zhruba v rozsahu od jednotek MHz do stovek MHz. Protože se jedná o analogový blok, mají fázové závěsy vlastní napájecí přívod, kterému by měla být při návrhu schématu i desky plošných spojů věnována náležitá pozornost.

Číslicové násobičky jsou umístěny v FPGA obvodu, podobně jako paměťové bloky, ve sloupci. V dnešních FPGA obvodech se vyskytují většinou násobičky o velikostech 9×9 bitů a 18×18 bitů. Násobičky lze zkonfigurovat jako jednu větší násobičku nebo několik menších. Bloky násobiček obsahují také sčítáčku, takže je lze použít i pro realizaci funkce násobení se součtem nebo jako násobičku s akumulátorem. Vložené bloky násobiček lze s výhodou využít při realizaci funkcí číslicového zpracování signálů jako jsou např. filtry s konečnou impulzní odezvou (FIR – angl. *Finite Impulse Response*), filtry s nekonečnou impulzní odezvou (IIR – angl. *Infinite Impulse Response*), rychlé Fourierovy transformace (FFT – angl. *fast Fourier transform*), diskrétní kosinové transformace (DCT – angl. *discrete cosine transform*) a dalších.

Detailnější zapojení všech výše uvedených stavebních prvků zde neuvádíme, protože se jedná o poměrně rozsáhlá zapojení, která se navíc u jednotlivých výrobců značně liší. Pro podrobnější seznámení s FPGA obvody doporučujeme prostudovat katalogové listy FPGA obvodů, které jsou volně dostupné na internetu přímo od výrobců těchto obvodů.

Všechny výše uvedené stavební prvky FPGA obvodů udržují svoji konfiguraci většinou ve statické paměti RAM (SRAM). Programovaní obvodu tedy lze opakovat bez omezení. Programování obvodu probíhá buď automaticky, při zapnutí napájecího napětí, z externě připojené paměti EEPROM nebo FLASH (většinou sériové), a nebo pomocí sériového rozhraní z počítače. Nejčastěji obvody podporují pro programování rozhraní standardu JTAG (*IEEE Std 1149.1-2001*), případně novějšího standardu *IEEE Std 1532-2002*. Díky tomu jsou tyto obvody snadno programovatelné již přímo v cílovém systému. Protože jsou dnešní FPGA obvody velmi rozsáhlé, dosahují i konfigurační data poměrně velkých objemů (řádově až desítky megabitů). Z tohoto důvodu podporují FPGA obvody kompresi konfiguračních dat. Díky tomu lze zmenšit velikost konfiguračních dat o cca 30–50 % a použít tak pro jejich uložení menší paměť. Vlastní programování obvodu trvá při konfiguraci z externí paměti řádově desítky až stovky milisekund, dle velikosti obvodu. Při konfiguraci obvodu pomocí rozhraní JTAG se programovací časy pohybují dle velikosti obvodu řádově v jednotkách sekund. Z důvodu ochrany návrhu (např. odcizení a následné plagiátorské výroby kopie celého zařízení) podporují nové FPGA obvody šifrování konfiguračních dat (např. pomocí AES128/256), takže obvod je konfigurován zašifrovanými daty a dešifrování a dekomprese probíhá až uvnitř FPGA obvodu. Dešifrovací klíč je uložen přímo v obvodu FPGA v paměti EEPROM, nebo ve vnitřní paměti SRAM, kterou je pak ale třeba zálohovat, např. baterií.

Dnešní obvody FPGA dosahují velikosti zhruba od 500 do 200 000 LUT tabulek a vyrábějí se v různých pouzdrech od typu PLCC, přes pouzdra typu QFP až k různým pouzdrům typu BGA. Přepočteme-li velikost FPGA obvodu v LUT tabulkách na počet ekvivalentních hradel, dostáváme se k velikostem zhruba od 10 000 k desítkám miliónů ekvivalentních hradel.

FPGA obvody mají několik napájecích vývodů. Jedno napájecí napětí je určeno pro jádro obvodu, další napájecí napětí jsou určena pro napájení I/O bloků a další pak pro napájení fázových závesů. I/O bloky jsou sdružovány do skupin, přičemž každá skupina I/O bloků má vlastní napájecí vývod. Díky tomu může být každá skupina I/O bloků napájena jiným napětím. Díky této funkci mohou FPGA obvody fungovat jako převodníky napěťových logických úrovní. I/O bloky dnes podporují mnoho napěťových standardů. Jmenujme zde alespoň ty nejčastější – LVTTL, LVCMOS (3,3 V; 2,5 V; 1,8 V a 1,5 V); 3,3 V PCI; SSTL a HSTL (1,8 V; 1,5 V a 1,2 V). Z diferenčních napěťových standardů pak jmenujme SSTL, HSTL, LVDS, LVPECL a HyperTransport. Napájecí napětí jádra FPGA obvodů se pohybuje dle použité technologie od 3,3 V (technologie 300 nm) přes 2,5 V (220 nm); 1,8 V (150 nm); 1,5 V (130 nm); 1,2 V (90 nm) až k 1,0 V (65 nm). Mezi největší a nejznámější výrobce obvodů FPGA patří například firmy Altera (např. řady FLEX, APEX, ACEX, Cyclone/II a Stratix/II) a Xilinx (např. řady Spartan/XL/II/IIE/3/3E/3L a Virtex/E/EM/II/4/5).

Od začátku devadesátých let do dnešní doby se zvětšila kapacita FPGA obvodů více než 200×, rychlosť se zvětšila více než 20× a cena přepočtená na ekvivalentní hradlo se snížila více než 300×. Výpočetní výkon, který lze v dnešních největších FPGA obvodech realizovat, je velmi vysoký. Pro představu můžeme uvést následující příklad. Např. sčítání v plovoucí řádové čárce s jednoduchou přesností lze dnes v obvodu XC4VLX200 (řada Virtex 4 firmy

Xilinx) realizovat s výkonem $49 \cdot 10^9$ operací v plovoucí řádové čárce za sekundu (FLOPS – angl. *Floating Point Operations Per Second*, 10^9 FLOPS = 1 GFLOPS), s dvojitou přesností pak s výkonem 21 GFLOPS. Současné FPGA obvody vyráběné 90 nm technologií dosahují velikosti přes 200 000 logických prvků (LUT + klopný obvod) a max. frekvence hodinových impulzů kolem 500 MHz. Přechod na 65 nm technologii umožní dosáhnout zhruba dvojnásobných hodnot. Dá se předpokládat, že přechod na 45 nm technologii umožní dosáhnout hodnot kolem 1 milionu logických prvků a max. frekvence hodinových impulzů kolem 2 GHz. Díky snadné a rychlé změně obsahu FPGA obvodu a velkému výpočetnímu výkonu, který lze v dnešních FPGA obvodech realizovat, se např. začínají FPGA obvody používat jako rekonfigurovatelné výpočetní koprocesory k CPU. Z uvedeného je vidět, že architektura FPGA obvodů zažila a i nadále zažívá rychlý vývoj.

13.5 Základní dynamické parametry PLD

Programovatelné logické obvody mohou pracovat jako obvody kombinační, nebo častěji jako obvody sekvenční. Pro dynamické parametry PLD proto platí přiměřeně totéž, co bylo vysvětleno v kapitole 6, 7 a 10.

- **Doba zpoždění** ve funkci kombinačního obvodu t_{pd} je doba od změny signálů na vstupech obvodu do změny signálů na jeho výstupech. Je podstatná pro režim bez hodinových impulzů u ryze kombinačního obvodu. U sekvenčního obvodu Mealyho typu je to zpoždění obvodu v době mezi hodinovými impulzy.
- Minimální **perioda hodinových impulzů** T_{CLK} (též maximální kmitočet hodinových impulzů f_{CLK}) je definována pro případ, že vstupní signály zůstávají konstantní nebo se mění jen v bezpečné době mezi hodinovými impulzy.
- **Doba předstihu** t_s je doba, po kterou vstupní signály musí být konstantní až do aktivní hrany hodinového impulzu.
- **Doba zpoždění po hodinovém impulzu** t_{pCO} je doba od aktivní hrany hodinového impulzu do změny výstupních signálů.

Symboly pro výše uvedené doby se mohou v různých firemních publikacích lišit, význam však zůstává.

Dynamické parametry u programovatelných logických obvodů jsou závislé na vnitřních cestách signálů. U obvodů CPLD je situace jednodušší, neboť cesty signálů jsou do jisté míry pevně dány a jedná se jen o jejich výběr. Výrobci uvádějí korekční vztahy pro výše uvedené doby, kterými jsou respektovány logické zátěže a způsob využití vnitřních bloků. Složitější situace je u obvodů FPGA, kde cesty signálů nejsou předem definovány a v procesu návrhu budou teprve vytvářeny. Jednotlivé doby proto musí dodatečně dopočítat návrhový systém.



MIKROPROGRAMOVÝ AUTOMAT

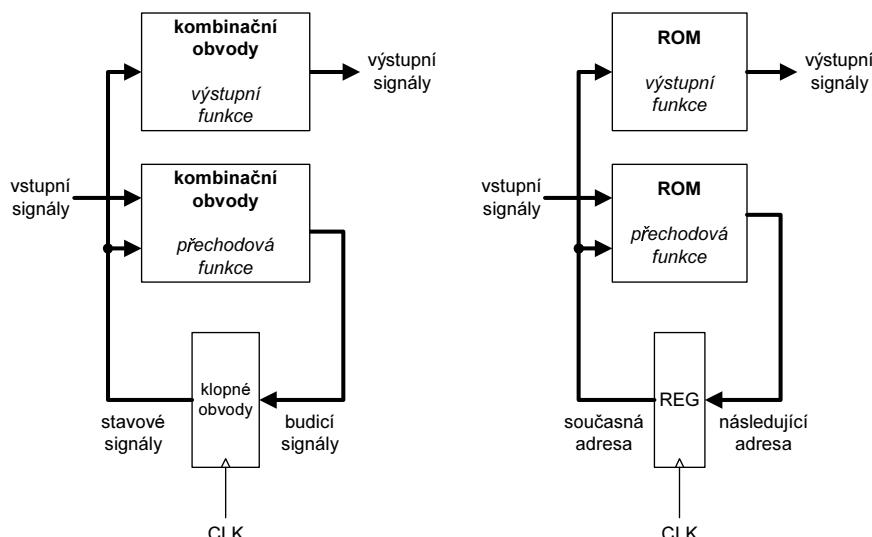
Mikroprogramový automat patří mezi konečné automaty. Jeho obvodová implementace je však odlišná od běžného synchronního sekvenčního obvodu, tak jak byl prezentován v předchozím textu. Tam se jednalo o implementaci, všeobecně nazývanou „pevně zapojená logika“. Mikroprogramový automat má následující specifické vlastnosti:

- Kombinační obvody, které jsou součástí sekvenčního obvodu, jsou v mikroprogramovém automatu realizovány **pamětí ROM**.
- Mikroprogramový automat má **velký počet vstupních proměnných**, ale **velmi omezené větvení** v grafu přechodů – velmi často jen na dvě větve.
- Používá se jiná terminologie.

Důvod pro název „mikroprogramový“ vyplýne z dalšího textu.

14.1 Základní obvody mikroprogramového automatu

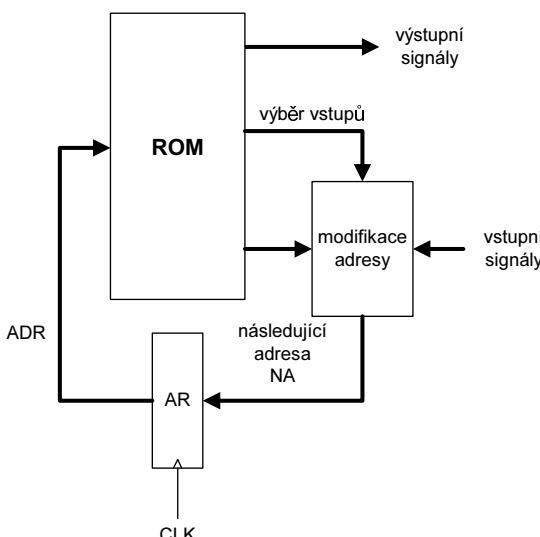
Obr. 14.1 vlevo ukazuje klasický tvar sekvenčního obvodu, obrázek vpravo ukazuje náhradu kombinačních obvodů pamětí ROM. Rozdělení na dvě paměti ROM odpovídá funkcím původních kombinačních obvodů (realizace přechodové funkce a realizace výstupní funkce). Ve skutečném obvodu může být ROM jen jedna jediná, nebo jich může být i více – rozhodující je využití jejich datových výstupů. Vstupem paměti jsou adresové signály, udávající **současnou adresu**. To odpovídá vnitřnímu stavu sekvenčního obvodu. Budicí signály na vstupu klopných obvodů levého obrázku pak na pravém obrázku odpovídají adrese, která se objeví na výstupu registru teprve po přivedení hodinového impulzu – proto mají význam **následující adresy**.



Obr. 14.1 Realizace sekvenčního obvodu pomocí paměti ROM

Vstupem kombinačních obvodů, realizujících přechodovou funkci, jsou všechny vstupní signály plus všechny vnitřní signály. Realizace této funkce pamětí ROM je však při velkém počtu vstupních signálů velmi neúsporná (znamená velký počet adres v paměti). Je třeba si uvědomit, že u velké většiny technických problémů se zpracovává značný počet vstupních signálů, zatímco větvení na následující stavy je malé – velmi často jen stupně 2 (vstupní podmínky splněna – nesplněna). V tom případě lze následující adresu určit vždy na základě prozkoumání stavu **jen jednoho** vstupního signálu, v daném okamžiku vybraného. Proto se pro generaci následující adresy využívají principy, které významně zjednoduší obvodové řešení.

K podstatné redukci rozsahu ROM může dojít tak, že ROM bude dodávat jen hrubou informaci o následující adrese. Ta je doplněna či **modifikována** v obvodech pro generaci následující adresy (též „modifikaci adresy“) v závislosti na stavu vybraných vstupních signálů. Důležitá je skutečnost, že z velkého počtu vstupních signálů se vybírá vždy jen malá skupina, nejčastěji dokonce jen jeden. Výběr vstupních signálů je řízen z paměti ROM. Obr. 14.2 ukazuje podstatu tohoto řešení. Bližší popis obvodů je v dalším textu.



Obr. 14.2 Zjednodušená generace následující adresy

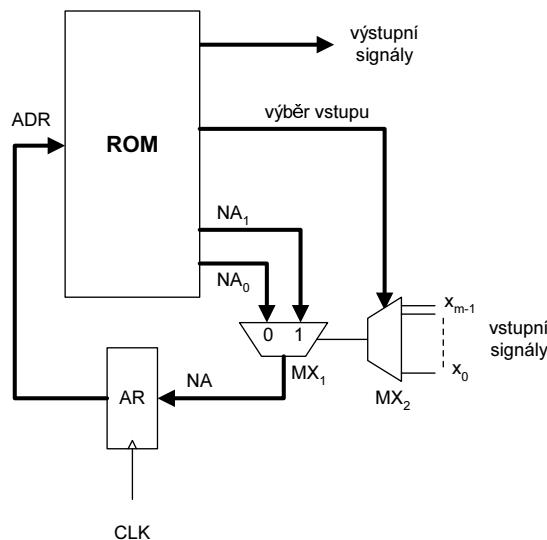
Obsah paměti ROM na jedné adrese je nazýván „**mikroinstrukce**“. Každá mikroinstrukce trvá **jeden takt** synchronizačních impulzů *CLK* a obsahuje informaci o výstupech a o následující adrese. Adresu za adresou tak proběhne celá posloupnost mikroinstrukcí, která se nazývá „**mikroprogram**“. Odtud tedy pramení název „**mikroprogramový automat**“.

Srovnáme-li schéma mikroprogramového automatu se schématem obecného sekvenčního obvodu, vidíme, že se jedná o rozdíly terminologické a technologické. Z hlediska vnějšího chování jsou si obvody rovny. Technologické rozdíly mají ovšem velmi závažný důsledek: zapojení mikroprogramového automatu může být standardní pro celou třídu úloh a jeho konkrétní chování je dáno obsahem ROM a nikoli obvodovým schématem, jako tomu bylo u pevně zapojené logiky. Přeprogramováním obsahu paměti mikroprogramu můžeme velmi

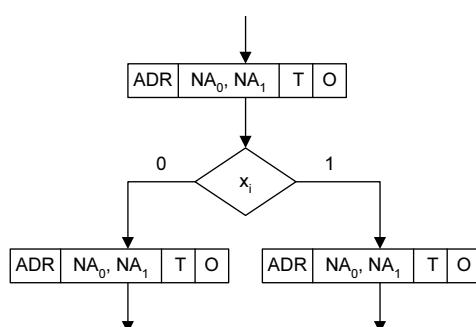
snadno a rychle realizovat změny v mikroprogramech. Mikroprogramový automat je ovšem **pomalejší** než pevně zapojená logika. Vybavovací doba ROM s obvody pro modifikaci adres je vždy delší než je doba průchodu signálu speciálně navrženým kombinačním obvodem, složeným z nejrychlejších součástek. V dalším textu si detailněji všimneme obvodů pro modifikaci adres, pro výběr podmínek přechodů z adresy na adresu, pro zpracování výstupů a dalších.

14.2 Obvody pro modifikaci adres

Nejjednodušší způsob modifikace adres ukazuje obr. 14.3. Zde se jedná o **výběr ze dvou** kompletne daných následujících adres na základě vyhodnocení stavu vybraného vstupního signálu. K prepnutí adresových signálů slouží multiplexor MX₁, k výběru vstupního signálu multiplexor MX₂. Obě možné následující adresy mohou být zcela libovolné.



Obr. 14.3 Výběr jedné ze dvou adres



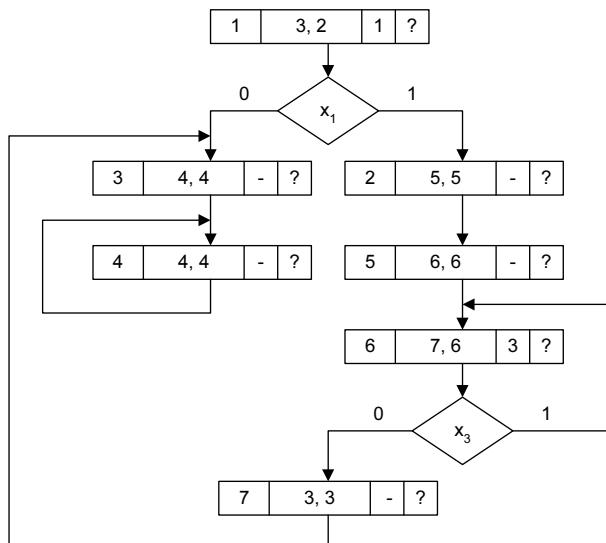
Obr. 14.4 Výsek z mikroprogramu s větvením

Při dané adrese ROM je multiplexorem MX₂ vybrán vstupní signál x_i . Je-li $x_i = 0$, bude $NA = NA_0$, při $x_i = 1$ bude $NA = NA_1$. Následující adresa je tedy závislá na vstupním signálu x_i .

Graficky lze odpovídající mikroprogram přehledně znázornit **vývojovým diagramem**. Výsek s větvením ukazuje obr. 14.4.

Mikroinstrukce je znázorněna jako blok, kde na prvném místě je současná adresa v paměti ROM, na druhém místě jsou obě následující adresy (Next Address), třetím místem T (Test) je definován výběr vstupního signálu, který řídí přechod na další mikroinstrukci, a na posledním místě O (Out) jsou definovány stavy výstupních signálů. K přechodům mezi mikroinstrukcemi dochází vždy s hranou CLK, každá mikroinstrukce tedy trvá **právě jeden takt CLK**.

Další obr. 14.5 ukazuje příklad mikroprogramu. Předpokládáme 4 vstupní signály (x_3, \dots, x_0), signál x_i je vybrán při $T = i$. Obrázek ukazuje všechny možnosti, které poskytuje zapojení mikroprogramového automatu s přepínáním dvou následujících adres. Pro stručnost budeme místo „mikroinstrukce na adrese XY“ psát „mikroinstrukce XY“.



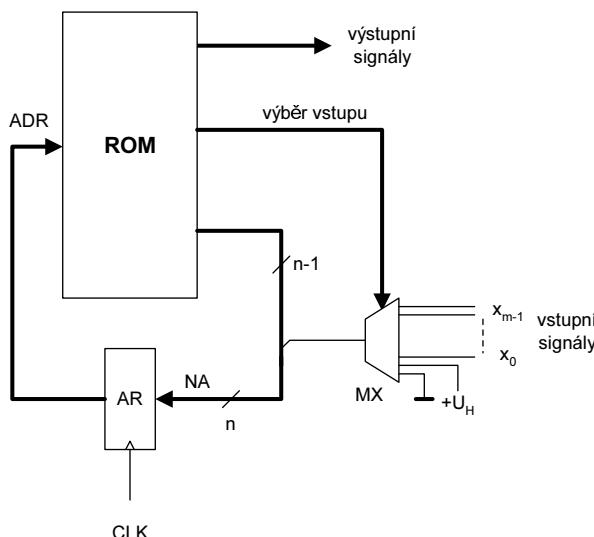
Obr. 14.5 Výběr jedné ze dvou adres

Mikroinstrukce 1 dovoluje dvojnásobné větvení s přechody na adresu 2 při splněné podmínce (tj. $x_1 = 1$) nebo na adresu 3 při nesplněné podmínce (tj. $x_1 = 0$). Výstupní signály nejsou v tomto příkladu rozebírány a proto pole „O“ jsou všude vyplněna otazníky. Na adrese 2 je naprogramován **bezpodmínečný přechod** na adresu 5 (obě následující adresy jsou shodné), obdobně i na adresách 5, 3, 4 a 7. Jelikož obě následující adresy jsou shodné, nemá žádný vstupní signál vliv na přechod na následující mikroinstrukci. Nastavení multiplexoru MX₂ je bezvýznamné a v poli T je neurčený stav, naznačený pomlčkou. Na adrese 6 je naprogramována mikroinstrukce **podmíněného setrvání** – přechod na 7 je možný jen při $x_3 = 0$, jinak se libovolně dlouho opakuje mikroinstrukce 6. Poslední možnost – **zablokování** mikroprogramu – je ukázána na adrese 4. Následující adresy jsou obě shodné se současnou

adresou a mikroprogram je tak ukončen. Dosud uvedené jednoduché schéma automatu nedovoluje opustit tento stav. Řešení bude ukázáno v dalším textu. Na konkrétní rozmístění mikroinstrukcí v ROM se nekladou žádná omezení, složitost obvodu zůstává stále stejná. Celé toto řešení je principiálně velmi jednoduché a i naprogramování paměti ROM je jednoduché. Obvodová realizace je však neúsporná. Existence dvou následujících adres o plné délce zvyšuje nároky na rozsah paměti ROM, a to většinou zbytečně. U prakticky využívaných mikroprogramů je totiž převážná většina přechodů bezpodmínečných a tudíž existence dvou následujících adres (a to shodných) ztrácí smysl.

Výše uvedený princip modifikace následující adresy lze dále rozšířit i na **vícenásobné větvení**. Obvody pro vyhodnocení podmínek přechodu pak dodávají vícebitový výstup – vektor – který prostřednictvím vícevstupového multiplexoru vybírá jednu z několika následujících adres. Tento způsob se však jeví z hlediska rozsahu obvodů jako neekonomický.

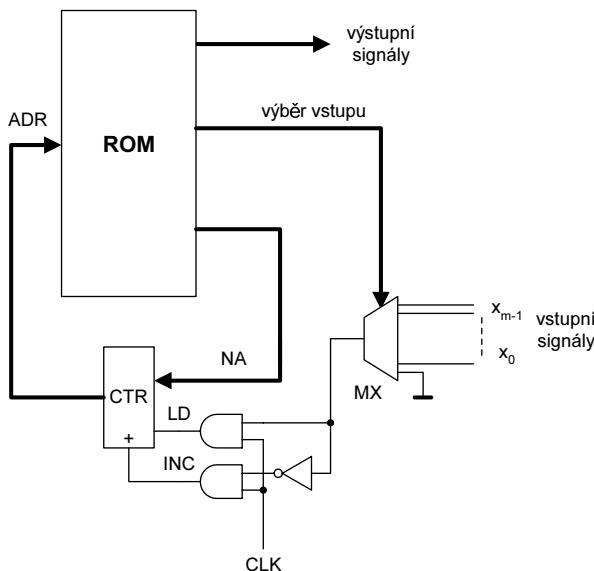
Další metodou modifikace adresy je **doplňování adresových bitů**. Nejjednodušší případ s doplňováním nejnižšího bitu ukazuje obr. 14.6. Následující adresa je z paměti ROM dodávána bez nejnižšího bitu, který je doplňován na 0 nebo 1 v závislosti na vyhodnocení vstupních signálů. Mikroinstrukce, následující po rozvětvení, jsou tedy v ROM vždy rozmístěny na sousedních adresách (lišících se v jedné adresové proměnné).



Obr. 14.6 Doplňování adresových bitů

Multiplexor pro výběr vstupů má dva vstupy navíc a lze tak jako vstupní signál vybrat i konstantu 0 či 1. Toto opatření umožňuje bezpodmínečné přechody mezi mikroinstrukcemi. Jestliže se vybere jeden z těchto vstupů, je poslední bit adresy vždy doplněn jen jedním způsobem. Lze tak volit sudou nebo lichou následující adresu. Uvedený způsob modifikace je z obvodového hlediska úsporný, v obsazování adres v paměti ROM však je menší volnost. Princip doplňování bitů lze jednoduše využít i pro **vícenásobné větvení**. Obvody pro vyhodnocení vstupních signálů pak dodávají ne jeden, ale **několik** posledních bitů následující adresy.

Třetí typ modifikace adresy využívá **přednastaviteľný čítač**. Jednoduché obvody na obr. 14.7 propouštějí na základě vyhodnocení vstupních signálů hodinový impulz CLK buď na počítací vstup čítače INC , nebo na jeho vstup pro vložení dat (přednastavovací povel LD).

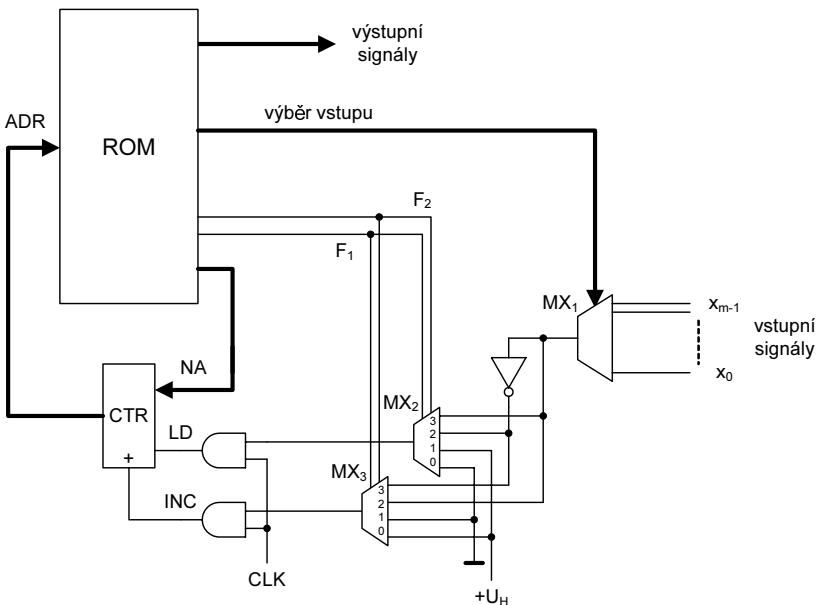


Obr. 14.7 Následující adresa generovaná čítačem

Čítač dodává adresu pro paměť ROM – adresový registr tedy **odpadá**. Následující adresa je získána buď inkrementací současné adresy (tj. jejím zvýšením o jedničku), nebo je dodána z paměti ROM. Mikroprogram tak připomíná strukturu běžného programu, kdy instrukce probíhají buď **postupně**, nebo se provede **skok**. Bezpodmínečný přechod podle obrázku nastává při nastavení multiplexoru na uzemněný vstup – to vede vždy k inkrementaci. Obvody pro ovládání čítače mohou být dále zdokonaleny a jejich činnost může být řízena zvláštěními výstupy paměti mikroprogramu podle obr. 14.8.

Rízení funkcí čítače zprostředkují signály F_2 , F_1 . Při nastavení multiplexorů MX_2 a MX_3 na vstupy 0 je čítač vždy inkrementován, při nastavení na vstupy 1 je vždy přednastaven. Při nastavení na vstupy 2 dojde k inkrementaci, pokud má vybraný vstupní signál hodnotu 1 a ke skoku při jeho hodnotě 0. Při nastavení na vstupy 3 je tomu obráceně.

S tímto složitějším, ale univerzálním uspořádáním kontrastuje další možnost aplikace čítače. Jedná se o velmi zjednodušenou verzi modifikace adresy, která nedovoluje větvení mikroprogramu. Je možné jen podmíněné setrvání mikroinstrukce nebo bezpodmínečný přechod. K tomu postačí **jen inkrementace** čítače, obvody pro přednastavení odpadají. Ačkoliv nemožnost větvení je podstatným ochuzením funkce automatu, existuje velký počet úloh, ve kterých toto zjednodušené chování postačí – jsou to úlohy typu „prováděj danou operaci stále, dokud se nesplní požadovaná podmínka“.



Obr. 14.8 Univerzální ovládání čítače

14.3 Obvody pro vyhodnocení podmínek přechodu

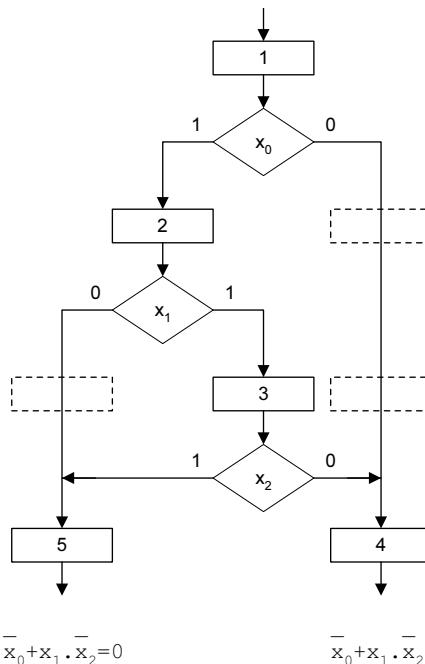
Dosud byl k vyhodnocení podmínek přechodu využíván multiplexor vstupních signálů. Jako podmínu přechodu je tak snadno možné vybrat jeden ze vstupních signálů. Pokud jako podmínka přechodu vystupuje nějaká **kombinace hodnot** vstupních signálů, vyjádřená např. logickým výrazem, je možné postupovat dvěma cestami:

- V několika po sobě následujících mikroinstrukcích **postupně vyhodnocovat** jednotlivé proměnné výrazu.
- Logický výraz **předem zpracovat** v předřazeném kombinačním obvodu a teprve jeho výstupní signál vyhodnotit.

Prvý případ je zřejmě velmi univerzální a nevyžaduje žádné dodatečné obvody – vše se děje v rámci sestavení mikroprogramu. Nevýhodou je ale jeho **pomalost**, neboť vyhodnocení celého výrazu vyžaduje několik mikroinstrukcí. Příklad vyhodnocení výrazu $\bar{x}_0 + x_1 \cdot \bar{x}_2$ ukazuje obr. 14.9.

V mikroinstrukci 1 je vyhodnocena proměnná x_0 , při hodnotě 0 nabývá výraz hodnoty 1 a vyhodnocení je ukončeno – pokračuje se mikroinstrukcí 4. Při $x_0 = 1$ bude záležet na proměnných x_1 a x_2 , ty jsou postupně též vyhodnoceny. Každá z cest vede přes jiný počet mikroinstrukcí, což může být někdy nežádoucí. K vyrovnání délek cest lze vkládat dodatečné mikroinstrukce, znázorněné v obrázku čárkováně. Výstupní signály nejsou v obrázku vyznačeny. Během celého vyhodnocení musí být konstantní, počínaje mikroinstrukcí 1, a to včetně případných vkládaných mikroinstrukcí. Teprve v mikroinstrukcích 4 a 5 mohou nabývat nových hodnot. Problémy mohou způsobit změny vstupních signálů během celého

vyhodnocení. Předpokládejme, že na začátku je stav $x_0 = 1$, $x_1 = 1$, $x_2 = 0$. Cesta vede přes mikroinstrukce 1, 2, 3, 4. Jestliže ale během mikroinstrukce 3 se x_1 změní z hodnoty 1 na 0, měl by výraz $\bar{x}_0 + x_1 \cdot \bar{x}_2$ mít hodnotu 0 – v mikroinstrukci 4 se však předpokládá hodnota 1. Chyba je v tom, že při vyhodnocení výrazu měly být všechny proměnné vyhodnoceny ve stejném okamžiku, zatím co mikroprogram je vyhodnocuje postupně.

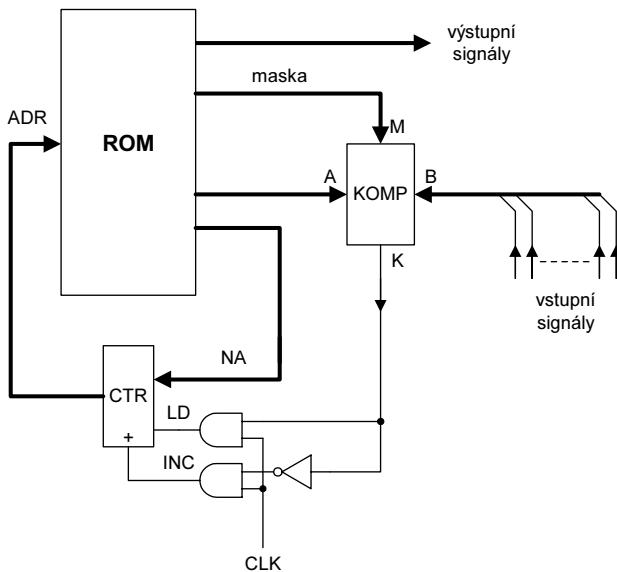


Obr. 14.9 Postupné vyhodnocení jednotlivých vstupních proměnných

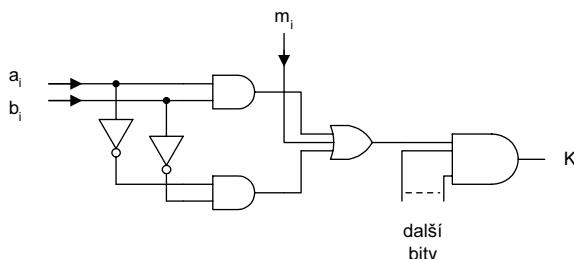
Druhá možnost pro vyhodnocení kombinace hodnot vstupních signálů je ukázána na obr. 14.10 v nejuniverzálnější variantě. Základem je **komparátor s maskováním**. Běžný komparátor porovnává dvě binární čísla o shodném počtu bitů a na svém výstupu (K) signalizuje stavem 1 shodu ve všech bitech. Tím lze zjistit, zda kombinace hodnot všech vstupních signálů se shoduje s požadovanou kombinací. Jsou však vždy uvažovány všechny signály a nelze tedy do kombinace vybírat jen některé z nich. Tuto možnost dává až komparátor s maskováním. Pro každý bit porovnávaných čísel existuje jeden bit masky, který umožnuje daný bit čísla vyřadit jako bezvýznamný a tudíž vyhodnocovat kombinace hodnot jen vybraných vstupních signálů.

Tak např. pro 4 vstupní signály x_3, x_2, x_1, x_0 by bez maskování při $A = 0010$ bylo $K = 1$ při kombinaci hodnot $x_3 = 0, x_2 = 0, x_1 = 1$ a $x_0 = 0$. Po zavedení maskování a např. čísle $M = 0100$ (předpokládejme, že stavem 1 je daný signál vyřazen) by bylo $K = 1$ při kombinaci hodnot $x_3 = 0, x_2$ libovolné, $x_1 = 1$ a $x_0 = 0$.

Jedno z možných řešení komparátoru s maskováním ukazuje obr. 14.11. Schéma odpovídá výrazu $\prod_i (a_i b_i + \bar{a}_i \bar{b}_i + m_i)$. Při $m_i = 1$ nezáleží na hodnotě a_i a b_i .



Obr. 14.10 Současné vyhodnocení několika vstupních proměnných



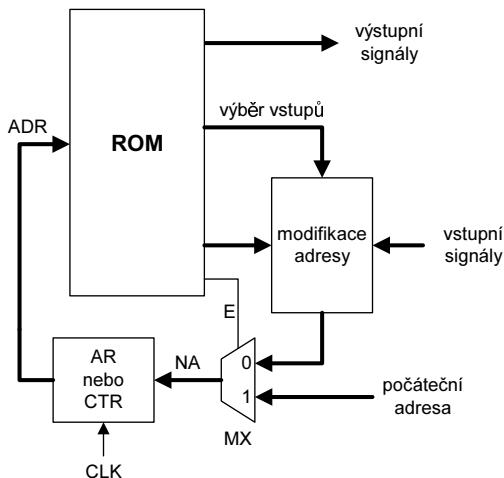
Obr. 14.11 Komparátor s maskováním

Komparátor s maskováním je univerzální řešení, ne vždy nutné. Mnohdy postačuje multiplexor, kterému jsou předřazeny kombinační obvody k předzpracování skupin vstupních signálů.

14.4 Nastavení počáteční adresy

Po zapnutí napájecího zdroje je adresový registr (nebo čítač) v náhodném stavu. Je možné jej vynulovat (tj. nastavit počáteční stav 0). V převážné většině případů však ROM obsahuje větší počet mikroprogramů, z nichž každý má svoji počáteční adresu. Pak je možné vyvolat požadovaný mikroprogram tak, že je do ROM vnučena z vnějších obvodů počáteční adresa mikroprogramu. Po prvé mikroinstrukci je další adresování již ponecháno mikroprogramu. Na obr. 14.12 je vyznačen adresový multiplexor, který je téměř po celý mikroprogram přepnuto na výstup obvodu pro modifikaci adresy. Přepnutí multiplexoru si řídí sám mikroprogram signálem E . V mikroinstrukci, která generuje $E = 1$, je adresa dána vnějšími obvody

a od této adresy se rozběhne nový mikroprogram. Během něj je signál $E = 0$, opět s výjimkou poslední mikroinstrukce.



Obr. 14.12 Zavedení počáteční adresy mikroprogramu

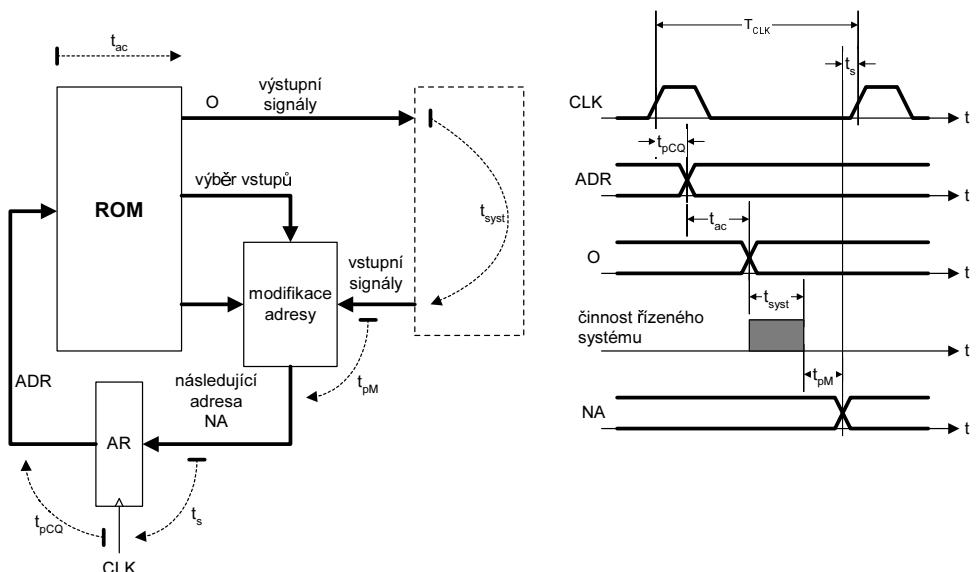
Na mikroprogram můžeme pohlížet jako na nejdetailejší popis ovládání nějakého systému. Každý z mikroprogramů vytváří sekvenci **ovládacích signálů** řídících činnost dalších obvodů. Např. mikroprogramový řadič je součástí mikroprocesoru a řídí všechny vnitřní obvody po dobu jedné instrukce, která může být velmi jednoduchá (přesuny obsahu registrů), nebo i velmi složitá (násobení, dělení). Počáteční adresy v paměti mikroprogramu pak jsou jednoznačně určeny kódem instrukce. Instrukcemi jsou tedy vyvolávány jednotlivé mikroprogramy.

14.5 Časování signálů mikroprogramového automatu

Obr. 14.13 ukazuje bloky mikroprogramového automatu podstatné pro rozbor časování. Výstupní signály slouží k ovládání řízeného systému, vstupní signály informují o stavu řízeného systému. Adresa pro ROM je dodávána z adresového registru nebo čítače. Po hraně CLK , se zpožděním t_{pCQ} , se změní výstupy adresového registru (případně čítače, je-li na něm založena tvorba nové adresy). Po uplynutí vybavovací doby t_{ac} paměti ROM jsou vytvořeny výstupní signály O . Ty působí v řízeném systému, který po době t_{syst} signalizuje změnu svého stavu (typicky je to dokončení operace). Po průchodu obvodů pro modifikaci adresy se zpožděním t_{pM} se objeví na vstupech registru nová adresa. Po uplynutí nutné doby předstihu t_s lze přivést nový hodinový impulz. Minimální perioda hodinových impulzů T_{CLKmin} je dána:

$$T_{CLKmin} = t_{pCQ} + t_{ac} + t_{syst} + t_{pM} + t_s$$

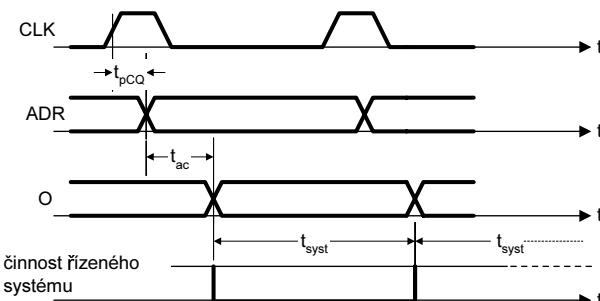
Současně je z časového diagramu zřejmé, že vstupní signály, které ovlivňují vytvoření následující adresy, se smí změnit **nejpozději** v okamžiku $t_{pM} + t_s$ před hranou CLK .



Obr. 14.13 Časování signálů v mikroprogramovém automatu

Mikroprogramový automat a řízený systém tvoří jeden funkční celek. Z celé periody T_{CLK} připadá na činnost systému jen doba t_{syst} . Tato doba může být relativně velmi krátká, pokud se jedná o velmi rychlý (tj. elektronický) řízený systém. Pak zbylé doby do T_{CLK} vlastně řízený systém zdržuje. Z těchto dob je nejdélší vybavovací doba paměti ROM.

Celou činnost lze zrychlit v taktech, ve kterých dochází k bezpodmínečným přechodům mezi mikroinstrukcemi (bez větvění). Pak se vytvoření adresy následující mikroinstrukce neváže na ukončení činnosti řízeného systému – signály z něj totiž nejsou vůbec vyhodnocovány. Obr. 14.14 ukazuje časový diagram.



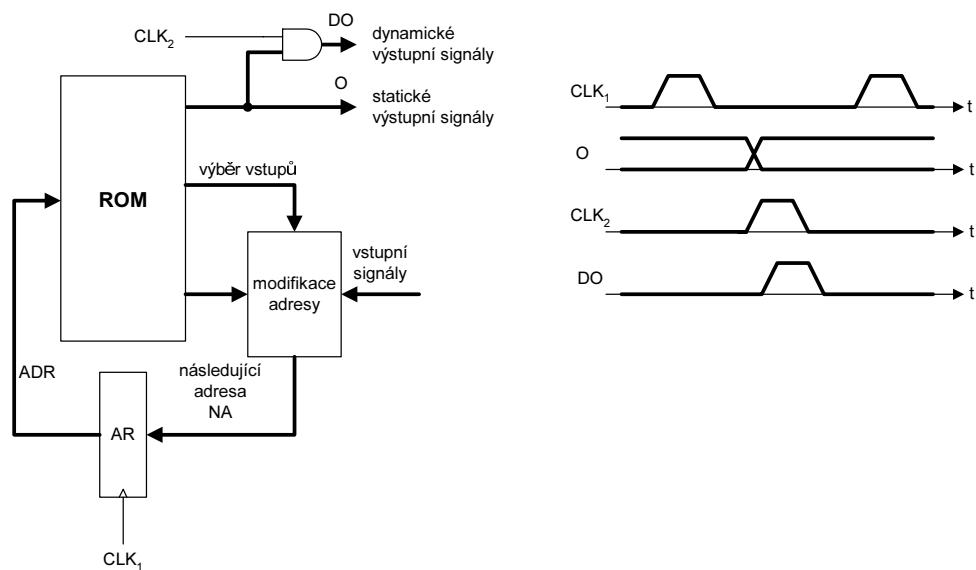
Obr. 14.14 Časování signálů při bezpodmínečných přechodech

Je zřejmé, že t_{syst} se nyní rovná periodě hodinových impulzů T_{CLK} a proto lze kmitočet hodinových impulzů zvýšit. To ovšem podle předpokladu platí jen v taktech bez vyhodnocování vstupů. Těch je sice převážná většina, větvení mikroprogramu se však nelze zcela vynout. Pokud je třeba provádět vyhodnocení vstupních signálů, není na ně dostatek času.

Jedno z řešení, a to velmi jednoduché, spočívá ve vložení **nadbytečné mikroinstrukce** před mikroinstrukcí s větvením. Výstupní signály je třeba u vložené mikroinstrukce naprogramovat shodně s mikroinstrukcí předcházející. Tím se získá čas (1 takt CLK) na ustálení vstupních signálů, takže jejich vyhodnocení v mikroinstrukci s větvením je již možné.

14.6 Dynamické výstupy

Dosud uvažované výstupy byly **statické** – trvaly celý takt T_{CLK} . Vedle těchto výstupů bývají zapotřebí i výstupy **dynamické**, které mají tvar krátkého impulzu, vyskytujícího se jedenkrát během taktu T_{CLK} . Takovéto impulzy jsou využívány např. jako zápisové impulzu pro registry uvnitř řízeného systému. Jejich vytvoření ilustruje zjednodušený obr. 14.15. Generátor hodinových impulzů zde dodává dvě fáze – CLK_1 a CLK_2 . Prvá fáze CLK_1 je použita v adresovém registru, případně čítači. Až po ustálení výstupů ROM se vytváří impulz CLK_2 a vznikne impulz DO (Dynamic Output).

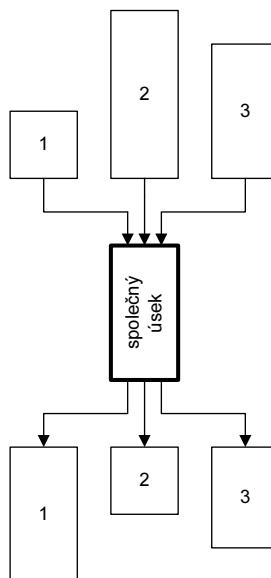


Obr. 14.15 Vytvoření dynamických výstupů

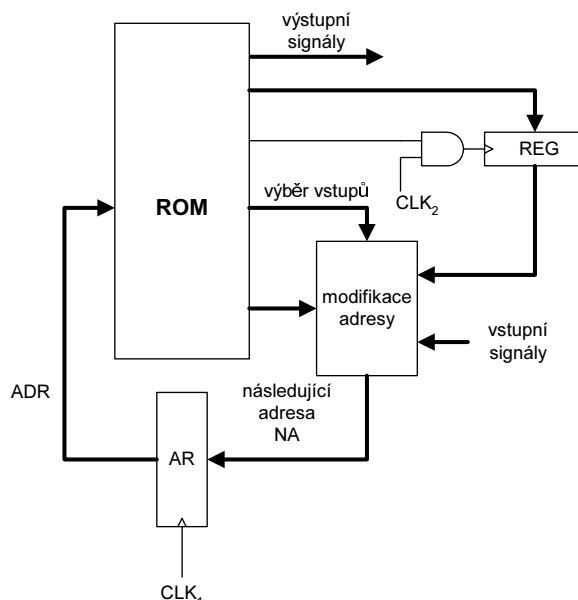
14.7 Pomocné obvody

Velmi často se u jednotlivých mikroprogramů v paměti ROM vyskytují **shodné úseky**. Je samozřejmě možné tyto úseky ponechat v patřičných mikroprogramech, které spolu nijak

nesouvisejí. To se však může ukázat jako neúsporné, zvláště pokud jsou společné úseky dlouhé. Pokud by se totiž společný úsek využil **vícenásobně**, zmenší se počet obsazených adres paměti ROM – viz obr. 14.16 se třemi mikroprogramy, označenými 1, 2, 3.



Obr. 14.16 Shodné úseky v několika mikroprogramech



Obr. 14.17 Zapojení pomocného registru

Na konci společného úseku je třeba rozhodnout o pokračování. Tato informace se musí získat ještě před vstupem do společného úseku a během něho se musí uchovat. To lze jednoduše řešit **pomocným registrem**, do kterého lze v kterémkoliv mikroinstrukci zapsat data. Výstupy registru lze vyhodnotit jako běžné vstupní signály a podle jejich stavu rozhodnout o větvení mikroprogramu – viz *obr. 14.17*. Takto dočasně zapamatovaná data lze využít zcela univerzálně.

Jiným užitečným obvodem může být **pomocný čítač** pro odpočítání počtu průchodu mikroprogramu smyčkou. Do čítače s vloží počáteční hodnota obdobně jako do pomocného registru. Čítač počítá dolů impulzy na jednom výstupu *DO* a ve zvolené mikroinstrukci se zjišťuje, zda je již ve stavu 0. Podle toho se mikroprogram vrací ve smyčce anebo pokračuje.

14.8 Použití mikroprogramového automatu

Po obvodové stránce představuje mikroprogramový automat velmi jednoduché řešení sekvenční logiky. Jednou navržené obvody mohou být využity pro celou řadu aplikací prostou **změnou obsahu** paměti. To je podstatnou výhodou proti klasicky řešenému sekvenčnímu obvodu (s pevně zapojenou logikou), kdy každá změna vyžaduje jeho totální přepracování.

Mikroprogramový automat se velmi dobře hodí pro **logické řízení** systémů. Logické řízení je zapotřebí velmi často – např. se může jednat o řízení různých strojů (zapínání a vypínání pohonů, zpracování signálů z koncových spínačů a senzorů, apod.), nebo o řízení elektronických systémů – např. **řadič** v procesoru je velmi často realizován mikroprogramovým automatem. Jistou výhodou je též snadné uvádění do chodu (oživování). Hodinové impulzy mohou být libovolně zpomaleny, případně i generovány po jednom. K prověření činnosti pak může stačit jen logická sonda.

Nevýhodou je **pomalejší činnost** proti pevně zapojené sekvenční logice. Hlavním důvodem zpomalení je vybavovací doba paměti ROM a všeobecně delší cesty signálů.

426410

A large, stylized number '15' is centered on the page. The '1' is a vertical line with a small horizontal stroke at the top. The '5' is a thick, rounded vertical line with a horizontal stroke at the top and a curved bottom. The entire graphic is composed of thin gray lines on a white background.

NÁVRH ROZSÁHLÝCH SYSTÉMŮ

Při práci s číslicovými systémy je třeba zvládnout dvě hlavní úlohy – specifikace systému a jeho následný návrh. Specifikace systému znamená popis jeho činnosti bez ohledu na to, jakými prostředky je požadovaná činnost vykonávána. Specifikace systému musí být úplná. Velmi častou chybou je nedostatečná specifikace, kdy se neuvažovaly všechny možné situace, které ale v reálném světě mohou nastat. Činnost systému v nepředpokládaných situacích pak může být zcela nežádoucí. Druhou chybou je zbytečně omezující specifikace, která naopak znemožňuje budoucí širší využití systému. Existuje řada možností, jak systém specifikovat:

- Formálním jazykem pro popis a simulaci systémů (např. VHDL)
- Běžným programovacím jazykem (např. C)
- Graficky (vývojové diagramy, stavové grafy)
- Tabulkami vstupních a výstupních hodnot

Se stále rostoucí složitostí integrovaných obvodů nabývá první možnost (např. jazyk VHDL) stále většího významu.

Pro rozsáhlé systémy se používá **hierarchický popis**, odpovídající hierarchickému uspořádání systému. Systém se dělí na subsystémy, subsystém se dělí na funkční bloky, ty se dále dělí na jednotlivé součástky. Jako nejjednodušší součástka je chápán logický člen, realizující jednu z elementárních logických funkcí – OR, AND, NOT a XOR. Poslední (nejnižší) úroveň elementárních elektronických součástek a jejich propojení – tzv. „fyzická úroveň“ – zde není uvažována. Je výslově záležitostí výrobce integrovaných obvodů a jeho technologických postupů.

Práce se standardními funkčními bloky, ať již kombinačními (dekodéry, multiplexory apod.) či sekvenčními (klopné obvody, čitače, registry apod.) podstatně usnadňuje návrh. Počítáčové návrhové systémy takovouto práci umožňují a nabízejí knihovny standardních funkcí. Funkční bloky mohou být i složitějšími celky, vhodně definované pro typickou skladbu systému.

Takovéto stavebnicové či modulární řešení umožňuje při návrhu postupovat **shora dolů** (angl. *top-down*) od rozdelení na subsystémy, které lze dále dělit na moduly (funkční bloky). Mnohdy lze jednotlivé moduly zakoupit jako hotové celky a tím etapu návrhu zkrátit. Je ovšem nutné mít velmi podrobné informace o funkci modulu a ujistit se, že vyhovuje funkci určené při postupu shora dolů. Jinými slovy – k určené funkci je třeba nalézt vhodný modul. Nevýhodou tohoto postupu je to, že optimální návrh na vyšších úrovních může zkomplikovat návrh na nižších úrovních (někdy jej dokonce znemožní).

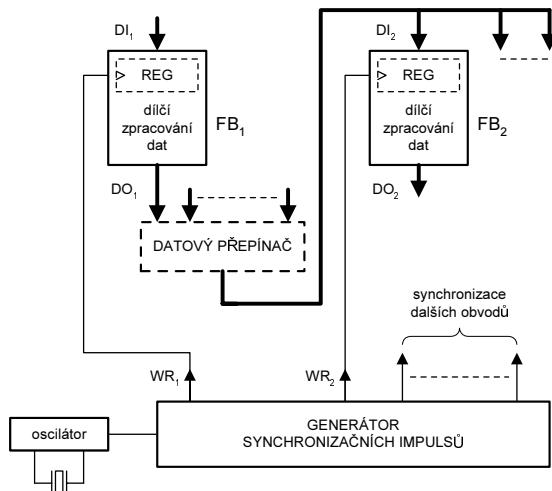
Opačný postup návrhu se nazývá **zdola nahoru** (angl. *bottom-up*). V případě postupu zdola nahoru se vychází z existujících součástek a modulů, které se propojí tak, aby bylo splněno zadání. Systém je tedy navrhován od nejnižší úrovni nahoru. Nevýhodou je to, že nelze předem garantovat, že optimální návrh na nižších úrovni nepovede nakonec k systému, jehož činnost se odchyluje od specifikace. Pak je nutné se vrátit a návrh opravit.

Nevýhody obou postupů lze potlačit **kombinovaným** postupem, kdy se vychází z principu „shora dolů“, ale současně se kontroluje, zda zvolená varianta rozdelení na vyšší úrovni umožňuje vhodnou implementaci nižší úrovně. Pokud ne, zkusi se jiná varianta. Úloha je zřejmě nejednoznačná, většinou lze systém navrhnout v několika variantách, z nichž všechny splňují zadání. Výběr nejvhodnější varianty pak bere v úvahu další kritéria, jako je cena, opravitelnost, možnost snadné modifikace v budoucnu, apod.

Příkladem hierarchického řešení systému může být počítač. Je zásadně dělen na subsystémy – procesor, paměť, periferní obvody, napájení. Paměť se dělí na moduly (vkládané do konektorů dle potřeby). Na modulech jsou součástky (integrované paměťové obvody a další podpůrné obvody). Na součástkách jsou tranzistory jako nejnižší úroveň hierarchie. Při návrhu se vyjde z nejvyšší úrovně. Některé subsystémy (procesor, paměť) se zajisté převezmou jako hotové moduly, jejichž vlastnosti musí být respektovány. Některé podsystémy (sousta-vy periferních obvodů a obvody napájení) však budou vyvíjeny dle speciálních požadavků na oblast použití počítače.

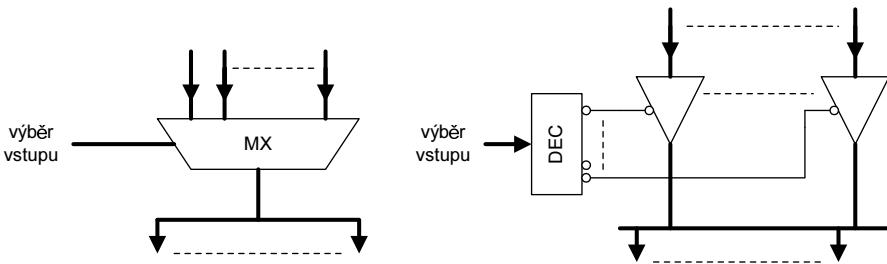
15.1 Synchronizace vnitřních obvodů systému

Pro vzájemnou spolupráci subsystémů a funkčních bloků je kritické časování vnitřních signálů. Data na vstupech systému jsou jednotlivými funkčními bloky postupně zpracovávána a na konec se dostanou na výstupy systému. Každý funkční blok v systému potřebuje určitou dobu na svoji činnost. Předávání dat uvnitř systému tedy není možné v libovolných okamžicích, ale musí být podřízeno jistému rádu. Tento řád je docílen synchronizací ze **společného** zdroje přesně časovaných impulzů – viz obr. 15.1.



Obr. 15.1 Synchronizace centrálním zdrojem synchronizačních impulzů

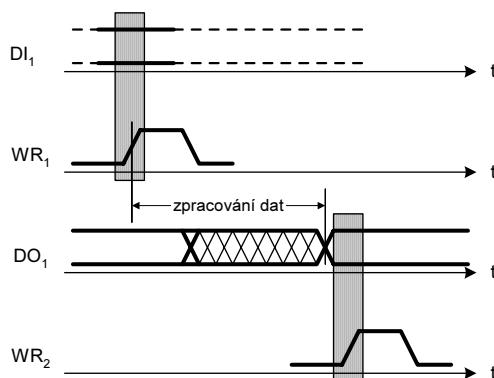
Funkční bloky FB_1, FB_2, \dots , jsou tomuto řízení přizpůsobeny zařazením vstupních registrů. Vložení nových dat na vstup funkčního bloku je řízeno dodáním zápisového impulzu (WR). V obrázku jsou pro jednoduchost znázorněny jen dva funkční bloky – ve skutečném systému jich může být větší počet. Každý z nich provádí částečné zpracování dat a svoje výstupní data předává dalším funkčním blokům. Vzniká tak **datová cesta**, podél které jsou data postupně zpracovávána. Rídicí (synchronizační) impulzy garantují předávání dat ve správných okamžicích. Přesné časování je odvozeno od oscilátoru řízeného krystalem. Pro větší univerzalitu lze mezi funkční bloky zařadit datové přepínače a dle potřeby měnit cesty dat. Dvě alternativy ukazuje obr. 15.2.



Obr. 15.2 Přepínání datových cest

Obě varianty jsou funkčně ekvivalentní, rozdíl je v konstrukčním usporádání. V obou případech lze data z jednoho vybraného zdroje přesunout současně do několika cílových míst. Vždy však lze vybrat jen jeden zdroj dat – **současné** přesuny z několika různých zdrojů nejsou samozřejmě možné. Toto omezení je důležité.

Časování signálů z obr. 15.1 ukazuje obr. 15.3.



Obr. 15.3 Časování přesunu dat mezi funkčními bloky

Jelikož jsou předpokládány vstupní registry řízené **hranou**, musí být vstupní data stabilní před i za aktivní hranou zápisového impulzu – viz zvýrazněnou oblast v obrázku. Pokud jsou v systému přítomny datové přepínače, musí se jejich zpoždění započítat do doby na zpracování dat.

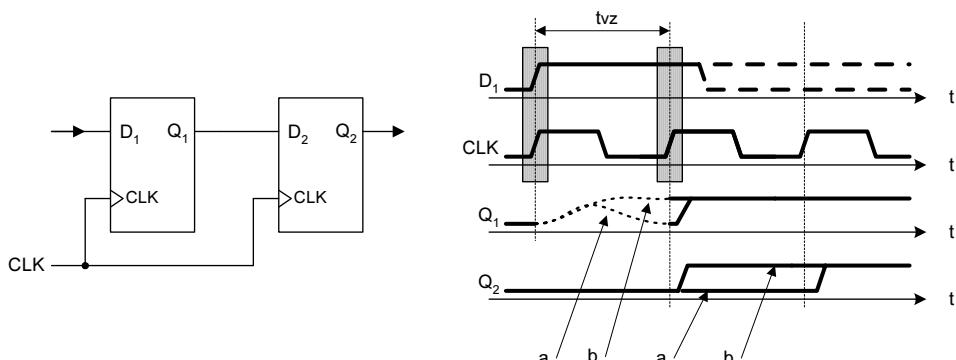
Doba na zpracování dat nemusí být u jednoho a téhož funkčního bloku konstantní – může se měnit např. v závislosti na typu a hodnotě dat. Vezměme jako příklad i tak jednoduchou operaci, jako je stanovení absolutní hodnoty čísla. Je-li číslo kladné, nemění se. Je-li záporné, musí se změnit znaménko. U čísla v dvojkovém doplňkovém kódu je ke změně znaménka nutno znegovat všechny bity čísla a následně přičíst jedničku. Rozdíl v době zpracování je zřejmý.

Při návrhu časování synchronizačních impulzů lze respektovat nejpříznivější situaci, tj. nejdélší doby zpracování dat. Systém bude pracovat spolehlivě, mnohdy však zbytečně pomalu. Zrychlení je možné prostřednictvím **variabilního časování**. Časování se přizpůsobuje stavu jednotlivých funkčních bloků. Informace o stavu je zavedena do generátoru řídicích

impulzů, který na jejím základě mění časování. Nejjednodušší metodou je prosté hlášení funkčních bloků o dokončení své dílčí operace. Generátor pak jednoduše pozdrží vydávání dalších impulzů, dokud probíhající operace neskončila.

15.2 Časování vstupních signálů

Vstupní signály synchronního systému se nemohou měnit v libovolných okamžicích. V kapitole o sekvenčních obvodech byly odvozeny doby předstihu a přesahu vstupních signálů proti hraně synchronizačního impulzu. U rozsáhlého systému, složeného z řady funkčních bloků, se otázka časování vstupních signálů samozřejmě týká jen prvého bloku podél cesty. Je podstatný rozdíl v tom, jak jsou vstupní signály interpretovány. V prvném (méně náročném) případě jsou jednotlivé signály vzájemně nezávislé, jejich kombinace hodnot nemají význam čísla, a jejich změny spolu nijak časově ani významově nesouvisejí. Typickým příkladem jsou např. dvojhodnotové signály z různých koncových spínačů elektromechanického systému, z ovládacích tlačítek, apod. K jejich změně může dojít i během náběžné hrany synchronizačního impulzu v prvním funkčním bloku. Situaci ukazuje obr. 15.4.



Obr. 15.4 Dvojitý vzorkovač

Při nevhodném časování se může první klopny obvod (Q₁) dostat do metastabilního stavu, jehož důsledkem může být náhodný stav KO – viz čárkováné průběhy v obrázku. Na výstupu prvého KO je tedy signál zcela neregulérní a nemůže být v této podobě zpracován navazujícími obvody. Proto je přidán další klopny obvod, takže vznikl dvojstupňový posuvný registr. Pokud se vstupní signál změnil např. ze stavu 0 do 1 (jak je v obrázku vyznačeno), bude při dostatečně dlouhé periodě synchronizačních impulzů CLK na výstupu Q₁ již ustálený (i když možná náhodný) stav při náběžné hraně dalšího impulzu CLK. Pokud došlo ke správnému překlopení Q₁ do stavu 1, překlopí se teď Q₂ rovněž do stavu 1 (průběh b). Pokud došlo k nesprávnému překlopení Q₁ do stavu 0 a pokud data na vstupu D₁ jsou ještě ustálená (ve stavu 1 podle obr. 15.4), překlopí se první KO do stavu 1 až teď, a druhý KO ještě o takt později (průběh a). Signál Q₂ má tedy vždy regulérní průběh. Náhodnost spočívá v tom, že změna na výstupu Q₂ se projeví buď po druhém impulzu CLK, nebo až po třetím. Dojde tedy k nejistotě v čase o 1 takt.

Uvedený obvod se nazývá **dvojitý vzorkovač**. Ten se zařazuje na jednotlivé vstupy signálů, které nelze synchronizovat. Podmínkou bezchybné činnosti je perioda vzorkovacích impulzů, která musí být delší než doba trvání případného metastabilního stavu. Bohužel, tuto dobu nelze přesně stanovit. Závisí na časovém odstupu mezi změnou vstupního stavu a aktivní hrany hodinového impulzu a též na strmosti jeho aktivní hrany. Žádný vzorkovač proto nemůže zaručit stoprocentní spolehlivost.

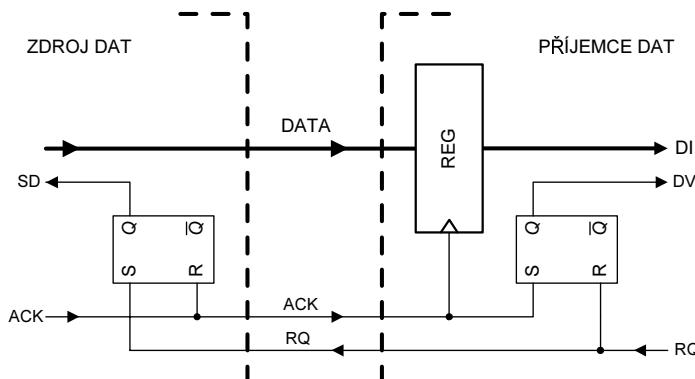
Z obr. 15.4 je dále zřejmé, že má-li být zachycena každá změna vstupního signálu (s případnou neurčitostí jednoho taktu, jak bylo vysvětleno), musí jak stav 0, tak i stav 1 na vstupu trvat nejméně jednu periodu vzorkování. Celkově tedy perioda vstupního signálu musí být delší, než dvojnásobek periody vzorkovacích impulzů – jinými slovy kmitočet vzorkování musí být nejméně dvojnásobkem kmitočtu vstupního signálu. To je zajímavý výsledek, který svědčí o široké platnosti Shannonovy věty o vzorkování.

Horší situace, než je tomu u samostatných signálů, nastává u časování vstupu vektorů – tedy **skupin signálů**, které k sobě významově patří. Jednotlivé složky vektoru mohou být např. interpretovány jako bity binárního čísla. Při změnách vektoru není zaručeno, že jeho jednotlivé složky se budou měnit absolutně současně. Na přechodnou dobu se tak mohou vyskytovat kombinace hodnot, které nepatří ani k předchozímu ustálenému stavu, ani k následujícímu ustálenému stavu. Např. při změně binárního čísla z hodnoty 0111 na 1000 (což je jen minimální změna, a to z dekadické hodnoty 7 na 8), a při neznámém pořadí změn v jednotlivých bitech, jsou možné jakékoli dočasné hodnoty (uvedeny v závorce):

0111	(1111)	1000	... nejvyšší bit se změnil jako první, ostatní až po něm
0111	(0000)	1000	... nejvyšší bit se změnil jako poslední, ostatní před ním
0111	(1011)	1000	... dva nejvyšší bity se změnily jako první
0111	(0100)	1000	... dva nejvyšší bity se změnily jako poslední
..... atd.			

Při vzorkování vektoru tak mohou vznikat velmi **významné chyby**, od kterých nepomohou dvojité vzorkovače. Východiskem je jen zavedení doplňkových signálů pro řízení přenosu dat mezi zdrojem a příjemcem.

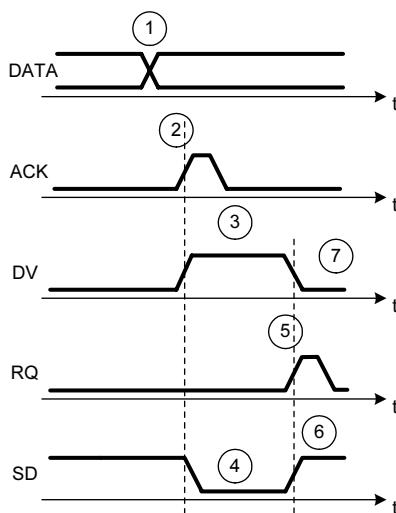
15.3 Korespondenční provoz



Obr. 15.5 Obvody a signály pro korespondenční provoz

Korespondenční provoz, též provoz dotaz–odpověď (angl. *handshake*) zavádí kromě datových signálů ještě dva signály pro řízení vzájemné spolupráce. Vyžaduje též malé rozšíření vstupních obvodů. Princip ukazuje obr. 15.5.

Zdroj dat potvrzuje platnost svých vydávaných dat impulzem *ACK* (angl. *acknowledge* – potvrzení), kterým se data zapíší do vstupního registru příjemce dat. Jelikož generace dat i impulzu *ACK* je plně v rámci zdroje dat, je garantováno správné časování zápisu do registru a nemohou vznikat problémy spojené se vzorkováním vektoru, tak jak byly výše popsány. Příjemce dat vyžaduje nová data impulzem *RQ* (angl. *request* – požadavek). Na obou stranách jsou doplněny pomocné klopné obvody, dodávající informaci o stavu přenosu. Časový diagram na obr. 15.6 ukazuje hlavní body vzájemné komunikace.



Obr. 15.6 Průběh signálů korespondenčního provozu

Zdroj dat vydá nová data (bod 1) a potvrzuje jejich platnost impulzem *ACK*. Tím se data zapíší do vstupního registru příjemce (bod 2). Současně je v příjemci dat překlopen pomocný klopný obvod v do stavu 1 a tím je signalizováno, že jsou v registru uložena nová data – úsek 3 a aktivní signál *DV* = 1 (Data Valid – data platná). Současně se překlopí i pomocný klopný obvod ve zdroji dat do stavu 0 a tím je dočasně zablokováno vydávání dalších dat – úsek 4 a neaktivní signál *SD* = 0 (Send Data – vyšli data).

Příjemce dat testuje stav klopného obvodu a zpracuje data ze vstupního registru jen při *DV* = 1. Po jejich zpracování generuje impulz *RQ* jako žádost o nová data (bod 5). Současně se tím vynuluje i pomocný klopný obvod a signálem *DV* = 0 je zakázáno další zpracování dat ze vstupního registru, dokud do něj nebudu zapísána nová data. Opakování čtení, aniž by byla do registru zapísána nová data, tedy není možné, jelikož vždy před čtením obsahu registru se testuje signál *DV*. Pokud je 0, zpracování se pozdrží.

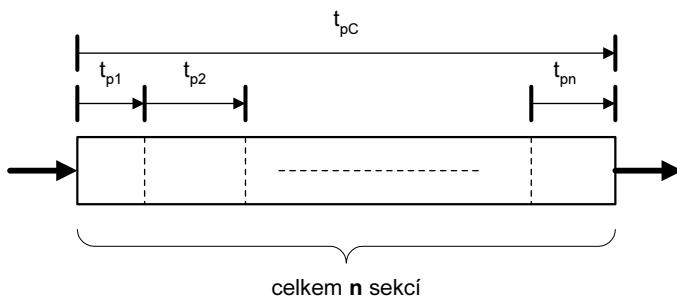
Impulz *RQ* současně překlopí pomocný obvod ve zdroji dat do stavu 1. Aktivním signálem *SD* = 1 se povolí generace nových dat (úsek 6). Nová data tedy nemohla být do registru příjemce zapísána dříve, než byl obsah registru zpracován.

Tento způsob vzájemné synchronizace, kdy se používají signály s významem „požadavek – potvrzení“, zabraňuje **chybnému čtení** dat, která se současně mění, zabraňuje **opakování starých dat**, a zabraňuje také **ztrátě** dat při jejich předčasném dodání.

V praktických realizacích jsou signály *ACK* a *RQ* mnohdy označeny jinými názvy a jsou případně negovány. Mohou mít i jiný průběh, než jaký je znázorněn na obr. 15.6. Princip vzájemného blokování a povolování operací je však zachován.

15.4 Zřetězené zpracování

Účinnou metodou pro zrychlení činnosti systému je zavedení zřetězené struktury (angl. *pipeline* – „potrubí“). Základní myšlenku ukazuje obr. 15.7. Původní systém je na vhodných místech rozdelen na dílčí sekce. Sekce si v prvém přiblížení můžeme představit jako kombinační obvody, i když metodu lze dobře aplikovat i na obvody sekvenční. V obrázku jsou vyznačena zpoždění sekcí, která obecně nejsou stejná. Řez totiž nelze udělat na kterémkoliv místě, ale je třeba respektovat hranice funkčních bloků, integrovaných obvodů, konektory, atd.

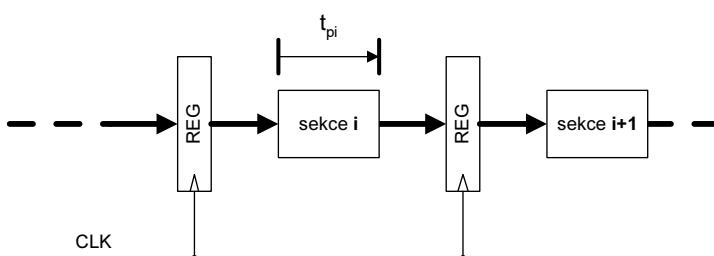


Obr. 15.7 Rozdělení systému na sekce v kaskádě

Součet zpoždění všech sekcí udává celkové zpoždění t_{pc} :

$$t_{pc} = \sum_1^n t_{pi}$$

Mezi sekce jsou vloženy vyrovávací registry. Ty slouží jednak pro odstranění případných hazardů, jednak pro krátkodobé uložení dat na výstupech sekcí – viz obr. 15.8. Do všech registrů se zapisuje společnými hodinovými impulzy.

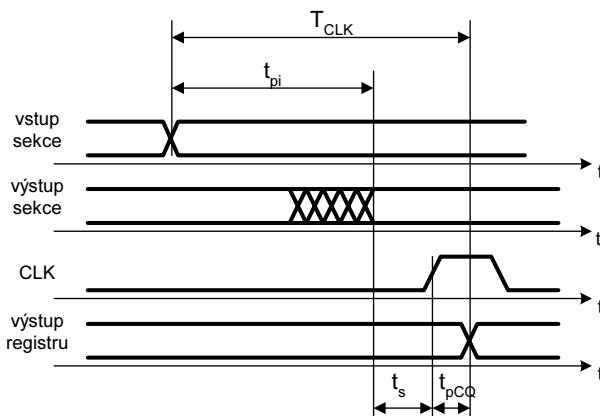


Obr. 15.8 Vložení oddělovacích registrů mezi sekce

Registry tvoří dočasnou paměť, která od sebe izoluje sekce po dobu taktu. Vstupní data, zpracovaná prvnou sekcí, jsou na konci prvého taktu uložena do prvého registru. Tím jsou dostupná pro druhou sekci, ve které jsou ve druhém taktu dále zpracována a pak uložena do druhého registru. Data tím definitivně opouštějí prvnou sekci. Vstupní data jsou takt za taktem posunována do dalších sekcí a postupně zpracovávána. Aby prvná sekce po uvolnění nezahálela, mohou na její vstup být ve druhém taktu přivedena nová data. Totéž platí i pro další sekce. Jednotlivá data, postupně s každým taktem dodávaná na vstup, tak postupují řetězcem podobně jako kapalina postupuje potrubím (odtud „pipeline“) a jsou postupně zpracovávána. Na výstupu řetězce se objevují zpracovaná data s každým taktem hodin.

Casový průběh signálů v jedné sekci ukazuje obr. 15.9. Doba průchodu signálů sekcí t_{pi} respektuje i odeznění případných falešných impulzů. Signály na vstupu registru musí být ustáleny minimálně o dobu předstihu t_s před hranou hodinového impulzu CLK. Klopné obvody registru se překlápejí po době t_{pCQ} od hrany CLK. Součet těchto tří dob určuje minimální periodu hodinových impulzů. Jednotlivé sekce však nemají shodná zpoždění, a proto perioda společných hodinových impulzů se musí určit podle sekce s největším zpožděním $t_{p\max}$:

$$T_{CLK} \geq t_{p\max} + t_s + t_{pCQ}$$



Obr. 15.9 Průběh signálů v sekci řetězce

Při maximálním kmitočtu hodinových impulzů (tedy minimálním T_{CLK}) je doba průchodu prvních dat celým řetězcem t_{DD} dáná jako:

$$t_{DD} = N \cdot t_{p\max} + N(t_s + t_{pCQ}),$$

což je doba delší, než doba zpoždění původního (nerozřezaného) systému. Zdánlivě tedy zřetězení nepřináší žádné zlepšení. Všimněme si však případu, kdy data jsou přiváděna na vstup původního (nerozřezaného) systému v pravidelných intervalech. Přivedená data jsou zpracována a po době t_{pc} se dostanou na výstup. Odtud jsou přesunuta do navazujících obvodů. Teprve teď lze přivést nová vstupní data. Tedy vstupní data mohou být přiváděna s periodou, dlouhou minimálně t_{pc} . Při zřetězeném zpracování mohou být data přiváděna

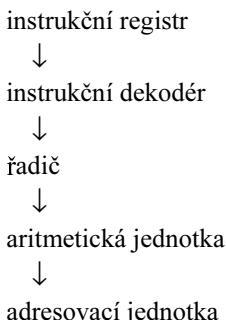
s periodou T_{CLK} . S toutéž periodou se budou měnit i data na výstupu. Při plynulém zpracovávání dat, dodávaných na vstup takt za taktem, tedy data na výstupu vycházejí s periodou T_{CLK} , což je doba mnohem kratší než doba t_{pc} u původního systému. Lze pak určit **zrychlení činnosti** vlivem zřetězení jako poměr časového odstupu mezi dvěma následujícími datovými slovy na výstupu původního (nerozřezaného) systému a na výstupu řetězce. Zrychlení a_{pl} je dáno vztahem

$$a_{pl} = \frac{\sum t_{pi}}{T_{CLK}} = \frac{\sum t_{pi}}{t_{p\max} + t_s + t_{pCQ}}$$

Jako příklad účinnosti zřetězení předpokládejme systém s pěti sekcmi o zpoždění 7 ns, 7 ns, 6 ns, 8 ns a 10 ns. Klopné obvody mají $t_s = 2$ ns, $t_{pCQ} = 4$ ns. Perioda hodinových impulzů pak vychází minimálně 16 ns ($f_{CLK} = 62,5$ MHz). Zrychlení $a_{pl} = 38/16 = 2,375$. To je jistě významný výsledek.

V ideálním případě, tj. při shodných zpoždění všech N sekcí a registrech s nulovým zpožděním, vyjde $a_{pl} = N$. Skutečnost je vždy o něco horší. Nutnou podmínkou úspěšného využití zřetězení je plynulá dodávka vstupních dat s každým taktem CLK .

Velmi často se princip zřetězení využívá u procesorů a u systémů pro číslicové zpracování signálů. U procesorů se tvoří řetězec např. takto:



Pokud jsou instrukce prováděny plynule jedna za druhou a jsou stejně dlouhé, je zřetězení velmi účinné. Pokud se však vyskytne instrukce skoku nebo volání podprogramu, je obsah řetězce nutné zcela obnovit a to pak znamená časovou ztrátu.

15.5 Systémy RTL

Systém RTL (z anglického *Register Transfer Level* – volně přeloženo „systém založený na přesunech mezi registry“) je produktem jedné z návrhových metod, která vede k důslednému rozdělení návrhu dvě dílčí úlohy:

- Návrh jednotek na zpracování dat (funkčních bloků) a příslušných datových cest.
- Návrh řídících obvodů, které generují signály pro ovládání funkčních bloků a pro přepínání datových cest.

Jednotky na zpracování dat provádějí dílčí operace nad daty a mohou to být jak kombinační, tak i sekvenční obvody. Některé operace (logické operace AND, OR, NOT apod.)

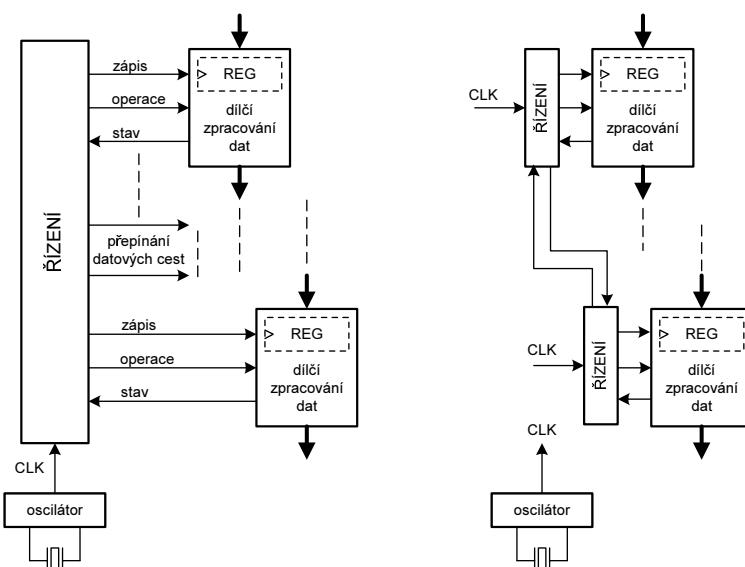
mohou být realizovány jednoduchými kombinačními obvody v rámci jednoho funkčního bloku, jiné operace (násobení, dělení apod.) vyžadují komplikované sekvenční obvody. Řídicí obvody generují zápisové impulzy do jednotlivých registrů a adresy pro přepínače datových cest. Tím určují „kdy-odkud-kam“ se přesouvají jednotlivé dílčí výsledky. U některých funkčních bloků též vybírají typ operace (např. u aritmeticko-logické jednotky v počítači).

Řídicí signály mohou být generovány v neměnném sledu, nebo častěji v závislosti na výsledcích činnosti (stavových signálech) z jednotlivých funkčních bloků. Řízení je vždy realizováno synchronními **sekvenčními** obvody, velmi často v podobě **mikrogramového** automatu.

Jak bylo ukázáno v obr. 15.1, funkční bloky jsou vybaveny registry. Přesun dat mezi dvěma registry tedy ve svém důsledku znamená dílčí operaci nad daty, neboť dat prošla funkčním blokem. Celá úloha zpracování dat, spočívající v sekvenci dílčích operací, se vlastně jeví jako sekvence přesunů mezi patřičnými registry. Odtud název „RTL systém“.

Pro návrh systému metodou RTL je charakteristické **oddělení návrhu** datových cest s funkčními bloky od návrhu řídicích obvodů. Je nutno zdůraznit, že se jedná výhradně o oddělení návrhu, nikoliv o **fyzické oddělení**. Řídicí obvody mohou být skutečně fyzicky odděleny od obvodů pro zpracování dat, tj. mohou být centralizovány, a právě tak mohou být rozděleny a po částech připojeny k funkčním blokům – to je pak decentralizované uspořádání. Vhodnost toho či onoho způsobu rozmístění je záležitostí detailů konstrukce systému.

Obr. 15.10 vlevo ukazuje centralizované uspořádání, obrázek vpravo pak uspořádání decentralizované. V případě centralizovaného uspořádání jsou jednotlivým funkčním blokům pro zpracování dat dodávány zápisové impulzy pro jejich vstupní registry, a podle konstrukce funkčních bloků případně i signály pro volbu operace. Funkční bloky mohou naopak informovat řídicí obvody o svém stavu. I přepínače datových cest (v obrázku nevyznačené – viz obr. 15.2) jsou řízeny centrálními obvody.



Obr. 15.10 Řídicí obvody centralizované (vlevo) a decentralizované (vpravo)

U decentralizovaného řízení jsou k funkčním blokům připojeny i části řídicích obvodů. V obrázku jsou vyznačeny signály, propojující navzájem jednotlivé dílčí řídicí obvody. Tyto signály jsou nutné pro vzájemnou koordinaci jednotlivých částí řídicích obvodů.

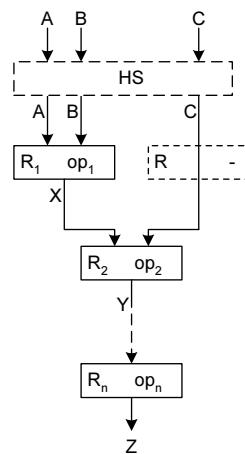
Obr. 15.10 ukazuje dva krajní případy – v praxi se bude jednat spíše o kombinaci obou přístupů.

Stavové signály, vedené z funkčních bloků do řídicích obvodů, jsou potřebné jen někdy. Je to hlavně pro informaci o těchto výsledcích operace:

- Ukončení operace – pokud doba na provedení operace je proměnlivá.
- Hlášení chyby – kontrola dat ukázala nesprávný formát, atd.
- Data mimo rozsah – data mohou mít záměrně omezený rozsah.
- Data neodpovídají nastavené operaci – dělení nulou, atd.
- Výsledek porovnání (komparace) dvou operandů: $=, >, <$.
- Případně další...

Stavové signály nejsou nutné např. u funkčních bloků vykonávajících logické operace (AND, OR, NOT, XOR, ...), kde doba zpracování je krátká a konstantní, a neexistují nesprávné hodnoty dat.

Datové cesty, neboli seřazení jednotlivých funkčních bloků, mohou být navrženy několika způsoby. V nejjednodušším případě tvoří **řetězec bez smyček** podle *obr. 15.11*. Funkční bloky jsou zde kresleny zjednodušeně – je uveden název jejich vstupního registru a operace, kterou blok provádí. Řídicí a stavové signály nejsou kresleny. Na vstupu systému mohou být zařazeny obvody pro korespondenční provoz (čárkované blok HS – „handshake“), které umožní spolupráci navrhovaného systému s nesynchronním zdrojem vstupních signálů – viz popis v odstavci 15.3.

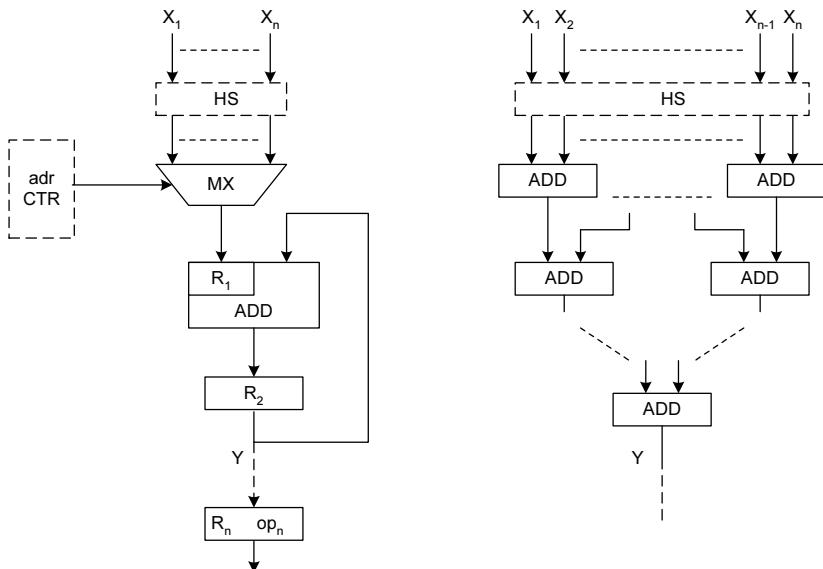


Obr. 15.11 Řetězec funkčních bloků bez smyček

Existence řetězce bez smyček, jehož jednotlivé funkční bloky mají přibližně shodná zpoždění, dává možnost využití principu zřetězeného zpracování. Je ale nutné zajistit, aby všechny vstupní signály procházely na výstup přes shodný počet registrů – jinak by např. v bloku

(R_2, op_2) se setkal signál C se signálem X , který je ale o jeden takt zpožděný, neboť prošel blokem (R_1, op_1) . Signál C tedy musí být zpožděn, což lze zařídit vložením registru do jeho cesty (čárkováný blok s registrém R). Na takovýto samotný registr lze pohlížet jako na funkční blok, nevykonávající s daty žádnou operaci.

Jiné možné uspořádání využívá některé **funkční bloky ve smyčce**. Příklad ukazuje obr. 15.12. Je třeba sečíst vstupních data z n kanálů a výsledek (aritmetického) součtu dále zpracovat. Pro data Y platí $Y = \sum_1^n X_i$.

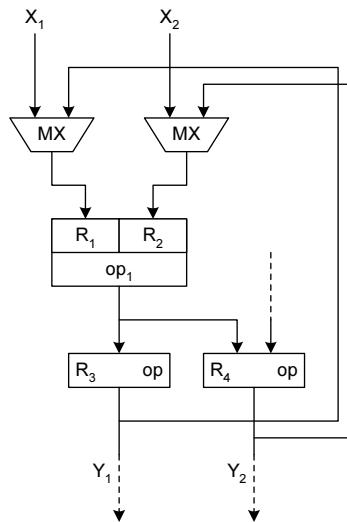


Obr. 15.12 Porovnání postupného (vlevo) a současného (vpravo) provádění operací

Postupné sečítání jednotlivých vstupních dat je ukázáno v levé části obrázku. Řídící obvody (nejsou naznačeny) budou postupně přepínat multiplexor. K tomu účelu budou muset obsahovat čítač adres, postupně inkrementovaný. V obrázku je vyznačen čárkováně (blok adrCTR). Na začátku musí být registry R_1 a R_2 vynulovány. Důležité je, že vstupní data musí zůstat konstantní až do skončení sečítání. Proto může být nutné zařadit do vstupů registry, případně obvody HS. Sčítáčka postačí jen jedna a je mnohonásobně využita, probíhá **postupné zpracování** signálů, avšak zpoždění ze vstupu na výstup je dlouhé.

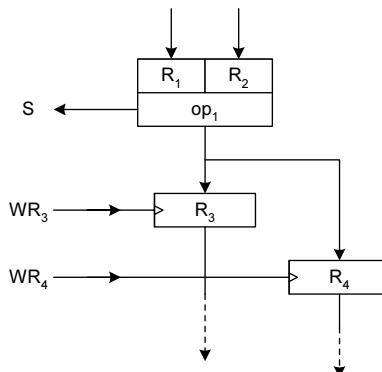
Obr. 15.12 ukazuje jinou možnost, a to **současné zpracování** všech vstupních dat kaskádou sčítáček (bez registrů), zapojených jako pyramida. Je tak zapotřebí mnohem větší množství obvodů, zpracování je ale rychlejší. Vstupní signály procházejí na výstup současně.

Výše uvedené struktury jsou sestaveny vždy pro jeden účel – datové cesty i operace jsou pevně dány. Větší variabilitu umožňuje zavedení **přepínání** datových cest a **podmínečného provádění** operací na základě stavových signálů. Na obr. 15.13 je ukázán příklad dvojího využití funkčního bloku provádějícího operaci op_1 , jednou s proměnnými X a podruhé s proměnnými Y . Všeobecně platí, že vícenásobné využití některých bloků šetří hardware, ale komplikuje nebo znemožňuje zřetězené zpracování – pro ně je totiž optimální přímá cesta dat bez smyček.



Obr. 15.13 Vícenásobné využití funkčního bloku

Obr. 15.14 ukazuje příklad uspořádání pro podmínečné provádění operací. Předpokládejme, že výsledkem operace op_1 jsou nejen výstupní data funkčního bloku, ale též stavové signály S . Např. aritmetická jednotka v počítači signalizuje přenos z nejvyššího bitu, nulovost výsledku, znaménko výsledku, atd. V závislosti na některém ze stavových signálů pak řídicí obvody generují zápisový impulz WR bud' pro registr R_3 , nebo R_4 a tím určují další cestu pro zpracování dat. Měnit podle potřeby operaci s daty lze též pomocí univerzálních funkčních bloků, jako je např. opět aritmetická jednotka v počítači.



Obr. 15.14 Větvění podle podmínek

Pro popis činnosti systému RTL byl zaveden **jazyk RTL** (zde RTL – Register Transfer Language). Z popisu v RTL lze snadno odvodit datové cesty. Přesuny dat mezi registry a dílčí operace, ke kterým přitom dojde, jsou vyjádřeny velmi jednoduše.

Pro symbolické označení přesunu dat se používá šipka „ \leftarrow “ nebo někdy dvojice znaků „ $:=$ “. Přesuny, případně s dílčími operacemi s daty, se symbolicky zapisují takto:

$R_2 \leftarrow R_1$;přesun obsahu R_1 do R_2
$R_3 \leftarrow R_1 + R_2$;přesun aritmetického součtu obsahů R_1 a R_2 do R_3
$R_3 \leftarrow R_1 - R_2$;přesun aritmetického rozdílu obsahů R_1 a R_2 do R_3
$R_3 \leftarrow R_1 * R_2$;přesun aritmetického součinu obsahů R_1 a R_2 do R_3
$R_3 \leftarrow R_1 \vee R_2$;přesun logického součtu (OR) obsahů R_1 a R_2 do R_3
$R_3 \leftarrow R_1 \wedge R_2$;přesun logického součinu (AND) obsahů R_1 a R_2 do R_3
$R_3 \leftarrow R_1 \oplus R_2$;přesun exkluzivního součtu (XOR) obsahů R_1 a R_2 do R_3

Současně probíhající přesuny (s případnými operacemi) se oddělují čárkami, operace probíhající po sobě se oddělují rádkami. Např. pro zpracování dat ze čtyř kanálů $X_1\dots X_4$ podle vztahu $Z = (X_1+X_2)*(X_3+X_4)$ lze sestavit systém, popsaný takto:

$$\begin{array}{ll} R_3 \leftarrow R_1 + R_2, & R_4 \leftarrow R_5 + R_6 \\ R_7 \leftarrow R_3 * R_4 & \end{array} \quad \begin{array}{l} ;provedou se součty dvou dvojic dat \\ ;provede se součin součtů \end{array}$$

K vyjádření podmínek pro provedení operace se používá symbolická konstrukce s dvojtečkou:

$$K : R_1 \leftarrow R_2 \quad ;je-li signál K = 1, provede se přesun, jinak ne$$

Ke složitější konstrukci podmínek se používají logické kombinace jednotlivých signálů:

$$\begin{array}{ll} \overline{K_1} \wedge K_2 : R_1 \leftarrow R_2 & ;k přesunu dojde, je-li K_1 = 0 a současně K_2 = 1 \\ K_1 \vee K_2 : R_1 \leftarrow R_2 & ;k přesunu dojde, je-li K_1 = 1 nebo K_2 = 1 \\ K_1 \oplus \overline{K_2} : R_1 \leftarrow R_2 & ;k přesunu dojde, je-li K_1 \oplus \overline{K_2} = 1 \\ \dots atd. \dots & \end{array}$$

Pro konstrukci typu „if-elseif...-else“ lze zapsat několik podmínečných operací do jedné řádky:

$$S_1 : R_4 \leftarrow R_1, \quad \overline{S_1} \wedge \overline{S_2} : R_4 \leftarrow R_2, \quad \overline{S_1} \wedge S_2 : R_4 \leftarrow R_3$$

Zřejmě se jedná o multiplexor, přepínající výstupy registrů R_1 , R_2 , R_3 do registru R_4 . Multiplexor je řídicími obvody ovládán na základě kombinací stavových signálů S_1 a S_2 . Stavové signály jsou dodávány z některých jiných funkčních bloků systému. Přepínání by se řídilo následujícím předpisem:

- při $S_1 = 1$, bez ohledu na S_2 : R_4 přepnut na R_1
- při $S_1 = 0$ a $S_2 = 0$: R_4 přepnut na R_2
- při $S_1 = 0$ a $S_2 = 1$: R_4 přepnut na R_3

Pro označení logických operací se signály podmínek (tedy na levé straně šipky) se v mnoha publikacích používají i symboly „ $+$ “ (pro OR) a „ \cdot “ (pro AND) – mohlo by tak dojít

k nedorozumění vzhledem k používání symbolu „+“ pro aritmetický součet na pravé straně šipky. Důležitá je, zda symbol „+“ se nalézá na levé nebo pravé straně šipky.

Je-li součástí systému paměť, je přesun dat z paměti a do paměti symbolicky zapsán takto:

$$R_1 \leftarrow M[R_2] \quad ;\text{obsah paměti z adresy uložené v } R_2 \text{ se přesune do } R_1$$
$$M[R_2] \leftarrow R_1 \quad ;\text{do paměti na adresu uloženou v } R_2 \text{ se přesune obsah } R_1$$

Jazyk RTL nenabízí zdaleka takové možnosti pro popis, simulaci a syntézu rozsáhlých systémů, jaké má jazyk VHDL. Dnes je alternativou pro popis jednodušších úloh. Zvláště rozšířen je však pro popis dějů při provádění jednotlivých **instrukcí v počítači** – tam se RTL používá prakticky výhradně.

Jako příklad může sloužit popis instrukce „JC addr“ – podmínečný skok na adresu „addr“ při nastaveném bitu „C“ (Carry). K adresování programové paměti slouží registr „PC“, k dočasnému uložení adresy skoku slouží registr „R_temp“. Instrukční registr je označen jako „IR“.

$IR \leftarrow M[PC]$;instrukční kód do IR
$PC \leftarrow PC + 1$;pokročit na adresovou část instrukce
$R_temp \leftarrow M[PC], \quad PC \leftarrow PC + 1$;adresová část instrukce do pomoc. reg.
$C : PC \leftarrow R_temp, \quad \bar{C} : PC \leftarrow PC + 1$;je-li $C = 1$, přepíše se obsah PC, ;jinak příprava PC na další instrukci

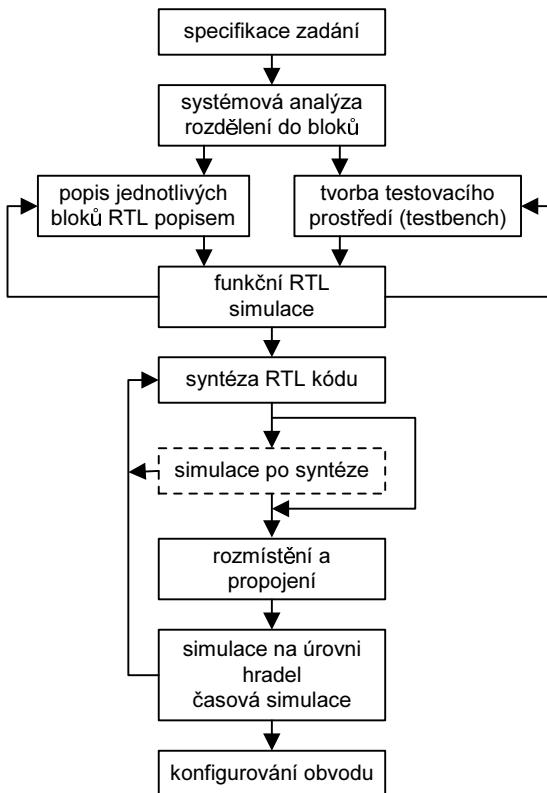
15.6 Návrh číslicového obvodu při použití RTL popisu

Zjednodušený postup návrhu (angl. *Design Flow*) číslicového systému od začátku do konce ukazuje obrázek 15.15. Každý návrh začíná specifikací zadání. Ve specifikaci by měla být zahrnuta zejména úplná funkce navrhovaného obvodu, veškeré další požadavky na návrh, dále např. provozní podmínky (jako teplotní rozsah, pracovní kmitočet, spotřeba), atd.

Dalším krokem je systémová analýza. Cílem tohoto kroku je navrhnut architekturu celého systému. Systém se rozdělí do jednotlivých funkčních bloků (hierarchická struktura) a definují se např. komunikační rozhraní mezi bloky (protokoly, signály), provádí se analýza navrženého řešení, atd. Ve chvíli, kdy je architektura navrhovaného systému známa, lze přikročit k vlastnímu návrhu jednotlivých bloků v některém z HDL jazyků (např. pomocí jazyka VHDL nebo Verilog). Návrh bloků zahrnuje popis jednotlivých funkčních bloků syntetizovatelným RTL popisem a dále také tvorbu testovacího prostředí (tzv. „testbench“) pro tyto bloky a nakonec i pro celý navrhovaný číslicový systém. Tyto dvě činnosti lze realizovat nezávisle na sobě. Někdy jsou tyto činnosti dokonce realizovány navzájem oddělenými týmy vývojářů (jedni tvoří RTL popis, druzí testovací prostředí), aby bylo zajištěno, že zadávací specifikace bude správně interpretována. Tvorba testovacího prostředí je velmi důležitý úkol, který je časově náročnější než tvorba vlastního syntetizovatelného RTL popisu, a ve velké míře na něm závisí bezchybná funkce výsledného obvodu.

Následuje funkční simulace jednotlivých funkčních bloků a po té celého systému pomocí testovacího prostředí. V případě, že systém nesplňuje úplně zadanou specifikaci, je třeba předchozí dva kroky opakovat, dokud funkční simulace nedá správné výsledky. Praxe ukazuje, že tvorba syntetizovatelného RTL popisu, testovacího prostředí a jejich simulace na úrovni RTL kódů zabírá asi 70% celkového času potřebného pro návrh systému. Ve chvíli, kdy je správná funkčnost navrženého systému ověřena funkční simulací na úrovni RTL kódů, lze přistoupit k syntéze. Při tom je transformován RTL kód do vzájemně propojených logických prvků, jimiž disponuje cílová technologie (např. konkrétní řada FPGA obvodů). Výsledek syntézy ovlivňuje řada faktorů, jako např. syntetizovaný RTL kód, nastavená omezení syntézy (angl. *constraints*, např. přiřazení vývodů pouzdra, časové parametry, atd.) a cílová technologie. Výsledkem syntézy je tzv. netlist (nejčastěji ve formátu EDIF), což je většinou textový soubor popisující propojení jednotlivých prvků cílové technologie. Po úspěšné syntéze lze provést funkční simulaci po syntéze (angl. *post-synthesis simulation*). Mnohem reálnější výsledky však dává časová simulace, kterou lze ovšem provést až po rozmístění a propojení. Z tohoto důvodu se většinou simulace po syntéze neprovádí.

Další operací je rozmístění a propojení. V průběhu operace rozmístění je provedena optimalizace logiky, transformace prvků netlistu do prvků cílového obvodu a následně rozmístění těchto prvků v cílovém obvodu (např. v FPGA obvodu). Následuje operace propojení, jejímž cílem je propojení prvků rozmištěných v předchozí operaci. Výsledkem operace rozmištění a propojení je netlist (často ve formátu *VHDL Output File - .vho*) a časové zpoždění jednotlivých prvků získané časovou analýzou (většinou ve formátu *Standard Delay Format Output File - .sdo*). Na základě těchto výsledků se následně provádí tzv. **časová simulace** celého systému. Tato simulace dává výsledky, které již poměrně přesně popisují chování vytvořeného číslicového systému v reálném cílovém obvodu. Tato simulace je ale také nejvíce časově náročná na běh v simulátoru. Trvá zhruba o jeden až dva řády déle než funkční simulace na úrovni RTL kódů. Pokud časová simulace navrženého číslicového systému dává správné výsledky, je velmi vysoká pravděpodobnost, že stejných výsledků bude navržený systém dosahovat i v reálném obvodu (např. v obvodu FPGA). Pokud je číslicový systém realizován v PLD obvodu, následuje ještě operace konfigurování. Tato operace zahrnuje generování konfiguračního souboru (angl. *bitstream*) a následně programování PLD obvodu tímto konfiguračním souborem.



Obr. 15.15 Postup návrhu číslicového obvodu při použití RTL popisu



LITERATURA

- [Alt] Altera : Katalogové listy obvodů CPLD a FPGA.
Dostupné z: <<http://www.altera.com/>>
- [Ash96] Ashenden, P.: The designer's guide to VHDL. Kaufmann, 1996.
ISBN 1-55860-270-4.
- [Coh96] Cohen, B.: VHDL coding styles and methodologies. Kluwer Academic Publ., 1996. ISBN 0-7923-9598-0.
- [DeM94] De Micheli, G.: Synthesis and optimization of digital circuits. McGraw-Hill, 1994. ISBN 0-07-016333-2.
- [Dou02] Stránky firmy Doulos: What is VHDL? [online]. 2002 [cit.2006-03-15].
Dostupné z: <http://www.doulos.com/knowhow/vhdl_designers_guide/what_is_vhdl/>
- [Edw00] Edwards S.: Languages for digital embedded systems. Kluwer Academic Publ., 2000. ISBN 0-7923-7925-X.
- [Erc99] Ercegovac M., Lang T., Moreno J.: Introduction to digital systems. John Wiley & Sons, 1999. ISBN 0-471-57299-8.
- [ESC06] O'Flaherty D., Specter L., Sundararajan P.: FPGA offload through a tightly coupled computing architecture. [cit.2006-03-15].
Dostupné z: <<http://www.celoxica.com/esc/default.asp>>
- [Hun96] Hunter R., Johnson T.: Introduction to VHDL. Chapman & Hall, 1996.
ISBN 0-412-73130-4.
- [IE93-1076] IEEE Std 1076-1993 – IEEE Standard VHDL Language Reference Manual. 1994. ISBN 1-55937-376-8.
- [IE02-1076] IEEE Std 1076-2002 – IEEE Standard VHDL Language Reference Manual. 2002. ISBN 0-7381-3247-0.
- [Koh70] Kohavi Z.: Switching and Finite Automata Theory. McGraw-Hill, 1970.
- [Kol00] Kolouch, J.: Programovatelné logické obvody a modelování číslicových systémů v jazycích ABEL a VHDL. ÚREL FEI VUT v Brně, 2000. ISBN 80-214-1733-1.
- [Lat] Lattice Semiconductor: Katalogové listy obvodů GAL, CPLD a FPGA.
Dostupné z: <<http://www.latticesemi.com/>>
- [Lew05] Lewis J.: VHDL-200X: The Future of VHDL [online]. únor 2005. [cit.2006-03-16].
Dostupné z: <http://www.synthworks.com/papers/vhdl200x_dvcon_2005_p.pdf>
- [Lis93] Liška M., Šulo V., Strelec J.: Programovatelná logická pole. Grada, Praha, 1993.
ISBN 80-85623-26-9.
- [Met] Metastable Response in 5-V Logic Circuits (Application Note SDYA006.pdf).
Dostupné z: <<http://focus.ti.com/lit/an/sdya006/sdya006.pdf>>
- [Nav93] Navabi, Z.: VHDL: analysis and modeling of digital systems. McGraw-Hill, 1993. ISBN 0-07-046472-3.
- [Per72] Perrin J., Denouette J., Daclin E.: Logické systémy 1,2. SNTL 1972.
- [Per94] Perry, D.: VHDL. McGraw-Hill, 1994. ISBN 0-07-049434-7.
- [Pir98] Pirsch P.: Architectures for digital signal processing. John Wiley & Sons, 1998.
ISBN 0-471-97145-6.
- [Rot98] Roth C.: Digital systems design using VHDL. PWS Publishing Comp., 1998.
ISBN 0-534-95099-X.

- [Rus00] Rushton A.: VHDL for logic synthesis. John Wiley & Sons, 2000.
ISBN 0-471-98325.
- [Sal00] Salcic Z., Smailagic A.: Digital systems design and prototyping: using field programmable logic and hardware description languages. Kluwer Academic Publ., 2000. ISBN 0-7923-7920-9.
- [Ska96] Skahill, K.: VHDL for programmable logic. Addison-Wesley Publishing Company, Inc., Menlo Park, 1996. ISBN 0-201-89573-0.
- [Slov2] Minihofner O., Kratochvílová J.: Anglicko-český slovník výpočetní techniky. SNTL Praha, 1987.
- [Slov1] Baloun J. a kol.: Anglicko-český a česko-anglický elektrotechnický a elektronický slovník. SNTL Praha, 1982.
- [TI] Texas Instruments: Katalogové listy číslicových součástek.
Dostupné z: <<http://www.ti.com/>>
- [Xil] Xilinx: Katalogové listy obvodů CPLD a FPGA.
Dostupné z: <<http://www.xilinx.com/>>

426410



SEZNAM ZKRATEK A ANGLICKÝCH TERMÍNŮ

anglicky	česky	kapitola (první výskyt)
----------	-------	-------------------------

<i>ABEL Advanced Boolean Expression Language</i>	jeden z prvních HDL jazyků	13
<i>ACK Acknowledge</i>	potvrzení	15
<i>BO borrow</i>	záporný přenos, výpůjčka	9
<i>bottom-up design</i>	návrh zdola nahoru	15
<i>burst read</i>	dávkové čtení	12
<i>burst refresh</i>	blokové (dávkové) obnovování	12
<i>burst terminate</i>	ukončení dávky	12
<i>burst write</i>	dávkový zápis	12
<i>bus driver</i>	budič sběrnice	5
<i>bus hold</i>	aktivní terminátor	5
<i>clock domain</i>	sféra synchronizace	10
<i>CPLD Complex PLD</i>	komplexní programovatelný logický obvod	13
<i>CS Chip Select</i>	výběr čipu	5
<i>CTR Counter</i>	čítač	9
<i>CY carry</i>	přenos	9
<i>data burst</i>	dávka dat	12
<i>DDR Double Data Rate</i>	dvojnásobná přenosová rychlos	12
<i>DN Down</i>	dolů	9
<i>DPM Dual-Port Memory</i>	dvojbránová paměť	12
<i>EDA Electronic Design Aids</i>	elektronické návrhové prostředky	1
<i>EDA Electronic Design Automation</i>	elektronické návrhové prostředky	3
<i>EDIF Electronic Design Interface Format</i>	formát pro výměnu dat mezi návrhovými systémy	3
<i>EDO Extended Data Output</i>	prodloužený výstup dat	12
<i>EEPROM Electrically Erasable PLD</i>	elektricky vymazatelný PLD	13
<i>EEPROM Electrically Erasable ROM</i>	elektricky vymazatelná PROM	12
<i>EIA Electronic Industries Alliance</i>	sdružení v odvětví elektroniky	3
<i>EN Enable</i>	povolit činnost	8
<i>EPROM Erasable PROM</i>	vymazatelná PROM	12
<i>exclusive OR</i>	vylučující NEBO	6
<i>fall time</i>	doba doběhu	2
<i>FAMOS Floating-Gate Avalanche-Injection MOS</i>	(technologie) MOS s plovoucím hradlem a lavinovou injekcí (elektronů)	12

anglicky	česky	kapitola (první výskyt)
----------	-------	-------------------------

<i>fan-in</i>	vstupní rozvětvení	6	<i>LSB Least Significant Bit</i> nejnižší (nejméně významný) bit	2
<i>fan-out</i>	výstupní rozvětvení	6	<i>LUT Lookup Table</i> vyhledávací tabulka	13
<i>FIFO First In – First Out</i>	paměť fronty	12	<i>memory bank</i> paměťový blok	12
<i>floating gate</i>	plovoucí hradlo	12	<i>monostable</i> monostabilní (klopný obvod)	11
<i>FPGA Field Programmable Gate Array</i>	programovatelné hradlové pole	13	<i>MSB Most Significant Bit</i> nejvyšší (nejvýznamnější) bit	2
<i>FPLA Field Programmable Logic Array</i>	architektura jednoho z prvních programovatelných logických polí	13	<i>non-volatile</i> energeticky nezávislá (paměť)	12
<i>handshake</i>	korespondenční provoz, dotaz–odpověď	10, 15	<i>OLMC Output Logic Macro Cell</i> výstupní makrobuňka	13
<i>HDL Hardware Description Language</i>	jazyk pro popis hardware	1, 3	<i>open drain</i> otevřený drain (kolektor)	13
<i>hold time</i>	doba přesahu	8	<i>OTP One-Time Programmable</i> jednou programovatelná (paměť)	12
<i>IEEE Institute of Electrical and Electronics Engineers</i>	celosvětová organizace sdružující elektroinženýry	3	<i>output enable</i> povolení výstupu	13
<i>IP Intellectual Property</i>	duševní vlastnictví	3	<i>pipeline</i> zřetězená struktura	15
<i>ISP In System Programmable</i>	programovatelný v systému	13	<i>PLA Programmable Logic Array</i> programovatelné logické pole	13
<i>latch</i>	(západka), klopný obvod	8	<i>PLD Programmable Logic Device</i> programovatelný logický obvod	6, 13
<i>LIFO Last In – First Out</i>	zásobníková paměť	12	<i>precharge</i> přípravné nabíjení	12
<i>LPM Library of Parameterized Modules</i>	knihovna parametrizovatelných modulů	3	<i>preload</i> přednastavení	9
			<i>product term</i> součinový člen (term)	13
			<i>PROM Programmable ROM</i> programovatelná ROM	12

anglicky	česky	kapitola (první výskyt)	
<i>propagation delay</i>	zpoždění průchodu (signálu)	2	<i>self refresh</i> automatické obnovování obsahu
<i>pull-down (resistor)</i>	snižovací rezistor	13	<i>set</i> nastavení
<i>pull-up (resistor)</i>	zdvihací rezistor	13	<i>setup time</i> doba předstihu
<i>PWM Pulse Width Modulation</i>	impulzní šířková modulace	11	<i>slew rate</i> rychlosť přeběhu
<i>RAM Random Access Memory</i>	paměť s náhodným přístupem	12	<i>SOP Sum of Products</i> součet součinů
<i>ROM Read-Only Memory</i>	permanentní paměť	12	<i>SPLD Standard PLD</i> standardní programovatelný logický obvod
<i>refresh</i>	obnovování informace	12	<i>state diagram</i> stavový diagram
<i>reset</i>	nulování	8	<i>time-distributed refresh</i> rozložené obnovování
<i>retriggerable</i>	znova spustitelný	11	<i>top-down design</i> návrh shora dolů
<i>rise time</i>	doba náběhu	2	<i>transparent refresh</i> skryté obnovování
<i>ROM Read Only Memory</i>	permanentní paměť	12	<i>UV Ultra Violet</i> ultrafialové (záření)
<i>RQ Request</i>	požadavek	15	<i>VHDL Output File</i> formát souboru v jazyce VHDL po syntéze
<i>RTL Register Transfer Language</i>	jazyk RTL	15	<i>VHDL VHSIC HDL</i> HDL jazyk, produkt projektu VHSIC
<i>RTL Register Transfer Level system</i>	systém založený na přesunech mezi registry	15	<i>VHSIC Very High Speed Integrated Circuit</i> velmi rychlý integrovaný obvod – projekt, z něhož vznikl jazyk VHDL
<i>RWM Read-Write Memory</i>	paměť pro čtení i zápis	12	<i>volatile memory</i> energeticky závislá paměť
<i>SDO Standard Delay Format Output File</i>	formát výstupního souboru popisujícího zpoždění	3	



PŘÍLOHA: ZJEDNODUŠENÁ SYNTAXE JAZYKA VHDL

Použité formátování pro výpisy kódu a syntaxi jazyka VHDL

[]	volitelné	{ }	lze použít opakování
	alternativa	()	seskupení
tučně	povinné	ID	identifikátor

Lexikální prvky

```
komentář      -- toto je text komentare
identifikátor  pismeno{[_]pismeno_nebo_cislice}
dekadicke č.   cislo{[_]cislo}{[_]cislo{[_]cislo}}[E[+|-]cislo]
jiné soustavy  soust#cislo{[_]cislo}{[_]cislo{[_]cislo}}#[Ecislo]
řetězec        "retezec"
bitový řetězec B|O|X"cislo{[_]cislo}"
```

Předdefinované výčtové typy

```
bit           '0', '1'
boolean       false, true
character     8bit znak dle ISO 8859-1:1987 (Latin-1)
file_open_kind read_mode, write_mode, append_mode
file_open_status open_ok, status_error, name_error, mode_error
severity_level note, warning, error, failure
```

Předdefinované celočíselné typy

```
integer       32bit nebo 64bit integer
natural       32bit nebo 64bit integer (0 až max.)
positive     32bit nebo 64bit integer (1 až max.)
```

Předdefinované fyzické typy

```
time          fs, ps, ns, us, ms, sec, min, hr
delay_length  time (0 až max.)
```

Předdefinovaný typ s plovoucí řádovou čárkou

```
real          rozsah -1,0 * 10E308 až +1,0 * 10E308 včetně
```

Předdefinované typy pole

```
bit_vector    (natural range <>) pole typu bit
string        (positive range <>) pole typu character
```

Předdefinované funkce

```
now           vrátí současný simulační čas, typ delay_length

procedure file_open (file f : FTYPEID; external_name : in string;
                     open_kind : in file_open_kind := read_mode);

procedure file_open (status : out file_open_status; file f : FTYPEID;
                     external_name : in string;
                     open_kind : in file_open_kind := read_mode);
```

```

procedure file_close (file f : FTYPEID);
procedure read (file f : FTYPEID; value : out TYPEID);
procedure write (file f : FTYPEID; value : in TYPEID);
function endfile (file f : FTYPEID) return boolean;

```

Předdefinované operátory (vzrůstající prioritní)

logické op.	and, or, nand, nor, xor, xnor
relační op.	=, /=, <, <=, >, >=
op. posuvu	sll, srl, sla, sra, rol, ror
op. sčítání	+, -, &
znaménkové op.	+, -
násobicí op.	*, /, mod, rem
různé op.	**, abs, not

Definice

Definice výčtového typu

```
type TYPEID is (ID {, ID});
```

Definice celočíselného typu

```
type ID is range cele_cislo downto | to cele_cislo;
```

Definice fyzického typu

```

type ID is range cislo downto | to cislo
  units
    ZAKLADNI_JEDNOTKA;
    {DALSI_JEDNOTKA;}
  end units [ID];

```

Definice typu s plovoucí řádovou čárkou

```
type ID is range desetine_cislo downto | to desetine_cislo;
```

Definice typu pole

```

type ID is array ({rozsah | TYPEID,}) of TYPEID;

rozsah (cislo | ENUMID downto | to cislo | ENUMID) |
  OBJID[reverse_]range | TYPEID range <>

```

Definice typu záznam

```

type ID is record
  ID : TYPEID;
  {ID : TYPEID,}
end record [ID];

```

Definice typu soubor

```
type FTYPEID is file of TYPEID;
```

Definice atributu

```
attribute ID : TYPEID;
```

Deklarace

Deklarace konstanty

```
constant ID : TYPEID := vyraz;
```

Deklarace signálu

```
signal ID : TYPEID [:= vyraz];
```

Deklarace proměnné

```
[shared] variable ID : TYPEID [:= vyraz];
```

Deklarace souboru

```
file ID : FTYPEDID [[open read_mode | write_mode | append_mode] is "jmeno"];
```

Deklarace aliasu

```
alias ID [: TYPEID] is OBJID;
```

Deklarace atributu

```
attribute ATTRID of OBJID : objekt {, objekt} | others | all is vyraz;
```

Deklarace komponenty

```
component ENTITYID is
    [generic ({ID : TYPEID [:= vyraz];});]
    [port ({ID : [in | out | inout | buffer | linkage] TYPEID [:= vyraz];});]
end component [ENTITYID];
```

Deklarace funkce

```
[impure | pure] function ID [({[constant | variable | signal |
    file] ID : [in] TYPEID [:= vyraz];})] return TYPEID [is
begin
    {sekvencni_vyrazy}
end [function] [ID];
```

Deklarace procedury

```
procedure ID[({[constant | variable | signal] ID : in | out |
    inout TYPEID [:= vyraz];})] is
begin
    {sekvencni_prikazy}
end [procedure] [ID];
```

Knihovní jednotky

Připojení knihovny

```
library jmeno_knihovny [{, jmeno_knihovny}];
```

Zpřístupnění obsahu knihovního balíku

```
use jmeno_knihovny.jmeno_baliku.[all | identifikator_nebo_operator];
```

Deklarace knihovního balíku

```
package jmeno_baliku is
  {deklarace_hlavicek_funkci_a_procedur}
  {deklarace_typu_a_podtypu}
  {deklarace_konstant_a_signalu}
  {deklarace_sdilenyh_promennych}
  {deklarace_souboru_a_aliasu}
  {deklarace_komponentu}
  {deklarace_a_definice_atributu}
end [package] [jmeno_baliku];
```

Deklarace těla knihovního balíku

```
package body jmeno_baliku is
  {deklarace_a_tela_hlavicek_funkci_a_procedur}
  {deklarace_typu_a_podtypu}
  {deklarace_konstant}
  {deklarace_sdilenyh_promennych}
  {deklarace_souboru}
  {deklarace_aliasu}
end [package body] [jmeno_baliku];
```

Deklarace entity

```
entity ENTITYID is
  [generic ({ID : TYPEID [:= vyraz]});]
  [port ({ID : [in | out | inout | buffer | linkage] TYPEID [:= vyraz]});]
  [{deklarace}]
begin
  {paralelni_prikazy}
end [entity] [ENTITYID];
```

Deklarace architektury

```
architecture ARCHID of ENTITYID is
  [{definice_a_deklarace}]
begin
  {paralelni_prikazy}
end [architecture] [ARCHID];
```

Paralelní příkazy

Nepodmíněné přiřazení

```
[NAVESTI :] SIGID <= [transport | [reject cas] inertial] vyraz_elem
  [, vyraz_elem];

vyraz_elem = vyraz [after cas] | null [after cas]
```

Podmíněné přiřazení

```
[NAVESTI :] SIGID <= [transport | [reject cas] inertial]
  {vyraz_elem when podminka else} vyraz_elem [when podminka];
```

Výběrové přiřazení

```
[NAVESTI :] with ID select
    SIGID <= [transport | [reject cas] inertial]
        vyraz_elem when vyber {[ , vyraz_elem when vyber}};

vyber =  vyraz | rozsah_hodnot | others
```

Process

```
[NAVESTI :] [postponed] process [(SIGID {, SIGID})]
    [{definice}]
    [{deklarace}]
begin
    [{volani_procedur}]
    [{sekvencni_ridici_prikazy}]
    [{sekvencni_pripravovaci_prikazy_pro_signaly_nebo_promenne}]
end [postponed] process [NAVESTI];
```

Příkaz generate

```
NAVESTI : for ID in od to | downto do generate
    [{blok_deklaraci}]
begin
    {paralelni_vyrazy}
end generate [NAVESTI];
```

```
NAVESTI : if podminka generate
    [{blok_deklaraci}]
begin
    {paralelni_vyrazy}
end generate [NAVESTI];
```

Volání procedury

```
[NAVESTI :] PROCID[([PARID =>] vyraz {, [PARID =>] vyraz})];
```

Příkaz bloku

```
BLOCKID : block [is]
    [generic ({ID : TYPEID [:= vyraz];});
     [generic map ([GENCID =>] vyraz {,[GENCID =>] vyraz})];
     [port ({ID: [in | out | inout | buffer] TYPEID[:=vyraz];});
      [port map ([PORTID =>] vyraz {, [PORTID =>] vyraz});]
     [{deklarace}]
begin
    {paralelni_prikazy}
end block [BLOCKID];
```

Použití komponentu (např. entity)

```
NAVESTI : COMPID
    [generic map ([GENCID =>] vyraz {, [GENCID =>] vyraz})]
    port map ([PORTID =>] vyraz {, [PORTID =>] vyraz});
```

Použití entity odkazem do knihovního balíku

```
NAVESTI : entity [LIBID.]ENTITYID [(ARCHID)]  
  [generic map ([GENCID =>] vyraz {, [GENCID =>] vyraz})]  
  port map ([PORTID =>] vyraz {, [PORTID =>] vyraz});
```

Použití konfigurace odkazem do knihovního balíku

```
NAVESTI : configuration [LIBID.]CONFID  
  [generic map ([GENCID =>] vyraz {, [GENCID =>] vyraz})]  
  port map ([PORTID =>] vyraz {, [PORTID =>] vyraz});
```

Příkaz assert

```
[NAVESTI :] [postponed] assert podminka  
  [report vyraz_typy_retezec]  
  [severity note | warning | error | failure];
```

Sekvenční příkazy

Sekvenční přiřazení do proměnné

```
VARID := vyraz;
```

Sekvenční přiřazení do signálu

```
[NAVESTI :] SIGID <= [transport | [reject cas] inertial] vyraz_elem  
  {, vyraz_elem};  
  
vyraz_elem = vyraz [after cas] | null [after cas]
```

Sekvenční příkaz wait

```
wait [on SIGID {, SIGID}] [until vyraz] [for doba];
```

Sekvenční příkaz if

```
[NAVESTI :] if podminka1 then  
  {sekvence_prikaz1}  
  [{elseif podminka2 then  
    {sekvence_prikazu2}}]  
  [else  
    {sekvence_prikazu}]\nend if [NAVESTI];
```

Sekvenční příkaz case

```
[NAVESTI :] case vyraz is  
  when hodnota { | hodnota} =>  
    {sekvence_prikazu}  
  {when hodnota { | hodnota} => {sekvence_prikazu}}  
end case [NAVESTI];
```

Sekvenční příkaz loop

```
[NAVESTI :] [iteracni_schema] loop
    {sekvence_prikazu}
end loop [NAVESTI];

iteracni_schema = while podminka
                for ID in od to | downto do
```

Sekvenční příkaz next (lze použít v těle loop)

```
[NAVESTI :] next [LOOPID] [when podminka];
```

Sekvenční příkaz exit (lze použít v těle loop)

```
[NAVESTI :] exit [LOOPID] [when podminka];
```

Sekvenční příkaz return

```
[NAVESTI :] return [vyraz];
```

Sekvenční příkaz assert

```
[NAVESTI :] assert podminka
            [report vyraz_typu_retezec]
            [severity note | warning | error | failure];
```

Sekvenční příkaz report

```
[NAVESTI :] report vyraz_typu_retezec
            [severity note | warning | error | failure];
```

Příkaz null

```
[NAVESTI :] null;
```

Příkaz procedure

```
procedure ID[({constant | variable | signal} ID : in | out |
            inout TYPEID [= vyraz];})] is
begin
    {sekvencni_prikazy}
end [procedure] [ID];
```

Sekvenční volání procedury

```
PROCID[([PARID =>] vyraz {, [PARID =>] vyraz})];
```

Příkaz function

```
[impure | pure] function ID [({constant | variable | signal |
            file} ID : [in] TYPEID [= vyraz];})] return TYPEID [is
begin
    {sekvencni_prikazy}
end [function] [ID];
```

Uživatelem definované atributy

```
attribute ID : TYPEID;  
  
attribute ATTRID of OBJID : objekt {, objekt} | others | all is výraz;  
  
objekt = architecture | component | configuration | constant |  
entity | file | function | group | label | literal |  
package | procedure | signal | subtype | type |  
variable | units
```

Předdefinované atributy typů

T'ascending	vrací <u>true</u> pokud má typ vzestupný rozsah
T'base	vrátí základní typ daného typu
T'left	vrátí hodnotu daného typu, která je nejvíce vlevo
T'right	vrátí hodnotu daného typu, která je nejvíce vpravo
T'high	vrátí nejvyšší hodnotu daného typu
T'low	vrátí nejnižší hodnotu daného typu
T'pos(x)	vrátí pozici hodnoty <u>x</u> v daném typu
T'val(x)	vrátí hodnotu na pozici <u>x</u> daného typu
T'succ(x)	vrátí hodnotu na poz. o jednu větší než je poz. hodnoty <u>x</u>
T'pred(x)	vrátí hodnotu na poz. o jednu menší než je poz. hodnoty <u>x</u>
T'leftof(x)	vrátí hodnotu vlevo od hodnoty <u>x</u> daného typu
T'rightof(x)	vrátí hodnotu vpravo od hodnoty <u>x</u> daného typu
T'image(x)	převede výraz <u>x</u> daného typu na textový řetězec
T'value(x)	převede textový řetězec na hodnotu daného typu

Předdefinované atributy polí

A'ascending[(n)]	vrací <u>true</u> pokud má <u>n-tá</u> dimenze vzestupný rozsah
A'left[(n)]	vrátí levou mez <u>n-té</u> dimenze
A'right[(n)]	vrátí pravou mez <u>n-té</u> dimenze
A'high[(n)]	vrátí horní mez <u>n-té</u> dimenze
A'low[(n)]	vrátí dolní mez <u>n-té</u> dimenze
A'range[(n)]	vrátí rozsah <u>n-té</u> dimenze
A'reverse_range[(n)]	vrátí obrácený rozsah <u>n-té</u> dimenze
A'length[(n)]	vrátí počet prvků <u>n-té</u> dimenze

Předdefinované atributy signálů

S'active	vrátí <u>true</u> , pokud byl aktivní (nastala transakce)
S'last_active	vrátí čas od poslední aktivity (transakce)
S'delayed[(t)]	vytvoří kopii signálu zpožděnou o čas <u>t</u>

S'stable[(t)]	vrátí <i>true</i> , pokud signál nezměnil po dobu \underline{t} hodnotu
S'quiet[(t)]	vrátí <i>true</i> , pokud byl signál po dobu \underline{t} neaktivní
S'transaction	při každém výskytu transakce (aktivitě) změní hodnotu
S'event	vrátí <i>true</i> , pokud změnil hodnotu (nastala událost)
S'last_event	vrátí čas od poslední změny hodnoty (události)
S'last_value	hodnotu signálu před jeho poslední změnou
S'driving	vrátí <i>true</i> , pokud je signál buzen
S'driving_value	vrátí budící hodnotu signálu

Předdefinované atributy pojmenovaných objektů

O'simple_name	vrátí řetězec se jménem objektu
O'instance_name	vrátí řetězec s cestou k objektu včetně jména objektu
O'path_name	vrátí řetězec s cestou k objektu (bez jména objektu)

A large, stylized letter 'R' is positioned in the upper right quadrant of the page. The letter is white with a thick grey outline. It has a sharp vertical stroke on the left and a rounded, sweeping curve on the right, ending in a small hook. The background behind the letter is a solid grey rectangle.

REJSTŘÍK

Rejstřík

A

abs • 40
after • 44
aktivní terminátor • 99, 282
alias • 42
and • 37
AND matice • 275
Architektura • 28
array • 35
asociativní zákon • 105
assert • 51, 57
asynchronní čítač • 190
asynchronní přenos • 197
attribute • 67
autonomní automat • 205

B

BCD kód • 156
bezpodmínečný přechod • 293
BiCMOS • 93
bit_vector • 35
block • 51
budící funkce • 214
budič sběrnice • 97

C

case • 54
citlivostní seznam • 47
CMOS • 89
component • 49
constant • 41
CPLD • 272

Č

časová simulace • 78
čtecí zesilovače • 246
čtení po stránkách • 249

D

datová cesta • 307
dělič kmitočtu • 190
děliče kmitočtu • 201
DeMorganovy zákony • 106
detektor změn • 227
diodová logika • 85
distributivní zákon • 105
doba latence • 252
doba předstihu • 163, 220, 288
doba přesahu • 163, 220
doba zpoždění • 16, 288
doběh • 17
doby náběhu • 17
doby odblokování • 96
doby zablokování • 96
dvojitý vzorkovač • 309, 310
dynamický hazard • 138

E

EEPROM • 255
ekvivalentní hradlo • 272
else • 45, 54
elsif • 54
entita • 27
EPROM • 255
exit • 57
EX-OR • 123

F

falešné impulzy • 128
file • 36, 43
FLASH • 256
for • 55
for generate • 48
FPGA • 272, 283
function • 59
Funkce • 59
funkční simulace • 78

G

GAL • 277
generate • 48
generátor logické funkce • 285
generic map • 49

H

Hardware Description Language • 20
hazard • 131
hazardní stavy • 128
HDL • 20

Ch

charakteristika s hysterezí • 83

I

if • 54
if generate • 49
implikanty • 110
inertial • 44
integer • 32
I/O blok • 282

J

jazyk RTL • 318
jednobitová sčítáčka • 152
Johnsonův čítač • 185

K

kapacitní zátěž • 83
Karnaughova mapa • 112
klopný obvod • 160
kombinační obvody • 104
komparátor • 125
kompatibilita • 92
komutativní zákon • 105
konečný automat • 204
konsensus • 106
kruhový registr • 185

L

lineární čítač • 187
loop • 55
LUT • 272

LUT tabulka • 285

M

makrobuňka • 281
Master Slave • 171
maxtermy • 107
MDNF • 111
Mealyho automat • 204
metastabilní stav • 161, 220
mezistavy • 135
mikroinstrukce • 291
mikroprogram • 291
mikroprogramový automat • 291
minimalizace • 111, 114
mintermy • 107
monostabilní klopný obvod • 228
Mooreho automat • 205
multiplexor • 120

N

nand • 37
natural • 32
nedestruktivní konflikt • 100
neurčené stavy • 108, 112
next • 57
nezapojené vstupy • 93
nor • 37
not • 37
null • 57
nulování • 181

O

Obnovovací cykly • 249
obnovování informace • 252
odběrová charakteristika • 87
omezovače přepětí • 90
or • 37

P

package • 75
package body • 75
paměť • 125
paměť EEPROM • 259
paměť EPROM • 258
paměť FLASH • 260

paměťový blok • 251
pipeline • 312
PLD • 272
plovoucí hradlo • 257
podmíněné setrvání • 293
pohlcení • 105
pokrytí • 110
pomalý průchod • 83
port • 27
 buffer • 27
 in • 27
 inout • 28
 linkage • 28
 out • 27
port map • 49
positive • 32
pravdivostní tabulka • 106
primární jednotky • 67
princip duality • 105
Procedura • 59
procedure • 59
process • 47
PROM • 255
přednastavení • 181
přechodová funkce • 204
převodní charakteristika • 82
příkon • 90
přípravné nabíjení • 246, 254
PWM • 234

R

record • 35
referenční napětí • 15
reject • 44
report • 58
return • 57
rol • 38
ROM • 255
ror • 38
rozklady funkcí • 121

Ř

řádkový dekodér • 240
řízení hranou • 165

S

sběrnice • 97
sekundární jednotky • 67
select • 46
Shannonova věta • 120
shared variable • 42
signal • 41
signed • 71
skupinová minimalizace • 111
sla • 38
sll • 38
sloupcový dekodér • 240
smyčka • 114
součinová matice • 275
sousední pole • 112
SPLD • 272
spojování • 105, 109
sra • 38
srl • 38
stavové diagramy • 205
std_logic • 69
std_logic_vector • 69
std_ulogic • 69
std_ulogic_vector • 69
string • 35
synchronizační impulzy • 205
synchronní detektor hran • 231
synchronní spínač • 175

T

tabulka implikantů • 110
terminátory • 97
testbench • 77
transport • 44
třívodičové spojení • 239
TTL obvody • 86
Typ bit • 31
Typ boolean • 31
typ delay_length • 33
Typ character • 31
typ real • 34
typ time • 33

U

ÚDNF • 107
ÚKNF • 107
units • 33
unsigned • 71
úrovní řízený • 163

V

variable • 42
VHSIC • 20
vícebitová sčítačka • 152
vybavovací doba • 239, 244
vyrovňávací registry • 136, 312
vysokoimpedanční stav • 96
výstupní funkce • 204
výstupní makrobuňka • 274, 278
výstupní rozvětvení • 83
vývojový diagram • 293
vzorkování vektoru • 311

W

wait for • 48, 53
wait on • 48, 53
wait until • 48, 53
when • 45, 55
with • 46

X

xnor • 37
xor • 37

Z

znovu spustitelný • 229
zpoždění kombinačního obvodu • 128
zpoždění logického členu • 119

Symboly

(mod) • 40
(rem) • 40

426410

INTEGROVANÉ OBVODY A JEJICH APLIKACE

POSKYTUJEME SLUŽBY SOUVISEJÍCÍ S NÁVRHEM A VÝROBOU
STANDARDNÍCH I ZÁKAZNICKÝCH INTEGROVANÝCH OBVODŮ

ASICentrum je vývojové centrum švýcarské firmy EM Microelectronic
a patří tak do mezinárodní skupiny Swatch Group.

Nás tým návrhářů vyvinul celou řadu obvodů pro telekomunikační, automobilové,
spotřební, počítačové a průmyslové aplikace. Stále se rozvíjející široké návrhové
zkušenosti a využívání efektivních vývojových postupů pomocí moderního HW a SW
vybavení umožňují návrhářům pružně reagovat na požadavky zákazníků.

SYSTÉMY S EXTRÉMNĚ
NÍZKÝM PŘÍKONEM
A NÍZKÝM NAPÁjecím
NAPĚTÍM

OBVODY A VÝVOJOVÉ
PROSTŘEDKY RFID
PRO VŠECHNA
FREKVENČNÍ PÁSMA

OBVODY PRO
SMART KARTY

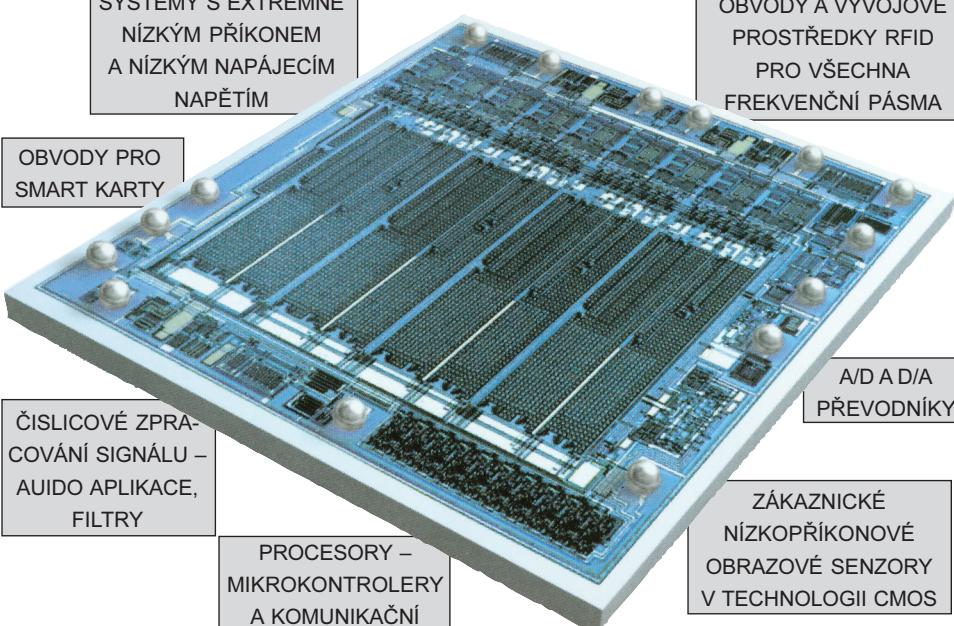
A/D A/D
PŘEVODNÍKY

ČÍSLICOVÉ ZPRA-
COVÁNÍ SIGNÁLU –
AUDIO APLIKACE,
FILTRY

ZÁKAZNICKÉ
NÍZKOPŘÍKONOVÉ
OBRAZOVÉ SENZORY
V TECHNOLOGII CMOS

PROCESORY –
MIKROKONTROLERY
A KOMUNIKAČNÍ
OBVODY

SYSTÉMY SE SENZORY TLAKU, TEPLITRY, ...



VOLNÁ PRACOVNÍ MÍSTA

SENIOR ANALOG MIXED SIGNAL IC DESIGNERS

SENIOR DIGITAL IC DESIGNERS

DIGITAL IC DESIGNERS

Kontakt:

Novodvorská 994, 142 21 Praha 4, tel. 239 043 478

www.asicentrum.cz, e-mail: hr@asicentrum.cz

Nabízíme kvalitní a cenově dostupné měniče!

**Využijte naši nabídku
- vzorky zdarma!**

- přes 1 800 standardních modelů
- výkon od 0,25 W do 30 W
- SMD, SIL a DIL pouzdra
- izolační napětí do 6000 VDC
- operační teplota $-40^{\circ}\text{C} - +85^{\circ}\text{C}$
- účinnost až 92 %
- zákaznické typy
- RoHS



Elektronické součástky
Luženice 10, 344 01 Domažlice
T: 379 723 915, F: 379 725 868
www.emi.cz info@atd-elektronik.cz



Podrobnější informace na: www.atd-elektronik.cz a na info@atd-elektronik.cz



- Indukčnosti, cívky, ferity, ...
- RF konektory, kabely, ...



Knihy nakladatelství BEN – technická literatura

DOPORUČUJEME



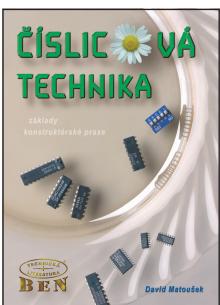
Mikroprocesory a mikropočítače

Cílem publikace je seznámení čtenáře s obecnou problematikou počítačů a mikrokontrolérů tak, aby byl schopen v krátké době začít používat kterýkoliv konkrétní typ procesoru, mikroprocesoru či mikrokontroléru. Výklad není zaměřen na jeden typ obvodu, ale jsou zobecněny základní vlastnosti. Vychází se přitom z průjezdu produkce hlavních světových výrobčů z posledních let. Hlavní důraz je kláden na mikrokontroléry, jejich vlastnosti, rozšíření o vnější obvody a zásady použití.

Text je doplněn přehlednými obrázky a souborem doporučené literatury. V přílohách je obsažen i stručný přehled nutných vstupních znalostí, které mohou být prospěšné pro ty čtenáře, kteří se číslicovou technikou v poslední době nezabývali.

Jiří Pinker, 160 stran B5, obj. číslo 121158

<http://shop.ben.cz/121158>



Číslicová technika

Kniha velice podrobně popisuje druhy číslicových obvodů a jejich použití. Je určena nejen začátečníkům, protože pokročilí „bastlíři“ zde nalezou řadu dosud nepublikovaných konstrukcí.

První tři kapitoly jsou určeny především začátečníkům. Jsou zde uvedeny a vysvětleny základní pojmy číslicové techniky včetně aplikací kombinacích obvodů (hradel) i sekvenčních obvodů (klopních obvodů, čítačů a posuvních registrů).

David Matoušek, 208 stran A5, obj. číslo 121060

<http://shop.ben.cz/121060>



Moderní učebnice programování PIC 1. díl

Výuka je vedena na osmibitových mikrokontrolérech PIC firmy Microchip, protože jsou z hlediska výuky nejvhodnější. Také spousta profesionálních systémů tyto mikrokontroléry bohatě využívá. Patří mezi nejvýkonnější osmibitové mikrokontroléry. Jsou vyráběny od nejjednodušších typů až po typy bohatě vybavené integrovanými perifériemi.

Výuka je vedena na množství příkladů a animací nalézajících se na doprovodném CD, takže je od samého začátku zajímavá a bez zbytečně dlouhých teoretických úvodů.

Tento první díl učebnice nás naučí používat mikrokontrolér, jeho paralelní vstupy/výstupy, základní skupinu instrukcí a základní zásady psaní programů v assembléru s využitím MPLABu.

Jiří Hrbáček, 96 stran A5 + CD, obj. číslo 121109

<http://shop.ben.cz/121109>



Mikrokontroléry PIC16F87X

Kniha „Mikrokontroléry PIC16F87X“ obsahuje popis funkcí a vlastností novější řady mikrokontrolérů PIC16F873, PIC16F874, PIC16F876, PIC16F877 od firmy Microchip, které umožňují ukládat data do paměti EEPROM nebo RAM a program do paměti FLASH. Popis je zpracován pro českého čtenáře.

Kniha „Mikrokontroléry PIC16F87X“ není a ani nechce být náhradou originálního katalogu. Přesto se domníváme, že tento popis může být účinnou pomůckou pro orientaci o možnostech této řady vysoké výkonné mikrokontrolérů. Najdete zde i mimo jiné informace o odlišnostech mezi mikrokontroléry PIC16F87X a PIC16F87XA.

Oldřich Peroutka, 256 + 40 stran B5, obj. číslo 121199

<http://shop.ben.cz/121199>



Knihy nakladatelství BEN – technická literatura:

EDICE PC & elektronika



Udělejte si z PC – 1. díl

– generátor, čítač, převodník, programátor...

Měření, řízení a regulace pomocí sériového portu PC a sběrnice I²C

Autor D. Matoušek, 176 stran B5 + CD, obj. číslo 121069

Udělejte si z PC – 2. díl

Měření, řízení a regulace pomocí portů PC prostřednictvím několika jednoduchých přípravků

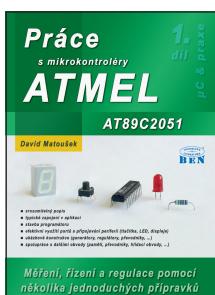
Komunikace PC s aplikacemi mikrokontrolérů řady AT89C2051, stavba jednoduchého programátoru mikrokontroléru AT89C2051

Autor Ing. David Matoušek, 180 stran B5 + CD ROM, obj. číslo 121114

Udělejte si z PC v Delphi – 1. díl

Autor D. Matoušek, 272 stran B5 + CD, obj. číslo 121161

EDICE µC & praxe



Měření, řízení a regulace pomocí několika jednoduchých přípravků

Práce s mikrokontroléry Atmel AT89C2051, 1. díl

Autor D. Matoušek, 248 stran B5 + CD, obj. číslo 121093

Práce s mikrokontroléry Atmel AT89S8252, 2. díl

Autor D. Matoušek, 304 stran B5 + CD, obj. číslo 121112

Práce s mikrokontroléry Atmel AVR, 3. díl

Autor D. Matoušek, 376 stran B5 + CD, obj. číslo 121130

Práce s mikrokontroléry Atmel ATmega16, 4. díl

Autor D. Matoušek, 320 stran B5 + CD, obj. číslo 1212510

Práce s mikrokontroléry Atmel AT89LP2052/4052, 5. díl

Autor D. Matoušek, 200 stran B5 + CD, obj. číslo 121275

edice ATMEL AVR

- Mikrokontroléry Atmel AVR – vývojové prostředí
- Mikrokontroléry Atmel AVR – popis procesoru a instrukční soubor
- Mikrokontroléry Atmel AVR – Assembler
- Mikrokontroléry Atmel AVR – programování v jazyce C
- Mikrokontroléry Atmel AVR – Pascal
- Mikrokontroléry Atmel AVR – programování v jazyce Bascom



*Veškerá technická
a počítačová literatura
pod jednou střechou*

Adresy prodejen technické literatury

PRAHA 10, Věšínova 5, tel. 274 820 211, 274 818 412

PLZEŇ, sady Pětatřicátníků 33, tel. 377 323 574

BRNO, Veveří 13, tel. 545 242 353

OSTRAVA, Českobratrská 17, tel. 596 117 184

centrála: **BEN, Věšínova 5, 100 00 PRAHA 10**
zásilk. služba: tel. 274820411, 274816162, fax 274822775
distribuce: tel. 274820211, 274818412, fax 274822775
Internet: <http://www.ben.cz>
adresa knihy: <http://shop.ben.cz/121736>
e-mail: knihy@ben.cz (objednávky zboží)
redakce@ben.cz (připomínky ke knize)

CENTRÁLA



**Věšínova 5,
100 00 PRAHA 10**

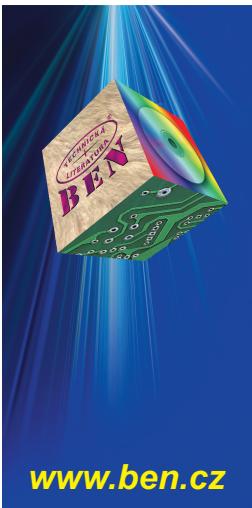
V naší centrále
jsou
soustředěna
všechna
oddělení:

prodejna
sklad
zásilková
služba
distribuce
nakladatelství

Po-Pá 9⁰⁰-18⁰⁰ So viz www

Pouhých 200 metrů od stanice metra „Strašnická“ !!

Pár slov o nakladatelství



Nakladatelství **BEN** – **technická literatura** se věnuje vydávání převážně počítačové a elektrotechnické literatury. Nakladatelství je součástí stejnojmenné firmy, která se zabývá prodejem a distribucí veškeré technické a počítačové literatury, jež v poslední době v České republice vyšla. Dále pak prodejem a distribucí zejména původních českých titulů na CD ROM a DVD. Přehledy české technické literatury – ediční plány (katalogy), vydáváme několikrát ročně, na vyžádání je zasíláme poštou. Celková nabídka je soustředěna do několika specializovaných prodejen.

Adresa této knihy na Internetu:

<http://shop.ben.cz/121736>

Jiří Pinker, Martin Poupa

Číslicové systémy a jazyk VHDL

Vydalo nakladatelství BEN – technická literatura, Praha 2009

1. dotisk 1. vydání

Vedoucí nakladatelství Libor Kubica

Vedoucí redakce a DTP Hana Züglerová

Odpovědný a technický redaktor Jan Paroulek

Sazba Jan Paroulek

Obálka Libor Kubica, foto na obálku firma Kontron

Rozsah 352 stran

Tisk Marten

objednací číslo 121736

EAN 9788073001988

ISBN 978-80-7300-198-5 (tištěná kniha)

978-80-7300-385-2 (elektronická kniha v PDF)

