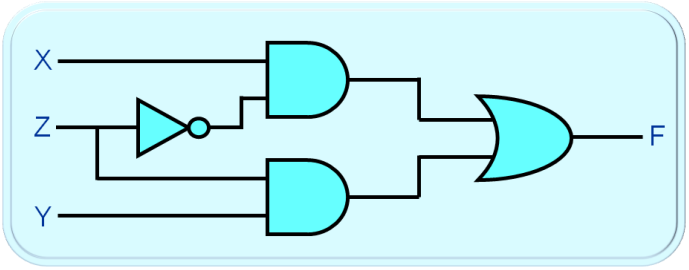


Logické obvody na FPGA

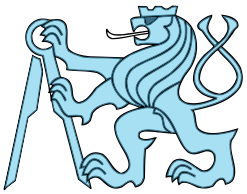
studijní text předmětu

Logické systémy a procesory

Richard Šusta

Katedra řídicí techniky

ČVUT-FEL v Praze



Verze 3.1 ze dne 5. dubna 2025

Autor: Richard Šusta,
susta@fel.cvut.cz, richard@susta.cz, https://susta.cz/

Ilustrace: Richard Šusta,
s výjimkou několika málo převzatých obrázků
— u těch se vždy explicitně uvádí jejich zdroj.

Publisher: Katedra řídicí techniky ČVUT-FEL v Praze,
Karlovo nám. 13
121 35 Praha 2
https://control.fel.cvut.cz/

Datum vydání: V3.0 únor 2025;
V3.1 duben 2025 *korekce překlepů a rozšíření kapitoly 7.7*

Délka: 160 stran

Domácí stránka dokumentu:

https://dcenet.fel.cvut.cz/edu/fpga/navody.aspx

Obsah

1 O učebnici Návrhy logických obvodů na FPGA

Návrhy obvodů se dnes popisují textovými příkazy v jazycích HDL (*Hardware Description Language*), k nimž patří například VHDL, Verilog či SystemVerilog. Zde musíme zdůraznit slovo "*description*". Ať si zvolíme kterýkoli z nich, potřebujeme napřed rozumět vlastnostem reálných logických zapojení a vědět, na co si máme dávat pozor při návrhu obvodů. Teprve pak můžeme popisovat obvod. Vybrané minimum znalostí se shrnulo do dvou učebnic:

Binární prerekvizita vysvětluje kódování celých čísel se znaménkem a bez něho, hexadecimální a BCD zápisy a vzájemné převody, i uložení písmen. Jde o základy, jejichž bezpečnou znalost předpokládají jak další učebnice, tak přednášky. Věříme, že většina čtenářů již zná kódování čísel, a kvůli tomu jsme jeho výklad osamostatnili.

Logické obvody na FPGA, které právě čtete, se věnují hlavním logickým konstrukcím a principům, bez nichž nelze cokoli navrhovat. Najdete v ní obecné znalosti o logických obvodech, ale bez jejich popisů v HDL jazycích. Jen v závěru se pro srovnání u některých čítačů a posuvných registrů uvedou jejich popisy ve VHDL i Verilogu. Jinak se vše ostatní vysvětluje na schématech a někde i s jejich realizací na FPGA, o nichž bude víc hned na str. 7.

- Napřed se projdou přímé aplikace teorémů Booleovy logiky na zapojení obvodů.
- Následuje kapitola o logických funkcích, a to od jejich zadání až po minimalizaci pomocí Karnaughových map.
- Poté se přiblíží vnitřní struktura CMOS hradel a jejich základní vlastnosti, ze kterých vyplývá nejen řada zapojení, ale i chování obvodů.
- Další kapitola se věnuje základním kombinačním obvodům. Začne se dekodéry a multiplexory. Nahlédne se zde i do nitra FPGA obvodu, ale pouze očima jeho uživatele.
- Následuje výklad vlastností obvodové aritmetiky, jako třeba sčítačky, komparátory a násobičky. Uvedou i vhodné převody pomalého dělení na násobení.
- Učebnici uzavírá kapitola o synchronních obvodech, které sice nemají komplikovanou podstatu, ale řada posluchačů se s nimi setkává poprvé. Jejich správné užití bývá někdy náročnější na vstřebání, aspoň podle mých zkušeností s výukou.

Učebnice „Logické obvody na FPGA“ tvoří odrazový můstek pro návrhy, v nichž lze pokračovat jakýmkoli HDL jazykem. Na naší katedře Řídicí techniky jsme zvolili VHDL, který pokládáme za výhodnější pro začátečníky. Jemu se věnují další naše skripta.

Pokud se někdo rozhodně pro Verilog nebo SystemVerilog, pak najde řadu učenic od jiných autorů.

Proč se učit logické obvody?

Význam logiky závisí na našem budoucím odborném směrování. V prostředí osobních počítačů vystačíme s jejími minimálními znalostmi, neboť budeme logikou tvořit leda podmínkami, třeba rozhodovacími příkazy typu *if-then-else* či *while*.

Odlišná situace nastává, využíváme-li nestandardní výpočetní systémy, třeba při vývoji dronů či jiných experimentálních zařízeních. Hodí se zde vědět, jak vlastně elektronika funguje a co od ní můžeme očekávat. Použijeme-li i nějaký procesor určený pro aplikace v experimentálních zařízeních, pak musíme k němu někdy připojit i svoje periférie, a ne ke všem existují sériově vyráběné obslužné moduly.

Tok dat může vypadat třeba takto:

1) technické zařízení vstupu či výstupu;
2) **k němu připojené logické obvody čtoucí/zapisující/předzpracovávající data;**
3) **rozhraní sběrnice procesoru;**
4) ovladač (driver) operačního systému;
5) uživatelský program.

Připojení periférie si musíme buď vyřešit sami, nebo zadat někomu návrh, avšak i zde se hodí vědět, co lze v obvodu realizovat. Obsluha rozhraní se často spojuje s **předzpracováním dat**.

Některé algoritmy se dají konvertovat na zapojení v obvodu, který pracuje rychleji než jejich výpočet v procesoru. Zde můžeme třeba uvést filtrace obrazového signálu či FFT, rychlou Fourierovu transformaci. Podobná obvodová řešení se nazývají **hardwarové akcelerátory**. Ušetří výpočetní čas a uvolní prostor jiným úkolům.

Kvůli čemu se učit strukturu obvodů, když se stejně popisují v HDL?

Učím logiku už desítky let a ověřil jsem si, že většina studentů obvody nenavrhuje, ale programuje. Napodobují totiž konstrukce, které se naučili ve vyšších programovacích jazycích jako C či Java. Z nich lze opravdu něco používat, ale rozhodně ne všechno, neboť v obvodech disponujeme jinými prostředky než instrukcemi asembleru.

Návrhář by měl za HDL popisem vidět vždy vznikající obvod, a ne nějaký program. Na něj se konvertují leda ojedinělé části určené k simulaci. Vše ostatní se zapojuje, a tak se vyplatí napřed vědět, jak se náš kód realizuje, a teprve pak začít s HDL popisy.

1.1 Jak se logika realizuje?

Logická schémata se dnes již nezapojují z jednotlivých obvodů pomocí jejich pospojování drátky, to se asi dělá už jen v zábavných stavebnicích. A úplný návrh monolitického integrovaného obvodu je extrémně nákladný.

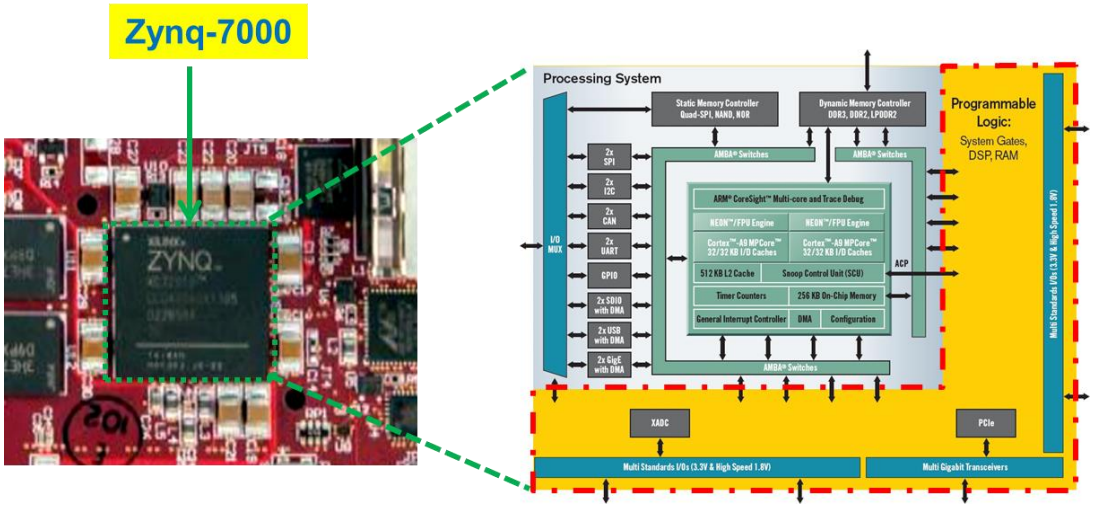
Malé série se cenově vyplatí realizovat pomocí univerzálních obvodů. Jejich velmi rozšířeným zástupcem je typ FPGA (*Field Programmable Gate Array*), jehož zkratka se do češtiny občas přepisuje jako programovatelná hradlová pole, ale častěji se nechává bez překladu. Odladňují na něm i prototypy zapojení monolitických integrovaných obvodů.

Zde musíme zdůraznit, že pojem „**programovat**“ se v době, kdy se objevily první předchůdci FPGA (cca 1983), chápal ve významu „**konfigurovat**“. Až později se svázal s programovacími jazyky[1]. A slovo „programovat“ rozhodně neladí s návrhy obvodů, které se zásadně neprogramují v dnešním slova smyslu, ale navrhují!

Zkratka FPGA by dnes měla přesnější znít třeba FCGA (*Field Configurable Gate Arrays),* avšak nejspíš se už nikdy nezmění. Zmíníme-li zavedený termín „programování FPGA“, budeme jím myslet vždy jedině to, že se do FPGA nahraje jeho konfigurace na nové zapojení.

[1] Posunul se významu více počítačových termínů. Vždyť i pojem „hacker“ se používal kolem roku 1960 k označení veleváženého odborníka na počítače. Až později získal hanlivější zabarvení, když jejich znalci začali svých vědomostí zneužívat.

Některé typy FPGA obsahují i celé procesory, třeba Zynq-7000 firmy AMD Xilinx, v němž se nacházejí dvě jádra procesoru ARM Cortex-A9. Jako příklad jeho užití můžeme uvést výukovou destičku MZAPO. Vyvinuli ji Pavel Píša a Petr Porazil z Katedry řídicí techniky Elektrotechnické fakulty v Praze. Žlutá část vnitřní struktury Zynq-7000 obsahuje volně programovatelnou logiku, pomocí níž vytvořili obsluhu periférií MZAPO.



Obrázek 1 – Zynq™-7000 na desce MZAPO (zdroj pravého obrázku Xilinx)

Podle druhu obvodu z řady Zynq-7000 lze ve žluté části nakonfigurovat desítky tisíc logických funkcí. Typ FPGA XC7z010, použitý v MZAPO, umožňoval obsluhu periférií vybudovat až s užitím 17600 logických funkcí, přičemž každá uměla šest vstupů. Doplňovalo je ještě 35200 klopných obvodů, přes 2 megabity rychlých SRAM pamětí a další podpůrné jednotky, jako hardwarové násobičky a moduly k obsluze sběrnic.

Některé typy FPGA se prodávají i bez procesorů, jen s programovatelnou logikou. Pomocí ní se lze rovněž vytvořit procesor, tzv. *soft-core processor*, který nabízí i vysokou variabilitu. A těch může být několik, třeba celé jejich spolupracující sítě.

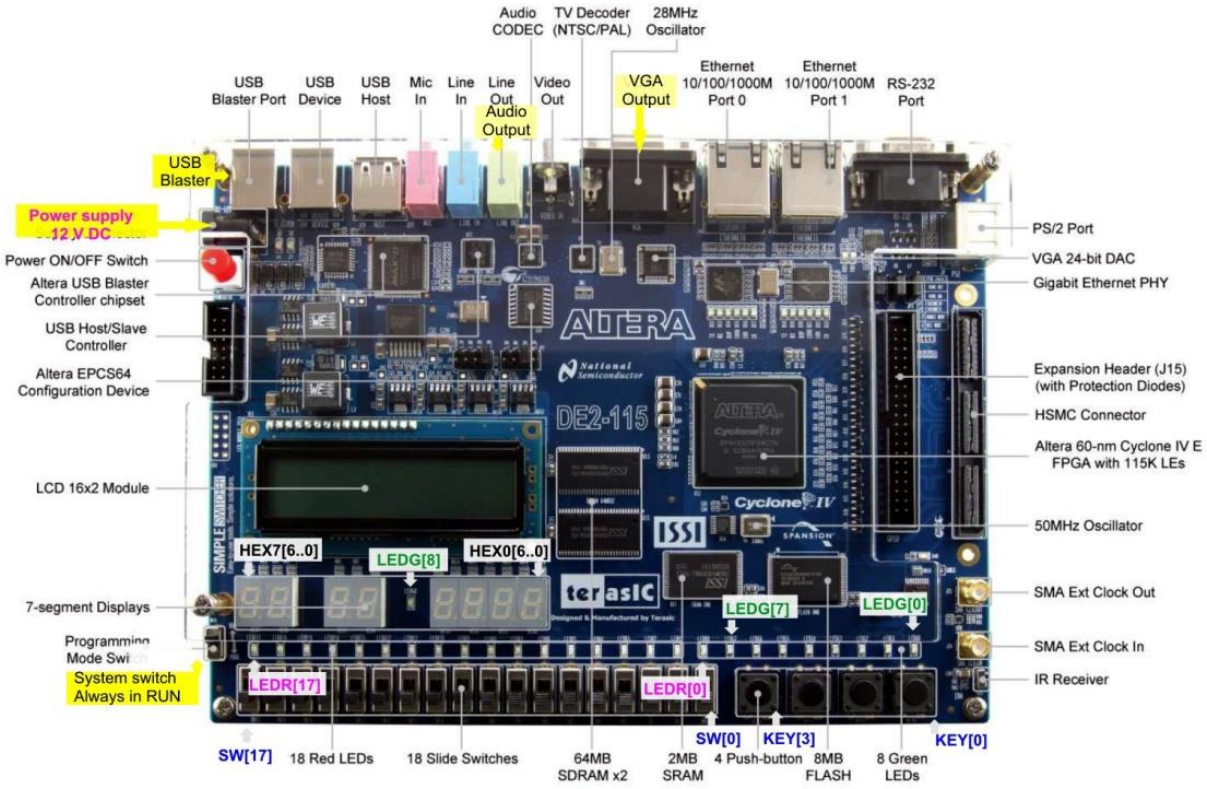
1.2 Co získáme použitím FPGA?

- FPGA součástky vznikly především k individuálním nebo malosériovým aplikacím, kde svou cenou suverénně překonávají mimořádně nákladné návrhy monolitických integrovaných obvodů. Zapojení s užitím FPGA vyjde často levněji a provede se i rychleji než tvorba plošného spoje osazeného individuální komponentami.
- Monolitický integrovaný obvod nejde opravit, najde-li se v něm chyba, plošný spoj někdy ano, ale pracně. Obvod realizovaný v FPGA lze snadno změnit. Stačí do něho nahrát opravenou konfiguraci.
- FPGA se využívá i v kosmických aplikacích pro možnost jejich dálkové korekce či vylepšení funkce. Pro ně se vyrábějí jejich speciální typy s posílenou odolností vůči radiaci.
- Častou FPGA aplikací bývají také emulátory procesorů, buď starých, které se už nedají sehnat, nebo typů ve vývoji, aby se odladily jejich funkce. Emulátor pak dovolí i vývoj softwaru ještě před ostrou sérií výroby.
- Žádné FPGA nepředběhne monolitické obvody stejného stupně integrace jak svým výkonem, tak hustotou skutečně užitých elementů, neboť komponuje zapojení jen na úrovni logických funkcí. Níže se nemůže již ponořit. Monolitické obvody rozkládají operace až na jemnou úroveň tranzistorů, což jim dovolí konstrukce nedostupné v FPGA.

- Existují ale úlohy, v nichž FPGA realizace předčí i výkon vícejádrových procesorů nebo grafických karet, třeba běh již hotové neuronové sítě, ta se na FPGA totiž zapojí paralelně, tedy právě tak, jak ve své podstatě funguje. V mnohých úlohách se jim naopak nevyrovná, např. proces učení neuronové sítě. Musíme tedy rozeznat, co se vyplatí řešit v FPGA, a co ne. I o tom bude učebnice, kterou právě čtete.
- Zapojení realizovaná na FPGA mají ve srovnání s klasickými programy nevýhodu náročnější tvorby. Zdrojový kód programu se ladí rychleji. Zvýšená pracnost obvodového řešení dané úlohy, nebo její dílčí části, přinese však razantní výhodu. Ušetří nejen čas procesoru, ale také odběr ze zdroje, což bude příznivé zejména u elektroniky napájené z baterií.

FPGA se prodávají jak samostatně, tak na vývojových deskách, které se dají ihned používat a lze je přímo zabudovat do koncových či uživatelských zařízení.

Na ukázku jsme napřed vybrali DE2-115 část vývojové desky Terasic VEEK-MT2, která se vyvinula k výuce. Používá se nejen na naší fakultě, ale i na stovkách předních světových univerzit. Obsahuje mnoho přídavných elementů vhodných k řešení studentských úloh. Jejím jádrem je FPGA typu Intel EP4CE115F29, o němž bude více na str. 88.

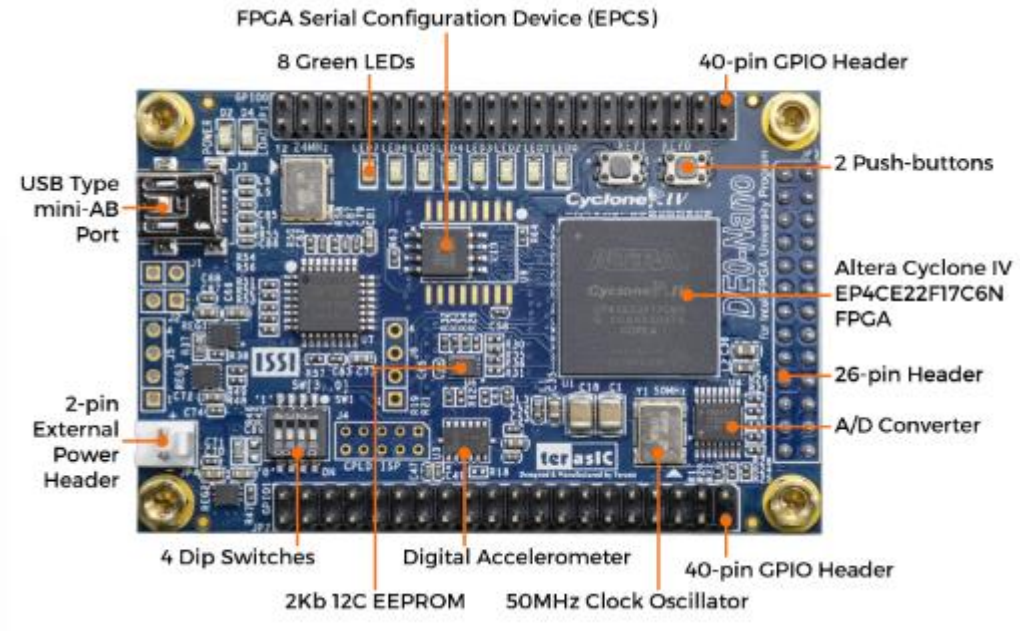


Obrázek 2 - DE2-115 část vývojové desky VEEK-MT2 (převzato od Terasic)

Experimentálním zařízením postačí jednodušší desky bez přídavných výukových elementů, třeba vývojová DE0 nano, jejíž cena začíná na $87 (v roce 2024). Hodí se k přímému zabudování i do dronů.

Obsahuje FPGA typ EP4CE22F17, který nabízí totožné konfigurovatelné elementy jako Cyclone IV z předešlé desky DE2-115, jen některých má méně (od 1/8 do 1/4), avšak sám o sobě přijde ~ $4, tedy na patnáctinu ceny svého pokročilého FPGA kolegy, a lze u něho též využívat bezplatnou verzi vývojového prostředí Quartus.

Pozn. V obrázku dole je za identifikací typu FPGA ještě sufix „C6N“ coby interní kód výrobce specifikující rychlostní třídu obvodu, speed grade. *Zde C6N znamená nejrychlejší.*



Obrázek 3 - Přední strana vývojové desky DE0 nano (Převzato od Terasic)

1.3 Historie textu

Fakulta Elektrotechnická (FEL), součást ČVUT v Praze, vyučuje předměty „Architektura počítačů“ (APO) a „Logické systémy a procesory“ (LSP), na nichž se podílí moje domovská Katedra řídicí techniky.

Během jejich mnohaleté výuky jsem vytvořil řadu návodů. Z nich jsem postupně vybíral nejdůležitější témata, která jsem rozšiřoval. Vznikly učebnice dovolující udržet výuku na snesitelné úrovni a přitom v širším rozsahu, avšak stále srozumitelnou všem. Nelze do vysokoškolských přednášek zahrnout pasáže určené naprostým začátečníkům. Výklad trivialit zaměřený pouze na ně by sebral čas zajímavějším částem a zkušenější studenti by se nudili.

Přednášky se učiní atraktivnější, budou-li předpokládat znalost lehčích částí, nenáročných na pochopení, k nimž stačí jen přečtení textu. Ve výkladu se především přiblíží hlavní témata. Zvídavější posluchači si je mohou sami rozšířit z textů, které se sice primárně vytvořily k postupnému čtení, ale lze je samozřejmě studovat i paralelně, kousek z jedné, pak něco z další, či je používat jako pouhé příručky.

V roce 2012 jsem kvůli tomu napřed vytvořil prerekvizitu **APOLOS**, která shrnovala minimální vstupní znalosti nutné k absolvování LSP a APO. Její první polovina rozebírala jednoduché pojmy z logických obvodů a druhá pak kódování čísel.

V roce 2019 jsem dopsal další učebnici uvádějící do VHDL stylu *concurrent*, na níž jsem v roce 2021 navázal dalším dílem věnovaným VHDL stylu *behavioral*, ale vysvětloval jsem v ní jak VHDL, tak prvky obvodů. Její stránky narůstaly a blížily se už ke stovce, a pořád se neobjasnily potřebné věci. A její přehlednost se snižovala mísením obvodové techniky s výkladem VHDL.

Rozhodl jsem přeuspořádat své studijní materiály. První polovinu APOLOS jsem přesunul na začátek nové učebnice „**Logické obvody na FPGA**“, kterou jsem rozšířil o aplikační části. Za ně jsem vložil obvodové techniky vyjmuté z nedokončené VHDL učebnice stylu *behavioral*.

Doplnil jsem k nim pasáže ze základních znalostí logiky, aby vznikla ucelená učebnice k předmětu LSP. Na přelomu roku 2024 učebnice rozšířila o posuvné registry a ukázky kódu ve VHDL a Verilogu a začátkem roce 2005 o sériové linky.

Učebnice se věnuje struktuře obvodů, a tak se hodí i jinde, třeba v dalších kurzech naší fakulty, v nichž se používá Verilog místo VHDL.

Části o kódování čísel v APOLOS se nezměnily. Nyní se jen osamostatnily, takže obsahují pouze látku společnou jak LSP, tak APO. Změna zpřehlednila i následné učebnice, v níž se nyní probírají VHDL styly kódu bez zdlouhavých vsuvek o obvodové technice.

Učebnice o VHDL se v současné době konvertuje na webové stránky, které nabízejí flexibilnější přístup k údajům.

1.4 Jazyková poznámka k textu a obrázkům

Většina odborné literatury se dnes píše v angličtině, a novějších českých textů bývá poskrovnu. K řadě pojmů tak neexistují ustálené překlady a jejich vymýšlením bych jen ztížil orientaci v cizojazyčných publikacích.

Raději jsem vkládal **anglické termíny,** *technical terms*, psané kurzívou přímo do českého textu. U nich zmiňuji i domácí ekvivalent v případě, že se mi podařilo dohledat aspoň trochu zavedený pojem v dostupných článcích.

Učebnice se bude dobře číst i studentům, kteří disponují spíše vizuální pamětí, protože obsahuje skoro 240 původních kreseb, avšak jen 190 je jich číslovaných, jelikož mnohé se vkládaly bez titulků, pokud jen doplňovaly text. V obrázcích se používaly popisky v angličtině, a to ke zrychlení překladu určeného zahraničním studentů naší fakulty.

Objeví se žel i četná spojení typu „**na Obrázek 10**“. Kvůli značnému množství vkládaných obrázku se nevyužil LaTeX, ale editor MS-Word, který však neumožňuje u všech typů křížových odkazů vložit jejich pouhé číslo, aby se návěští dalo skloňovat.

Některé stránky mají na svém dolním konci **delší volné části**. Nezanedbalo se u nic formátování dokumentu, ale vkládaly se úmyslně, aby se synchronizovalo stránkování mezi českou a anglickou verzí učebnice. Externí odkazy mají tak stejné reference v obou jazykových verzích učebnice.

1.5 Poděkování

Děkuji všem, kteří svými připomínkami a radami přispěli ke zlepšení učebnice.

Vývojáři z praxe Ing. Jaroslav Houdek a Ing. Jan Kelbich mi ochotně provedli její odbornou korekturu, při níž našli řadu ostudných prohřešků. Patří jim moje vděčnost.

Rád bych také ocenil své studenty, kteří využívali dosud nedokončené verze učebnice, a posílali mi k opravení překlepy a chyby, a doktora Pavla Píšu za jeho připomínky ke kapitole 7.7.

Určitě v učebnici zůstaly i neodhalené nedostatky a uvítám upozornění na ně.

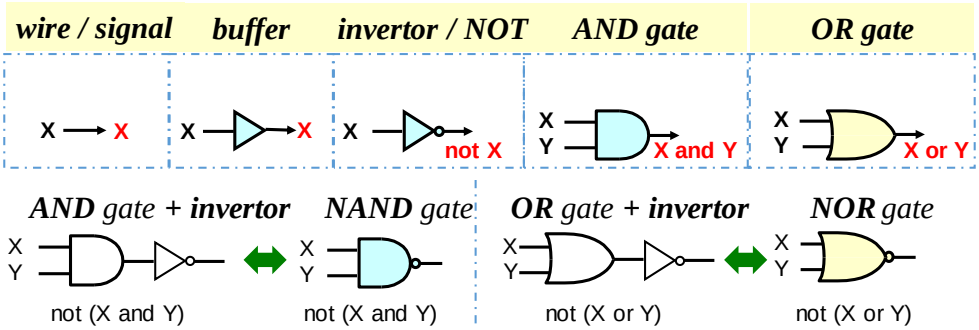
Richard Šusta

2 Logické funkce

Všichni určitě znáte logickou '1' (TRUE) a logickou '0' (FALSE), a to minimálně z programovacích jazyků, v nichž existují typy nazývané Boolean či bool a logické funkce jako unární NOT (negace), operace AND (logický součin) a OR (logický součet).

V logických schématech se reprezentují grafickými prvky, jimiž se popisuje strom vyhodnocení výrazu. Jeho uzly tvoří hradla, *gates*, která nám realizují elektronické prvky. Ty provádějí zadanou operaci a na svůj výstup posílají hodnotu stanovenou ze svých okamžitých vstupů. Schéma tak popisuje tok dat v hardwaru.

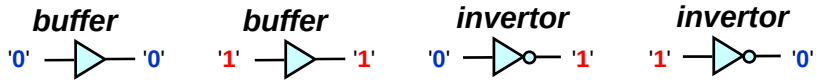
Operace NOT se kvůli zkrácení často vyznačuje jen bublinkou na výstupu hradla.



Obrázek 4 - Základní logické operace a jejich symboly

Pro další text zavedeme uspořádání logických hodnot předpisem **'0' < '1'**, slovy logická '0' je menší než logická '1'. Pomocí něho si lehce zapamatujeme základní operace logiky:

- **Prvek *buffer*** či ***wire*** nebo **signál** (cz: budič?) kopíruje vstupní hodnotu na výstup. Jde-li o pouhé spojení, fyzicky se realizuje vodičem, odtud i *WIRE*. Použije-li se elektronický prvek kvůli oddělení nebo k získání vyššího výstupního proudu či ke změně úrovně napětí, pak se skutečnost vyznačí pojmem *BUFFER*.
- **Logická funkce *NOT***, realizovaná hradlem typu invertor, angl. *invertor* nebo *negation* či *complement*, mění minimum '0' na maximum '1', a maximum '1' na minimum '0'.



- **Logická funkce AND** posílá na svůj výstup logickou '1' jen tehdy, když oba její vstupy jsou v logických '1'. Provádí tedy **výběr minimální hodnoty vstupů**, tj. bude-li jakýkoliv její vstup v logické '0', pak na výstup pošle minimální hodnotu '0'.
- **Logická funkce OR** má svým způsobem inverzní k funkci AND. Na jejím výstupu bude logická '0' pouze v případě, pokud oba její vstupy jsou v logických '0'. Realizuje tím **výběr maximální hodnoty vstupů**, tj. bude-li jakýkoliv její vstup v logické '1', pak maximální hodnota bude '1'.

Zapamatujte si:

- AND, **výběr minima**, se rovná '1' pouze při jediné kombinaci svých vstupů, když jsou všechny v logických '1', tedy v maximech.
- OR, **výběr maxima**, se rovná '0' pouze pro jediné kombinaci svých vstupů, když jsou všechny v logických '0', tedy v minimech.

Chápání logických funkcí AND a OR jako výběrů minima a maxima dovoluje jejich přímočaré rozšíření na libovolný počet vstupů.

- **Logická funkce AND s n vstupy**, F=and(x_{n-1},...,x_1, x_0), vybírá minimum z hodnot všech svých n vstupů. F bude v logické '**1**' tehdy a jen tehdy, pokud má všechny své vstupy x_i = '**1**'. Bude-li na jednom nebo na více vstupech logická '**0**', pak minimum je '**0**'.
- **Logická funkce OR s n vstupy**, G=or(x_{n-1},...,x_1, x_0), vybírá maximum z hodnot všech svých n vstupů. G bude v logické '**0**' jen tehdy, budou-li všechny vstupy x_i = '**0**'. Bude-li logická '**1**' na jednom vstupu nebo na více vstupech, pak maximum bude '**1**'.

Pokud člen obsahuje všechny proměnné z nějaké zadané množiny, pak se mu říká **minterm** při jejich spojení operací AND a **maxterm** při jejich spojení OR. *Pozn. Pojem si na str. 37 rozšíříme na obecnější termín implikant.*

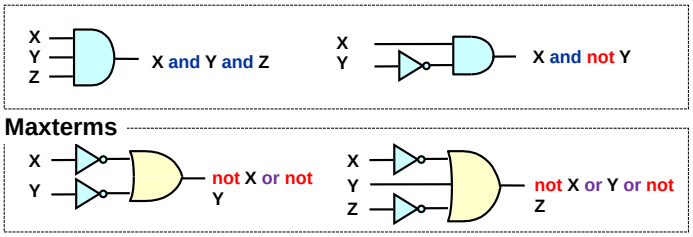
Příklady:

(X and Y and Z) - je **minterm**, který nabývá '**1**' jen při všech vstupech v '**1**', jinak je '**0**'.

(X and not Y) - je **minterm** dávající '**1**' při X='**1**' a Y='**0**' (not Y='**1**').

(not X or not Y) - je **maxterm**, který nabývá '**0**' jen pro X='**1**' a Y='**1**', jinak je '**1**'.

(not X or Y or not Z) - je **maxterm** dávající '**0**' jen pro X='**1**', Y='**0**' a Z='**1**', jinak je '**1**'.

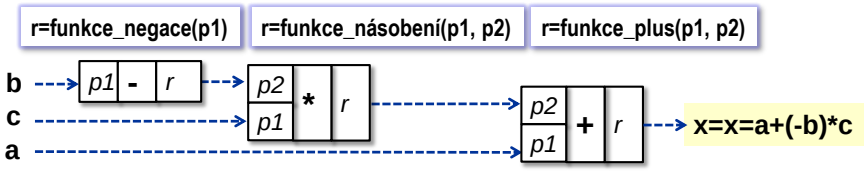


Obrázek 5 - Realizace mintermů a maxtermů

Minterm lze zapojit jedním hradlem AND, zatímco maxterm hradlem OR.

2.1 Operátory a logické funkce

Vezme-li běžný matematický výraz, například x=a+(-b)*c, pak syntaktický analyzátor (slangově *parser*) musí zkratkovité operátory, usnadňující zápis, konvertovat na zřetězené volání funkcí dle jejich priority. Jeho výsledek můžeme vyjádřit výrazovým stromem:

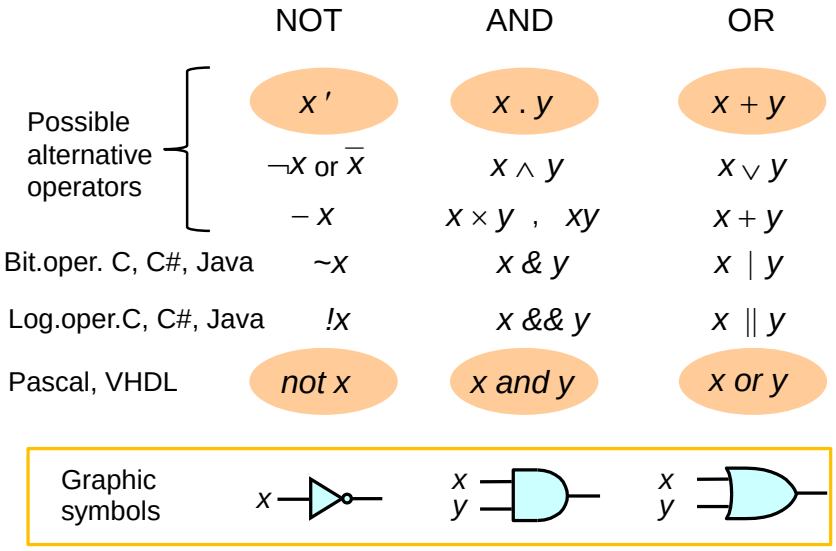


Matematicky se strom zapíše: x = funkce_plus(**a**, funkce_násobeni(**c**, funkce_negace(**b**))). Unární funkce funkce_negace má jeden vstupní parametr p1 a vrací výsledek r (*result*), zatímco zbylé funkce jsou binární, tj. mají dva vstupní parametry p1 a p2.

2.1.1 Logická schémata

Logickou funkci lze zapsat jak výrazem, tak ji graficky vyjádřit logickým diagramem či schématem, jímž se vyznačí postup jejich vyhodnocení. Operace se tady vyznačují grafickými symboly použitých prvků, které jsou vzájemně propojené.

Z technických důvodů (znaky na klávesnici počítače) se v logickém výrazu zapisují operace AND často **.** a OR symbolem **+**. Unární NOT se vyznačuje postfixovým apostrofem. Ke srovnání lze uvést přehled různých zápisů NOT, AND a OR, tedy zavedených formalit.



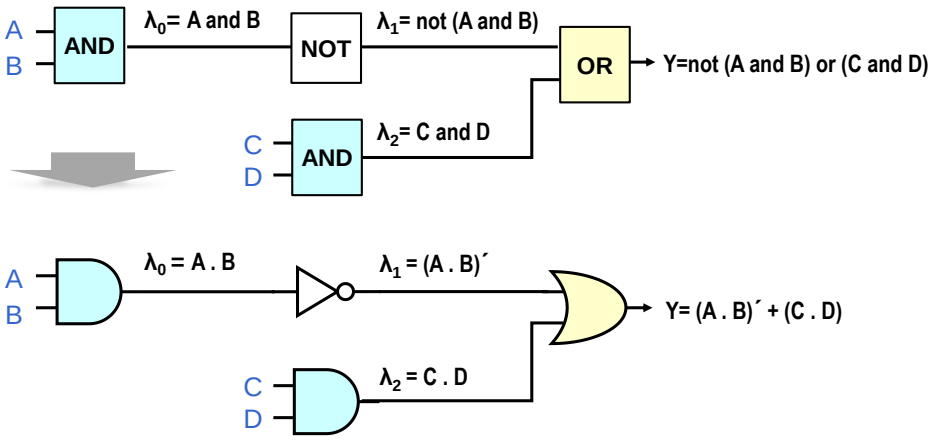
Obrázek 6 - Operátory logických operací

Zde si dovolíme připomenout, že v programovacích jazycích C, C# a Java se členy logických operací !, && a || vyčíslují jen tak dlouho, dokud není jasná hodnota výsledku. Bitové operátory se evaluují celé, čímž se víc podobají logice, v níž souběžně pracuje každý její prvek.

Vezmeme-li například logickou funkci Y= (not (A and B)) or (C and D). Jelikož unární operace mají obecně vyšší prioritu než binární operace, vynecháme tučně vyznačené červené závorky a napíšeme funkci jako Y= not (A and B) or (C and D), respektive i pomocí zkratek operátorů "+" , "." a "´" (apostrof značí negaci), jako Y=(A . B)´ + (C . D).

Její vyhodnocení může začít třeba levou operaci AND: λ_0=A.B [λ_0=A and B], kde λ_0 označuje její mezivýsledek. Poté se provede jeho negace λ_1=(A.B)´ [λ_1=not (A and B)]. Dále se spočte další operace AND: λ_2=C.D [λ_2=C and D]. Nakonec se oba mezivýsledky λ_1 a λ_2 spojí pomocí operace OR na Y=λ_1+λ_2 = (A.B)´ + (C.D) [Y= not (A and B) or (C and D)].

Postup vyhodnocení ukazuje Obrázek 7. Nahoře jsou jednotlivé operace zapsané jmény logických funkcí, avšak dole je stejné schéma nakresleno mnohem častějším způsobem pomocí schematických značek pro logické operátory, tedy logická hradla (angl. *logic gates*).



Obrázek 7 - Logické schéma a jeho logický výraz

2.2 Zákony Booleovy logiky

Booleova logika obsahuje dva prvky, nám známé logické '**0**' a '**1**', a dále dvě binární operace AND a OR a jednu unární NOT.

Zde musíme mít na paměti:

1. V Booleově logice mají obě operace AND a OR **totožnou prioritu!** Řada programovacích jazyků zjednodušuje zápis výrazů tím, že dává operaci AND prioritu vyšší než OR. Nesmíme tohle předpokládat v Booleově logice, jinak dostaneme chybné výsledky. AND a OR mají v ní shodné priority. Je vhodné doplnit závorky.

   Příklad: Předchozí funkce Y (Obrázek 7 na str. 14) by se v jazyce C zapsala pomocí příkazu: Y=!(A && B) || C&&D. V Booleově logice se priorita musí vyznačit závorkami a napsat Y=(A.B)' + (C.D), nebo slovními operátory: Y = not (A and B) or (C and D).
2. Booleova logika zná jen '**0**' a '**1**'.
3. Rozšířením Booleovy logiky je Booleova algebra definovaná nad větším počtem logických hodnot, tedy nad více prvky než '**0**' a '**1**'. Operace NOT, AND a OR se v ní definují tabulkami. Při návrzích se běžně používá až devět hodnot. Jednu další výstupní hodnotu '**Z**' rozebereme na str. 64 při výkladu třístavového hradla.

V textu zatím zůstaneme u Booleovské logiky, tedy u pouhých dvou hodnot '**0**' a '**1**'.

Booloeva logika splňuje Huntingtonovy postuláty. Ty se přijímají bez důkazů jako její teoretický základ a specifikují minimální požadavky na to, aby vůbec šlo o Booleovu logiku. Z nich lze pak odvodit další teorémy.

V praxi se téměř nepoužívají náročnější algebraické úpravy logických funkcí, a to kvůli jejich nepřehlednosti, která zvyšuje riziko omylu. Existují bezpečnější metody. Základní pravidla Booleovy logiky mají však svůj nezastupitelný význam při tvorbě logických obvodů. Ukážeme si hlavně jejich aplikace, tedy, co nám daný teorém či postulát obvodově dovoluje.

Prvním postulátem je **Uzavřenost** (eng. *Closure*), tedy výsledkem jakýchkoli operací s logickou '**0**' a '**1**' bude v Booleově logice opět '**0**' nebo '**1**', nic jiného se v ní neobjeví.

Hned dalším postulátem je **komutativita**.

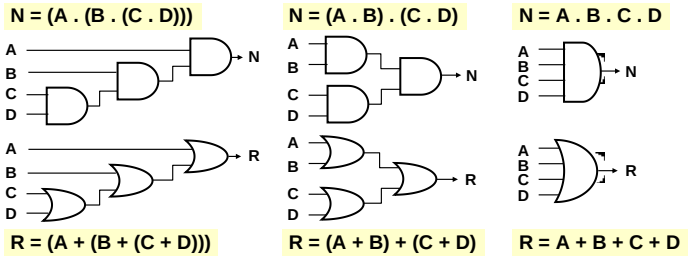
Postulát	*OR verze*	*And verze*
Komutativita *Commutative Law*	$x + y = y + x$	$x \bullet y = y \bullet x$

U vícevstupových hradel nebude tedy záležet na pořadí, v jakém zapojíme signály na jejich vstupy. Výsledek operace bude pokaždé stejný.

V programovacích jazycích se může někdy odvíjet výsledek od pořadí, v jakém se uvedou členy v komutativním výrazu, a to kvůli vedlejším efektům, jelikož se hodnoty členů výrazu vyčíslují postupně. V logických obvodech ale vše běží **souběžně**, což je jejich základní vlastností. Všechny komponenty pracují paralelně spolu s ostatními, jako kdyby každá z nich běžela na samostatném jádře procesoru. Aplikace emulující činnost obvodů tohle napodobují třeba frontou událostí, někdy zpracovávanou i v náhodném pořadí, třeba ModelSim.

Pokud do postulátu komutativity substituujeme výrazy místo proměnných x a y, pak dostaneme teorém **asociativity**.

Teorém	*OR verze*	*And verze*
Asociativita *Associative Law*	$a + (b + c) = (a + b) + c$	$a \bullet (b \bullet c) = (a \bullet b) \bullet c$

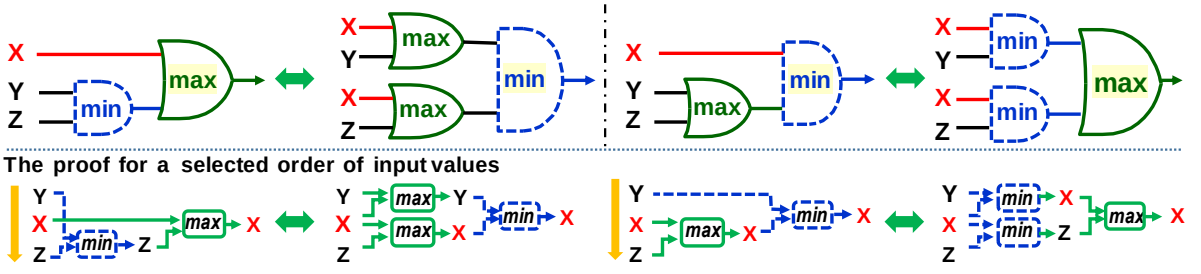


Obrázek 8 - Asociativita

Máme-li k dispozici hradla s menším počtem vstupů, než právě potřebujeme, pak jejich pospojováním vyrobíme vícevstupové hradlo OR nebo AND. Z hlediska logické operace bude úplně jedno, jak jejich vrstvení provedeme. Asociativní rozklad zas naopak urychlí operace, jak si později ukážeme na sčítačce a odčítačce konstanty 1 v kapitole 6.1.2 na str. 103.

Propojení do kaskády, na obrázku vlevo, se nemusí ani vyhodnocovat pomaleji kvůli delší cestě od vstupu D na výstup N či R. Návrhové prostředí, které bude stát mezi naším popisem obvodu a jeho realizací uvnitř FPGA, minimalizuje zadané výrazy a v závěru implementuje i úplně jinak, ale se shodnou funkcí. Všechny způsoby výpočtu R i N, které se uvedly nahoře, dávají námi požadovaný výsledek, což je nejdůležitější ze všeho.

Postulát	*OR verze*	*And verze*
Distributivita *Distributive Law*	$x + (y \bullet z) = (x + y) \bullet (x + z)$	$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$



Obrázek 9 - Distributivita

Symboly ● a **+** jinde označují aritmetické násobení a sčítání, která nejsou vzájemně distributivní, takže teorém distributivity vypadá nepřirozeně z jejich pohledu. V logice se jimi zapisují operátory výběru minima (AND) a maxima (OR), které rovnocenné postavení.

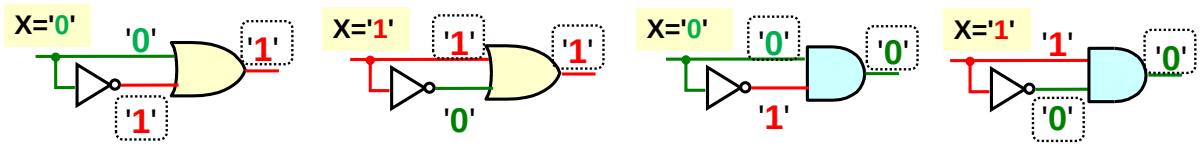
Důkaz lze provést třeba dosazením všech šesti možných případů[2], které existují pro velikosti tří vstupních hodnot X, Y a Z.

Výpočet dole na obrázku demonstruje jeden z nich, náhodně vybraný, kdy Y má největší hodnotu a Z nejmenší. Uvažovali jsme v něm Booleovu algebru s více hodnotami ke zdůraznění, že distributivita operací minimum a maximum platí nejen u dvouhodnotové Booleovy logiky,

[2] Důkaz, viz třeba: https://proofwiki.org/wiki/Max_and_Min_Operations_are_Distributive_over_Each_Other

ale v jakémkoli oboru, v němž je definované uspořádání na základě zavedení ≤ relace, třeba i u reálných čísel.

Postulát	*OR verze*	*And verze*
Komplementarita *Complementation*	$a + a' = 1$	$a \bullet a' = 0$

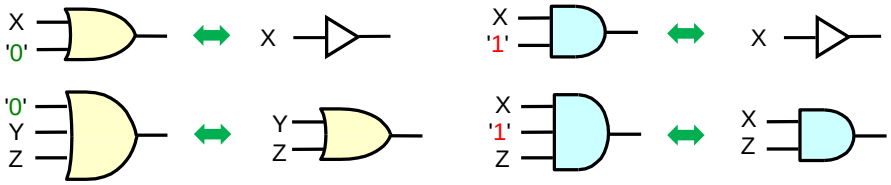


Obrázek 10 - Komplementarita

Při současném přivedení vstupu a jeho negace na hradlo, tedy vstupu a jeho komplementárního doplňku, bude vždy jedna hodnota v logické '0' a druhá v '1'. OR operace (výběr maxima) z nich vybere pokaždé maximum '1', což bude jejím konstantním výstupem, zatímco AND jako výběr minima nám bude dávat vždy konstantu minimum '0'.

Komplementaritu využijeme později k minimalizaci logických funkcí.

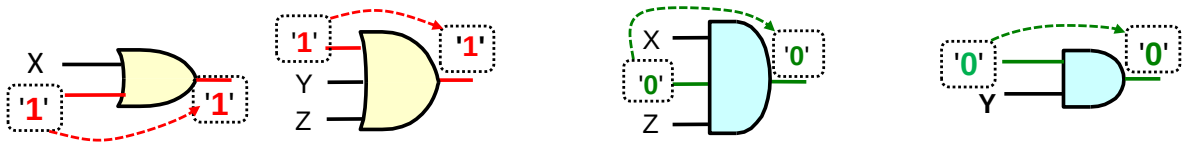
Postulát	*OR verze*	*And verze*
Neutralita *Identity Law*	$x + '0' = x$	$x \bullet '1' = x$



Obrázek 11 - Neutralita

Neutralita specifikuje vlastnost operací výběru **maxima** (OR). Je-li některý vstup na minimu, tedy v '0', pak neovlivní výsledky z dalších vstupů. Analogicky u výběru **minima** (AND) nemůže mít vstup v maximu, tedy v '1', vliv na výslednou hodnotu.

Teorém	*OR verze*	*And verze*
Agresivita *Annulment Law*	x + '1' = '1'	x ● '0' = '0'



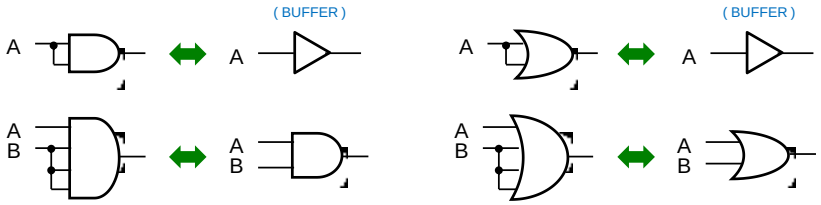
Obrázek 12 - Agresivita

Agresivita rovněž plyne přímo z funkcí AND a OR. Bude-li jeden vstup OR v maximální hodnotě '1', pak výstup bude vždy '1' bez ohledu na další vstupy. Již se dosáhlo možného maxima. Analogicky při vstupu v minimu, tedy v '**0**', má AND již zvolenou hodnotu minima, tedy '0', a další vstupy již neovlivní výběr.

Zapamatujte si:

- **AND**, výběr **minima**, má neutrální vstupní hodnotu maximum '**1**' a agresivní minimum '**0**'.
- **OR**, výběr **maxima**, má neutrální vstupní hodnotu minimum '**0**' a agresivní maximum '**1**'.

Teorém	*OR verze*	*And verze*
Idempotence *Idempotent Law*	x + x = x	x ● x = x

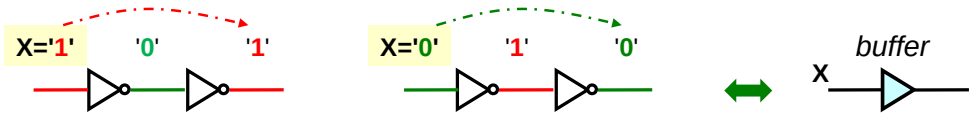


Obrázek 13 - Idempotence

Operace AND a OR je obě idempotentní, tedy opakovaným použitím stejného vstupu vznikne stejný výstup podobně jako jeho jediným použitím.

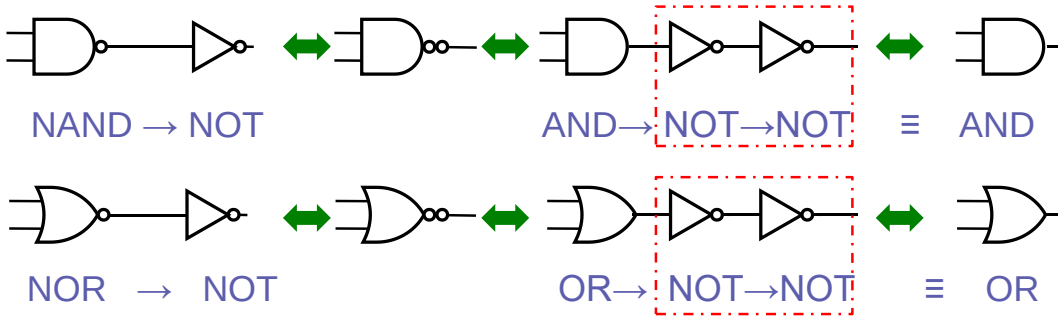
V praxi můžeme vícevstupové hradlo využít i jako člen s méně vstupy. Zapojíme vstupní signál na více výstupů. Případně můžeme nadbytečné vstupy připojit i na neutrální prvek dané operace, u AND na '1' a u OR na '0'. Výsledek bude totožný. Závisí na nás, co se nám líbí.

Teorém	
Dvojí negace *Double negation*	not (not x) = x



Obrázek 14 - Dvojí negace

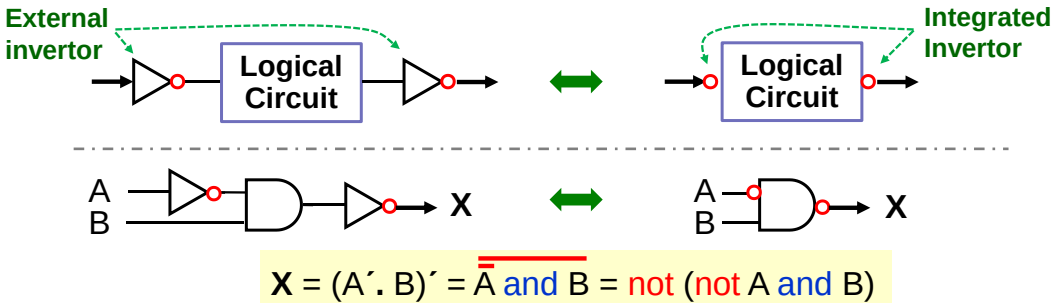
Dvojí negace se anulují, a tak se jejich spojení chová, z hlediska čisté logiky, jako *buffer*. Invertor se často zkracuje bublinkou na výstupu.



Obrázek 15 - Dvojí negace u hradel

Dvojí negace se využívá k manipulaci s hradly, zejména ve spojení s De Morganovým teorémem, který bude dále.

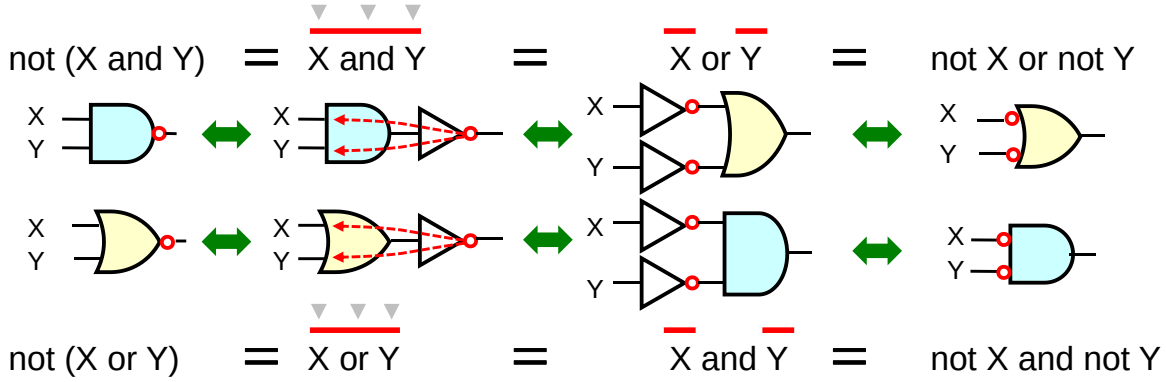
Bublinka nahrazuje někdy i s invertory, které leží před vstupem či za výstupem funkce, a to zejména ve schématech, v nichž se šetří místem.



Obrázek 16 - Úsporné bublinkové značky invertorů

Teorém	*OR verze*	*And verze*
De Morgan	not (x + y) = not x ● not y	not (x ● y) = not x + not y

De Morganův teorém patří k nejčastěji uplatňovaným operacím, neboť dovoluje rozepsat NOT před závorkou. Jeho platnost lze dokázat několika různými cestami. Nejsnáze si teorém ověříme, pokud si vytvoříme pravdivostní tabulku logické funkce na jeho levé i pravé straně.



Obrázek 17 - DeMorganův teorém

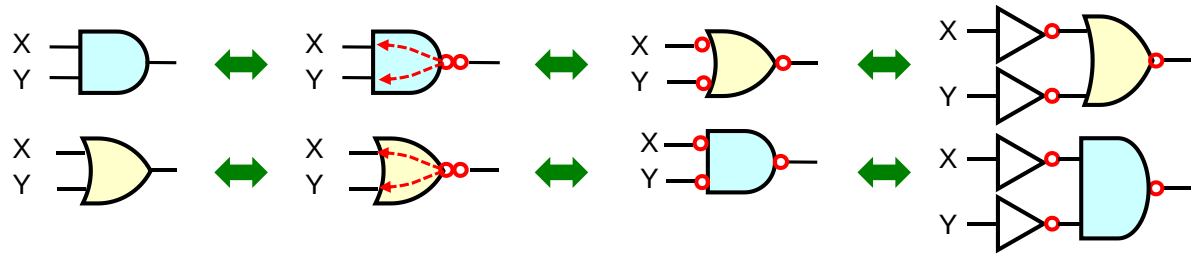
De Morganův teorém umí rovněž změnit typ logické operace. V počátcích logických hradel, první se vyráběla od roku 1963, se operace NAND realizovala snáze na více emitorových transistorech tehdejší TTL logiky. Kvůli tomu se veškeré logické funkce konvertovaly pomocí vkládání dvojitých negací a jejich rozepisováním tak, aby obsahovaly pouze invertory a NAND hradla, ale analogickým postupem lze samozřejmě převést zapojení i na výlučné užití NOR.

V návrzích určených FPGA nemusíme již provádět konverze typů hradel. Vývojová prostředí, ve kterých se náš logický popis překládá, si stejně vstupní návrh přetvoří podle jim dostupných koncových prvků.

Změna typu hradla se však hodí na úrovni integrovaných obvodů, kde se snažíme vytvářet hradla s negací výstupu, která se klopí rychleji, jak bude rozebrané dále v kapitole 4, věnované jejich interní CMOS struktuře.

Princip úpravy spočívá v symbolickém posunutí bublinky negace skrz AND/OR hradlo. Jeho typ se jejím průchodem změní na opačný a vstupy/výstupy se invertují. Bublinka negace musí však vždy vycházet ze všech jeho vstupů, či na nich končit. Při úpravách můžeme kdykoli přidat dvě bublinky za sebou, pokud se nám to hodí, neboť dvě negace se vzájemně ruší dle

teorému o dvojí negaci, viz str. 18, a jednu bublinku posunout skrz hradlo. Postup je grafickou aplikací De Morganova teorému.



Obrázek 18 - Konverze mezi hradly AND a OR

U výrazu musíme nutně dodržet pořadí provádění logických operací. Nechť máme test shody tří bitů. Všechny musí být v logické '1' nebo v logické '0'. Funkci napíšeme snadno:

EQ3(X,Y, Z) = X . Y . Z + X' . Y' . Z'

Zde máme však **zavádějící zápis**! V Booleově logice, i algebře, mají AND a OR stejnou prioritu. Předchozí výraz napodobuje programovací jazyky, v nichž se uměle zavádí preference AND nad OR, aby se uživatelům psaly snáze výrazy. Raději specifikujeme pořadí operací závorkami a změníme post-fixovou negaci na unární not operátor.

Pro další výklad si napíšeme výraz s použitím jednoznačných slovních názvů operátorů:

EQ3(X, Y, Z) = (X and Y and Z) or (not X and not Y and not Z) (1)

Nyní již vidíme možnost dostat not před zvýrazněnou závorku. POZOR, v Booleově logice nelze ale vytknout unární operaci not. Výraz not (X and Y and Z) ≠ (not X and not Y and not Z)!

Operaci not odstraníme správně tak, že před (vložíme dva not operátory a druhý rozvedeme aplikací De Morganova teorému, který změní and na or a vyruší se negace členů (podle teorému o dvojí negaci):

EQ3(X, Y, Z) = (X and Y and Z) or not not (not X and not Y and not Z)

= (X and Y and Z) or not (X or Y or Z) (2)

Negovanou funkci NEQ3 vytvoříme i pouhým doplněním negace, kterou si rozepíšeme. Pozor, De Morganův teorém aplikujeme na členy výrazu, jimiž jsou zde logické funkce! Před maxtermem se vyruší dvě not podle teorému o dvojí negaci (str. 18):

NEQ3(X,Y) = not ((X and Y and Z) or not (X or Y or Z)) (3)

= not (X and Y and Z) and not not (X or Y or Z)

= not (X and Y and Z) and (X or Y or Z) (4)

Mohli bychom i levý člen vztahu (4) rozepsat De Morganovým teorémem na:

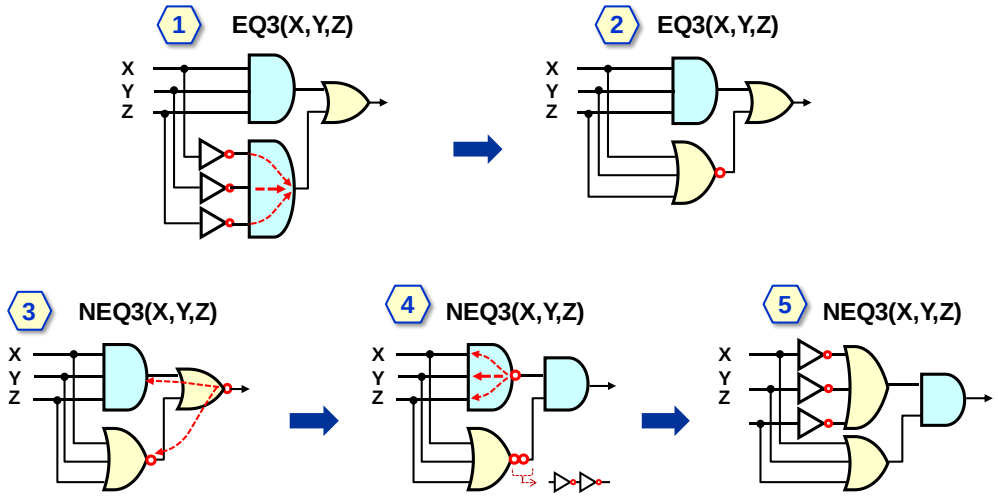
NEQ3(X,Y) = (not X or not Y or not Z) and (X or Y or Z) (5)

Výraz by se tím komplikoval dalšími invertory, raději ho necháme ve tvaru (4).

Platnost vztahu (4) si snadno ověříme i úvahou

- Pouze při X='1', Y='1' a Z='1' bude maxterm (X and Y and Z) v '1', tudíž jeho negace v '0', a tak též celá NEQ3 podle teorému o agresivitě '0' vůči operaci AND.
- Naproti tomu jedině při X='0' a Y='0' a Z='0' bude maxterm (X or Y or Z) v '0', tudíž i NEQ3.
- Ve všech ostatních případech bude NEQ3 v '1', čímž hlásí, že tři bity nejsou shodné.

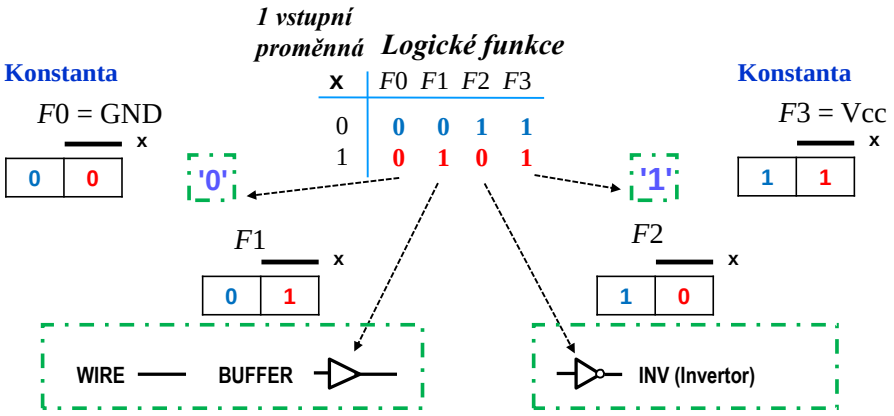
Aplikaci teorému na EQ3 můžeme provést i graficky. Čísla v obrázku dole odpovídají předchozím rovnicím.



Obrázek 19 - Grafická aplikace De Morganova teorému na EQ3

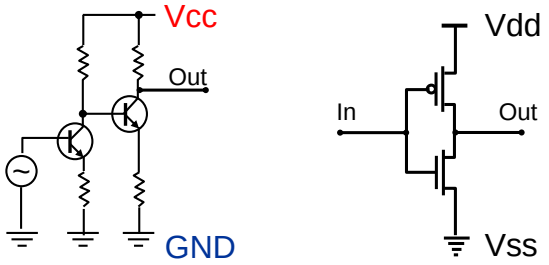
2.3 Logické funkce jedné a dvou vstupních proměnných

Obrázek 20 ukazuje všechny logické funkce **jedné vstupní proměnné**, včetně používaných schematických značek. Dvě již známe, a to Wire/Buffer a NOT/Invertor. Další dvě představují konstanty '0' a '1', a to F0 mající na výstupu vždy logickou '0' a F3 s trvalou logickou '1'.



Obrázek 20 - Logické funkce jedné vstupní proměnné

Konstantní funkce logické '0' se ve schématech nejvíce označuje jako Gnd (*Ground*), zatímco logická '1' se specifikuje Vcc. Jde o tradiční značky pocházející z dob transistorů zapojených se společným kolektorem. Vývojová prostředí si je ponechala.



Obrázek 21 - Označení napětí v obvodech

Používají se následující značky pro napětí (dnes nejčastěji v pozitivní napěťové logice nejběžnější v počítačích, kde logická '1' je vyšším napětím a logická '0' nižším napětím):

GND *ground* - symbol společné nuly, a tou v obvodech bývá napětí 0 V. Vstupy, které mají být trvale na úrovni logické '0', se v schématech zapojují na GND.

V_{CC} (též jako U_{CC} v některé literatuře) pochází z *Common Collector Voltage* a jde o obecně zavedenou zkratku pro použité napájecí napětí. Vstupy trvale na úrovni logické '1', se zapojují na V_{CC}.

Poznámka: Velikost V_{CC} závisí na obvodu. Podle jeho typu může třeba být 24 V (průmyslová logika), 5 V (TTL), 3.3 V (LVTTL) či u CMOS třeba 1.2V (dle jejich typu), ale požívají se i mnohem menší hodnoty, např. 0.6 V na 7 nm technologii.

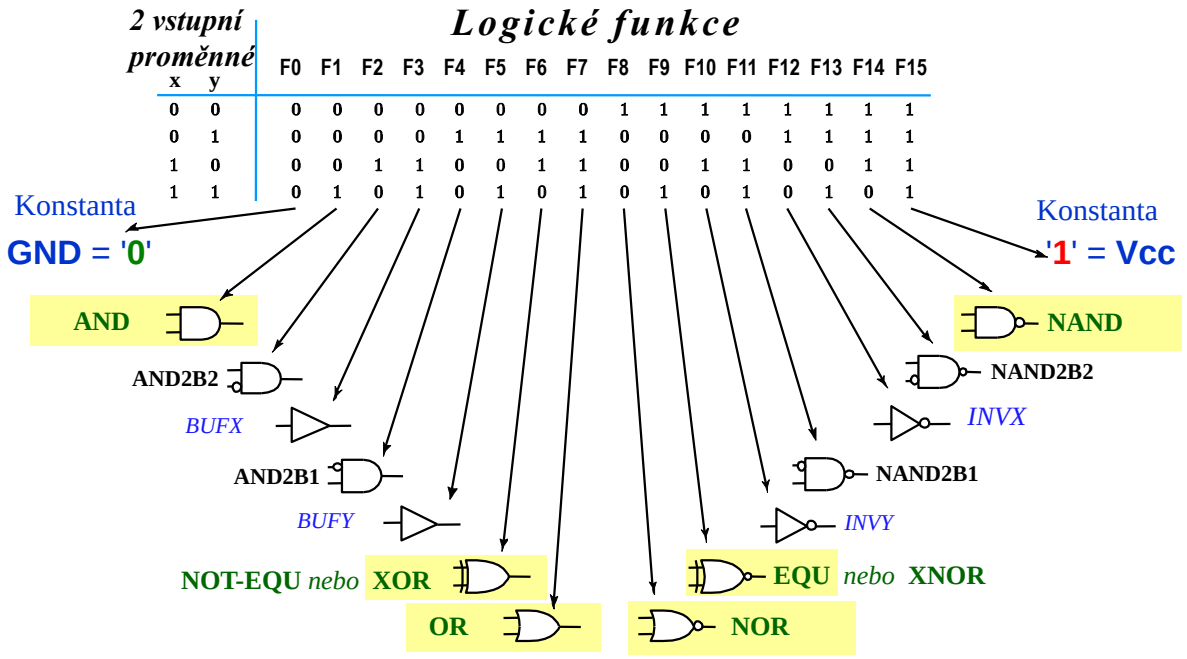
V_{DD} pochází z *Voltage Drain-Drain* napětí CMOS obvodů. V některých publikacích se používá místo V_{CC}, neboť je terminologicky přesnější k dnešní situaci, jelikož většina logických obvodů je na bázi CMOS. Hodně programovacích nástrojů ale používá dál pro napájecí napětí tradiční značku V_{CC}, a tak se ji musíme přidržet i my.

V_{SS} *Voltage for Substrate & Sources* a přestavuje nejnižší napětí CMOS obvodů, které mohlo u některých typů být i záporné.

Obrázek 22 dole ukazuje všechny **logické funkce dvou vstupních proměnných**, opět včetně jejich schematických značek. Když si ho prohlédnete, zjistíte, že je jich sice 16, ale 6 z nich vyznačených modrým písmem, lze nahradit logickými funkcemi jedné proměnné.

První z nich jsou funkce F0 a F15, které nezávisí na vstupech. Jde o konstanty GND a Vcc známé z Obrázek 20, jen ve dvouvstupovém provedení.

Další funkce BUFX, BUFY, INVX a INVY mající výstupní hodnotu závislou jenom na jednom vstupu, a takže je lze nahradit prvky BUFFER nebo invertor (INV) připojený k tomu vstupu, který u příslušné logické funkce ovlivňuje výstup.



Obrázek 22 - Logické funkce dvou vstupních proměnných

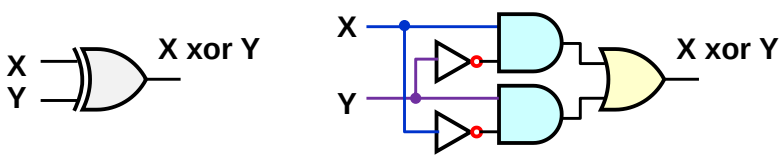
Ze zbývajících 10 logických funkcí se prakticky používá jenom 6, neboť se snadno pamatují – ty jsou v obrázku vyznačené žlutým zvýrazněním, a to AND, XOR, OR, NOR, XNOR a NAND. Ostatní sice existují, v čemž jejich přínos zpravidla končí. Zapisují složenými výrazy.

2.3.1 Funkce XOR

Logická funkce XOR se používá tak často, že jí věnujeme samostatnou podkapitolu.

- **Logická funkce XOR**, *eXclusive OR* (vylučující nebo), dává na svém výstupu '1', pokud má lichý počet svých vstupů v logické '1'.
- Dvouvstupový operátor **xor** existuje snad ve všech návrhových prostředích. V programovacích jazycích C, C# a Java se zapisuje jako binární bitová operace ^ a v matematických rovnicích často ⊕ či přímo slovem "xor".
- Logický výraz xor lze napsat pomocí operací NOT, AND a OR. Lichý počet vstupů v '1' nastane jen při kombinaci X='1' a Y='0', a X='0' a Y='1'. Obě popíšeme mintermy.

$$X \text{ xor } Y = (X \text{ and not } Y) \text{ or } (Y \text{ and not } X) \quad (6)$$



Zajímavou vlastnost xor dostaneme, přivedeme-li jeden jeho vstup '0', alternativně pak logickou '1'. Nezáleží na tom, který vstup využijeme, operace xor je komutativní.

X xor '0' = (X and not '0') or ('0' and not X)
= (X) or ('0') = X (7)

X xor '1' = (X and not '1') or ('1' and not X)
= ('0') or (not X) = not X (8)

Vidíme, že operace XOR může fungovat jako řízený člen, který je buď typu *buffer*, je-li druhý její vstup v '0', nebo se jeho uvedením do '1' změní na invertor.

Obrázek 23 - XOR jako řízený invertor

Použití dvouvstupového xor v obvodech

- Nejčastějším obvodovým využitím xor bývá přepínání jedním jeho vstupem, jímž se volí, zda druhý se chová jako *buffer* či *invertor*, což se hodí v mnoha obvodech, třeba i v aritmetice k přepnutí mezi sčítání a odčítáním, viz Obrázek 119 na str. 103.
- Dále je xor hlavním členem binárních sčítaček. Neuvažujeme-li přenosy do vyšších řádů, pak platí (v tomto odstavci bude + binárním sčítáním) '0'+'0'='0' a '1'+'1'='0', zatímco '1'+'0'='1', '0'+'1'='1'. Jinými slovy, binární součet je logická '1' jen při lichém počtu vstupů v '1', což je právě vlastnost xor.
- Pomocí xor lze zjistit i nerovnost dvou bitů. Dává na výstupu '1', pokud má lichý počet vstupů v '1', tedy za X='1' a Y='0', nebo X='0' a Y='1', kdy jsou X a Y různé.

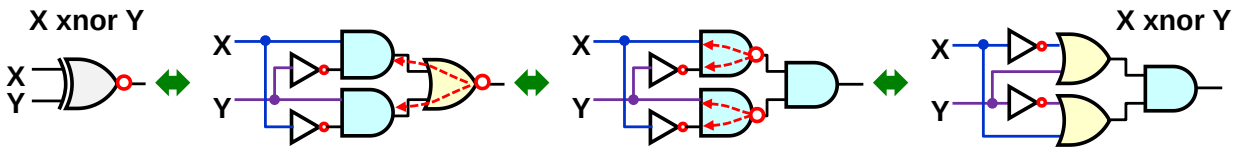
Negovaná XOR, tedy XNOR, *eXclusive NOT OR*, dává na výstupu logickou '1', při sudém počtu vstupů v '0'. U jeho dvouvstupové verze jde tedy o případ, kdy mají oba její vstupy stejnou hodnotu. Zde se někdy používá EQU, *EQUivalency*.

Obrázek 24 - Funkce xor a xnor

Opakované užití equ, resp. xnor, **nedává ale shodu vstupů!** Výraz X equ Y equ Z se bude rovnat '1', bude-li sudý počet vstupů v '**0**', tedy dva či žádný. Kvůli tomu je přesnější název xnor.

Pozor, De Morganův teorém platí pouze u výrazy obsahující not, and a or. Nedá se aplikovat na funkce xor jako celek, která je složenou operací popsanou logickou rovnicí. Musíme xor rozepsat a využít teorém na členy. Vyjdeme z výrazu (6)

X xnor Y = not (X xor Y) = not ((X and not Y) or (Y and not X)) (9)
= not (X and not Y) and not (Y and not X) (10)
= (not X or Y) and (not Y or X) (11)

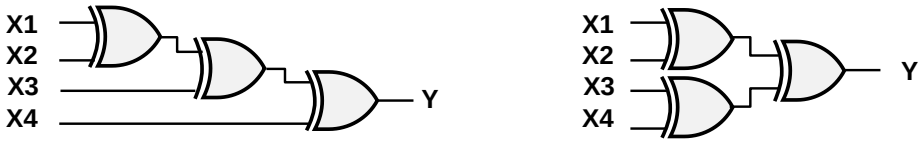


Logickou funkci XOR lze též aplikovat na více vstupů, neboť jsme ji specifikovali jako vracející příznak '1' při lichém počtu vstupních bitů.

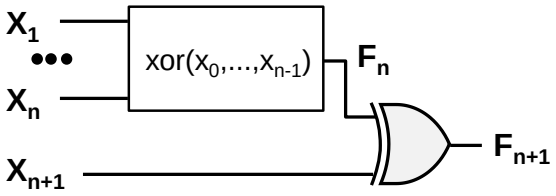
Vícevstupové XOR se už častěji nazývá **paritou**, protože má praktický význam asi jen k výpočtu bitové parity v komunikačních aplikacích. U té se přidává další bit k datovému slovu tak, aby celkový počet bitů v '1' včetně paritního bitu byl sudý či lichý. Z tohoto pohledu počítá XOR sudou paritu, doplní '1' při lichém počtu datových bitů '1'.

Dvouvstupová xor můžeme vrstvit vcelku libovolně, výsledek bude vždy stejný.

Y= ((X1 xor X2) xor X3) xor X4 = (X1 xor X2) xor (X3 xor X4)



Tvrzení si dokážeme matematickou indukcí, třeba pro uspořádání vlevo na obrázku nahoře.



- Předpokládejme, že máme logickou funkci Fn pro n-vstupů, n>=2, která vrací '1' za předpokladu lichého počtu svých vstupů v '1'. Takovou známe jako dvouvstupové XOR.
- Dává-li F_N na výstupu '1', pak bude lichý počet X_1 až X_N vstupů v '1'. F_{n+1} dá na výstupu '1' jedině tehdy, kdy X_{n+1} je v '0'. Zůstal tedy zachovaný počet lichých vstupů v '1' po rozšíření o X_{N+1}.
- Má-li F_N na výstupu '0', pak bude sudý počet X_1 až X_N vstupů v '1'. F_{N+1} dá na výstupu '1' jedině tehdy, kdy X_{N+1} je v '1', tedy při lichém počtu vstupů v bitech X_1 až X_{N+1}, jinak 0.
- Předchozí úvahy vedou k závěru, že i F_{N+1} bude v '1' jedině tehdy, pokud bude lichý počet vstupů v '1', což jsme chtěli dokázat.

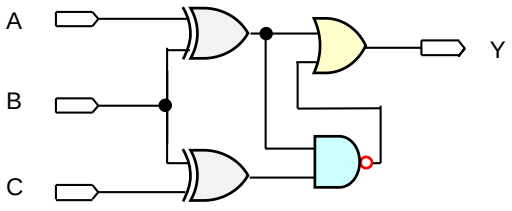
Důkaz lze analogicky provést i u stromečkové struktury. U ní dojdeme ke stejnému výsledku.

Podobně lze též ukázat, že použijeme-li xnor ve vrstvení na obrázcích nahoře místo xor, pak výstup bude v '1' při sudém počtu vstupů v logické **'0'**.

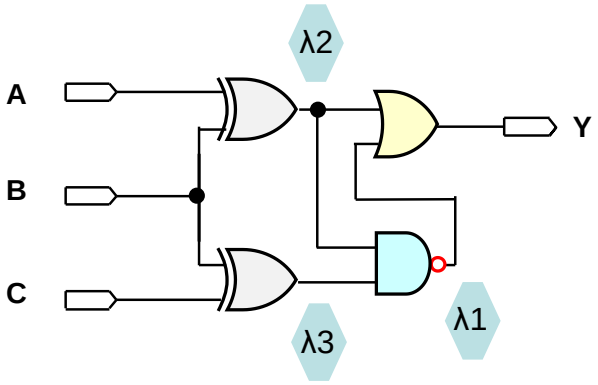
2.4 Převod logického schéma na výraz

Na závěr této části si ještě ukážeme převod logického schéma na logický výraz. Jelikož schéma představuje postup vyhodnocení logické funkce, stačí ho tedy jen procházet a operace zapisovat jako logický výraz, co demonstrujeme na příkladu.

Příklad: Napište logický výraz odpovídající logické funkci na obrázku:



Řešení: Logický výraz můžeme sestavit buď odleva, tedy ve směru výpočtu, nebo naopak od konce. Ukážeme si druhý způsob, který bývá univerzálnější, protože jím konvertujeme i složité obvody s vnitřními smyčkami. Označme si napřed výstupy jednotlivých bloků.



Obvod obsahuje XOR hradla, a tak zvolíme zápis operátorů slovy. Výstup Y je OR funkcí:

$$Y = \lambda 2 \text{ or } \lambda 1 \qquad \text{(eq1)}$$

λ1 je dáno funkcí AND s bublinkou negace, tedy λ1 = not (λ2 and λ3). Substituujeme vztah pro λ1 do (eq1), čím dostaneme:

$$Y = \lambda 2 \text{ or not } (\lambda 2 \text{ and } \lambda 3) \qquad \text{(eq2)}$$

Dále vypočteme λ2 = A xor B a dosadíme za dva jeho výskyty v (eq2)

$$Y=(A \text{ xor } B) \text{ or not } ((A \text{ xor } B) \text{ and } \lambda 3) \qquad \text{(eq3)}$$

Zůstane nám jen stanovit λ3 = B xor C a to dosadit do rovnice (eq3), čím obdržíme výsledek:

$$Y=(A \text{ xor } B) \text{ or not } ((A \text{ xor } B) \text{ and } (B \text{ xor } C)) \qquad \text{(eq4)}$$

Rovnici (eq4) lze zapsat i pomocí symbolů pro operátory, xor necháme názvem

$$Y=(A \text{ xor } B)+((A \text{ xor } B).(B \text{ xor } C))´$$

~o~

Umíme již vytvořit grafické schéma z logické funkce, a naopak, ale pořád ještě sestavujeme výrazy spíš intuicí než metodou.

I když budeme používat návrhová prostředí, pořád jim potřebujeme specifikovat chování logické funkce. Tu lze sice popsat i seznamem '0' a '1', ale zápis výrazem je v mnoha případech úspornější způsobem. Musíme se tedy podívat na metodiku, jak ho můžeme vytvořit, než začneme s výkladu základních obvodů.

V následující části se tedy podíváme na možné specifikace hodnot logických funkcí, čehož využijeme k optimalizaci jejich výrazů pomocí Karnaughových map. S nimi již získáme vše potřebné k pozdějšímu výkladu základních obvodů.

3 Popis logické funkce

Mějme logické proměnné, které nabývají jen hodnot z nějaké konečné množiny B.

Úplně definovanou logickou funkcí n vstupních proměnných (*Completely Specified Logic Function*) $y = f(x_1,x_2,x_3,..x_n)$ nazveme zobrazení:

$$B^n \rightarrow B, \text{ kde } (x_1,x_2,x_3,..x_n) \in B^n , x_i \in B, y \in B.$$

Obsahuje-li B pouze logickou nula a jedničku, B={'0','1'}, pak má i mohutnost |B|=2 a určuje **dvouhodnotovou logiku**[3] (*two-valued logic*).

Kartézským součinem B^n vytvoříme všechny možné n-tice prvků z B, pro |B|=2 je $|B^n|=2^n$ a zobrazením (*mapping*) $B^n \rightarrow B$ jim přiřadíme výstupy. Pro n logických proměnných existuje 2^{2^n} různých přiřazeních, tedy odlišných logických funkcí n proměnných.

Pro n=0 jsou jen 2, a to konstanty GND='0' a Vcc='1'. Při n=1 jich máme $2^{2^1} = 2^2 = 4$, pro n=2 již $2^{2^2} = 2^4 = 16$, pro n=3 pak dostaneme $2^{2^3} = 2^8 = 256$ logických funkcí.

Příklad: Mějme B={'0','1'}. Logickou funkci dvou vstupů zapíšeme jako y=f(x1, x2). Kartézský součin B^2 vytvoří čtyři dvojice, tj. B^2 = { ('0', '0'), ('0', '1'), ('1', '0'), ('1', '1') }. Můžeme jim přiřadit jednu ze 16 různých kombinací výstupních hodnot. Vybereme si jedno z nich. Výstupu přiřadíme logickou '1' jenom při lichém počtu vstupů v '1', což známe jako funkci xor. Definujeme y=xor(x1, x2) zobrazením:

xor: B2→B =	('0', '0') → '0'	zjednodušený zápis	0 0 → 0
	('0', '1') → '1'		0 1 → 1
	('1', '0') → '1'		1 0 → 1
	('1', '1') → '0'		1 1 → 0

Pravdivostní tabulka představuje leda jiný jeho zápis: Zobrazení v ní zapíšeme třeba jako

x1	x2	xor
0	0	0
0	1	1
1	0	1
1	1	0

nebo i takto:

x1	x2	xor
1	1	0
1	0	1
0	1	1
0	0	0

či ještě jinak:

x1	x2	xor
0	0	0
1	1	0
1	0	1
0	1	1

Tabulky popisují totožnou logickou funkci. Nezáleží na pořadí jejich řádků, ty lze je uvést v libovolném sledu, jen musíme výstup přidělit všem. I fyzická realizace logické funkce vyžaduje znát výstupní hodnotu pro každou možnou kombinaci hodnot vstupů. V HDL jazycích lze však zadat příkaz, aby se všem dosud neurčeným přiřadila námi zvolená hodnota.

Kombinační logický obvod definujeme seznamem m logických funkcí tvaru:
$y_k = f(x_1, x_2, x_3,\ldots,x_n)$; kde k=1 až m

[3] Při návrhu logických obvodů nevystačíme jen s logickou ´0´ a logickou ´1´. I v tomto textu si brzy zavedeme 3hodnotovou logiku přidáním hodnoty X (*don´t care*), protože se bez ní neobejdeme. V profesionální práci se pak hodně používá 9hodnotová logika MVL-9, o níž bude více v učebnici o *concurrent* VHDL.

Po změně vstupů x_j sice proběhnou v kombinačním obvodu dočasné přechodové děje, ale po jejich ustálení se objeví výstupy y_k , které závisí jen na současných vstupech, jinými slovy, tytéž hodnoty vstupů x_1 až x_n vedou na pořád stejné hodnoty výstupů y_1 až y_m.

Poznámka: Odlišným případem budou sekvenční logické obvody, téma závěrečné kapitoly 7, které obsahují paměťové členy, a tak jejich výstupy závisí na sekvenci předchozích hodnot vstupů a dat v pamětích. Stejné okamžité vstupy mohou pokaždé dávat jiné výstupy.

Vypisování všech kombinací je zdlouhavé, a tak se často spojuje několik funkcí do jedné tabulky. Například můžeme spolu s *xor* napsat i další běžné logické funkce:

x1	x2	xor	xnor	and	nand	or	nor
0	0	0	1	0	1	0	1
0	1	1	0	0	1	1	0
1	0	1	0	0	1	1	0
1	1	0	1	1	0	1	0

Někdy se hodí i snížení počtu řádků. Například pro přidělení požadavku na přerušení potřebujeme znát index nejvyššího vstupu xi v logické '1', aby se obsloužilo dle priority požadavku.

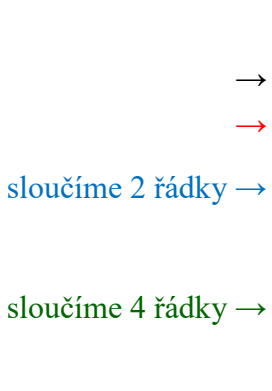
Pro 3 vstupy může funkce mít například tabulku vpravo.

- Výstup p3 je 00, pokud žádný vstup není v '1'.
- Výstup p3 přejde do 01, pokud jen vstup x1 je '1'.
- Pokud bude x3 v '0' a x2 v '1', pak výstup p3 má hodnotu 10, bez ohledu na vstup x1, protože chceme, aby x2 mělo vyšší prioritu než x1.
- Výstup p3 bude 11 při nejvíce prioritním vstupu x3 v '1' bez ohledu na stav ostatních vstupů.

x3	x2	x1	p3		index
0	0	0	**0**	**0**	**0**
0	0	1	**0**	**1**	**1**
0	1	0	**1**	**0**	**2**
0	1	1	**1**	**0**	**2**
1	0	0	**1**	**1**	**3**
1	0	1	**1**	**1**	**3**
1	1	0	**1**	**1**	**3**
1	1	1	**1**	**1**	**3**

Předchozí tabulku lze zkrátit použitím příznaku, že stejná hodnota výstupu se opakuje pro některý vstupní bit jak v '1', tak v '0'. Ten nahradíme třeba znakem - (pomlčka) pro reprezentování jakési "*wildcard*" (divoké karty, zástupného znaku).

x3	x2	x1	p3	
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1



x3	x2	x1	p3	
0	0	0	0	0
0	0	1	0	1
0	1	-	1	0
1	-	-	1	1

Tabulka 1 - Slučování vstupů zástupnými wildcards

Nová tabulka (vpravo nahoře) má už jen 4 řádky. Aplikací *wildcards* se zkracuje zápis slučováním vstupů (*merged inputs*), jde tedy o jakýsi předpis pro generování řádků tabulek.

Snadno teď napíšeme i větší funkci pro 10 vstupů přerušení vracející číslo nejvyššího požadavku. Místo 2^{10} =1024 řádků, které bychom museli uvést při vypisování celé tabulky, nám jich stačilo jen 11:

x10	x9	x8	x7	x6	x5	x4	x3	x2	x1	p10				index
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	1	1
0	0	0	0	0	0	0	0	1	-	0	0	1	0	2
0	0	0	0	0	0	0	1	-	-	0	0	1	1	3
0	0	0	0	0	0	1	-	-	-	0	1	0	0	4
0	0	0	0	0	1	-	-	-	-	0	1	0	1	5
0	0	0	0	1	-	-	-	-	-	0	1	1	0	6
0	0	0	1	-	-	-	-	-	-	0	1	1	1	7
0	0	1	-	-	-	-	-	-	-	1	0	0	0	8
0	1	-	-	-	-	-	-	-	-	1	0	0	1	9
1	-	-	-	-	-	-	-	-	-	1	0	1	0	10

Poslední řádek tabulky s 9 *wildcards*, `1---------`, ve skutečnosti reprezentuje předpis, který vygeneruje 2^9 =512 řádků, neboť každý použitý zástupný *wildcard* nabývá 2 hodnot, jak '0', tak '1'. Všechny vytvořené řádky mají stejný výstup p10=1010 (=index 10).

Jiný příklad: Tabulka vlevo je ve skutečnosti zkráceným zápisem tabulky vpravo:

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

→

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

U určitých funkcí se neobejdeme bez zástupných *wildcards*, jako například u předchozí prioritní funkce `p10` pro deset vstupů. Při ručním zápisu se však jejich nadměrným používáním snižuje názornost, jak je patrné i z levé tabulky nahoře, z níž na první pohled nepoznáme, zda jsme skutečně uvedli všechny možné kombinace vstupů. Používání *wildcards* není nutnost, leda pomůckou, kterou si sami usnadňujeme zápis. Hojně se uplatňují především k zadání pravdivostních tabulek během počítačové minimalizace logických funkcí.

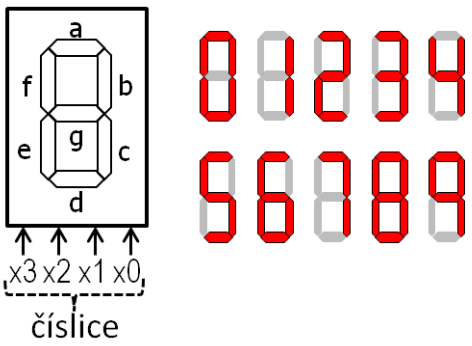
3.1 Hodnota X - don´t care

Pokud bychom třeba měli zapsat pravdivostní tabulku pro dekodér převádějící dekadické číslice na 7segmentový displej, vytvoříme ji pro vstupní hodnoty 0 až 9 (binárně kódované jako *unsigned integer*, tedy "0000" až "1001").

Jaké výstupní hodnoty máme ale přiřadit vstupům 10 až 15 (*unsigned* 1010 až 1111), které zadání nespecifikuje? Můžeme si pro ně něco vymyslet, ale v době návrhu ještě nevíme, zda námi náhodně vybrané hodnoty neztíží pozdější operace, jako třeba minimalizaci logických funkcí. Moudřejší bude zatím odložit rozhodnutí o jejich hodnotách.

Jako znamení odloženého rozhodnutí použijeme příznak zvaný "***don´t care***", který specifikuje, že nám na výstupní hodnotě nezáleží. Ten se často zapisuje jako X.

Pomocí X a zástupného znaku '-' už snadno zapíšeme tabulku dekodéru převádějící dekadickou číslici na její 7segmentový obraz. Nechť jednotlivé LED svítí při logické '1'.



Obrázek 25 - 7segmentový displej

Číslice	bity čísla				LED						
	x3	x2	x1	x0	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	0	0	1	1
10-11	1	0	1	-	X	X	X	X	X	X	X
12-15	1	1	-	-	X	X	X	X	X	X	X

Tabulka 7segmentového displeje vpravo na Obrázek 25 je téměř profesionální, až na dělení vstupů a výstupů do jednotlivých sloupců. Při stručnějším zápisu se logické hodnoty často spojují do sekvencí, respektive vektorů, což výrazně zmenší tabulku.

Například místo:

x3	x2	x1	x0
0	0	0	0

zapíšeme jen 0000 a do záhlaví tabulky uvedeme pořadí logických proměnných v sekvenci. Tabulku, kterou uvádí Obrázek 25 nahoře, lze zkrátit na stručnější zápis vpravo:

Číslice	bity čísla				LED						
	x3	x2	x1	x0	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	0	0	1	1
10-11	1	0	1	-	X	X	X	X	X	X	X
12-15	1	1	-	-	X	X	X	X	X	X	X

Číslice	Binárně x:3210	LED abcdefg
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	1011111
7	0111	1110000
8	1000	1111111
9	1001	1110011
10-11	101-	XXXXXXX
12-15	11--	XXXXXXX

Sekvence logických '0' a '1' mají i praktický význam ke zkrácení zápisů logické funkce v profesionálních vývojových nástrojích. Logické hodnoty se v nich často zpracovávají ve formě vektorů ke zkrácení kódu. Naproti tomu se v nich téměř nepoužívá zdlouhavé definování logické funkce pomocí vyplňování tabulek dělených na jednotlivé sloupce.

Více o "don't-care"

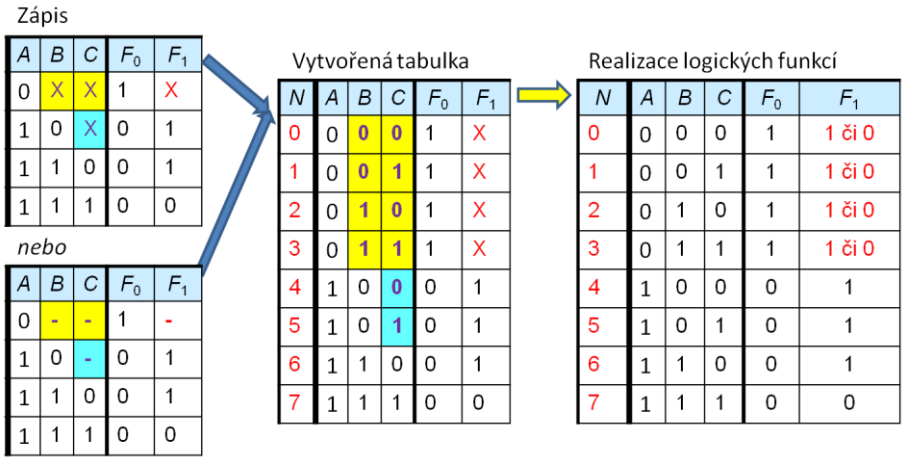
- "*don't-care*" označuje pouze návrhářovu poznámku, že se o hodnotě výstupu rozhodne během další kroků návrhu, a to podle toho, co se ukáže výhodnějším. Jedná se tedy o znamení odloženého rozhodnutí (tj. něco jako značka *to-do*).

- "*don't care*" nemá význam neznámého výstupu, ačkoliv se tak někdy nesprávně překládá. V češtině by jeho významu odpovídaly přesněji pojmy "zatím nezadaný", nebo "dosud nespecifikovaný", či "ještě neurčený".
- "*don't-care*" nelze fyzicky realizovat v obvodech, a tak se nakonec všechny symboly "*don't-care*" nahrazují nějakými konkrétními realizovatelnými logickými hodnotami, tedy např. logickou '0' či logickou '1'.[4]
- "*don't-care*" se nepoužívá ve spojení se vstupy. Fyzická realizace logických funkcí žádá vždy jejich znalost. U vstupů se píší zástupné *wildcards*, viz str. 28, které mají význam sloučení definic. Hodnotu "*don't-care*" si později zvolíme, u *wildcards* je již daná.

V publikacích se nezavedla jednotná syntaxe zástupných *wildcards* pro sloučené vstupy a "*don't care*" výstupy označují stejnými symboly, zpravidla znaky X, které jsou mnohem výraznější než pouhá pomlčka. Bez ohledu na použité symboly se však snadno zorientujeme podle jejich umístění v pravdivostní tabulce.

Význam závisí na tom, zda se symbol nachází v části vstupů nebo výstupů:

- Vstup logické funkce: Je-li například napsaný kód "0 - -" nebo "0 X X" (dle autorem použité notace), pak se vygenerují 4 řádky vstupů 000, 001, 010 a 011 s naprosto stejnými výstupními hodnotami, protože znak, ať už 'X' či '-', má zde postavení zástupného *wildcard*, tedy předpisu pro generování hodnot.
- Výstup logické funkce: Například kód "1X" či "1-" bude u výstupu znamenat odložené rozhodnutí o jeho hodnotě. Tady má naopak symbol vždy význam "*don´t care*". U výstupu totiž nemůžeme použít žádné generování zástupnými *wildcard* znaky ─ každý výstup musí mít v konečné tabulce použité pro realizaci logické funkce vždy jen jednu fixní hodnotu, a lze pouze dočasně odložit rozhodnutí o tom, jaká nakonec bude.



3.2 Zápis pravdivostní tabulky pomocí výčtu hodnot

Tabulka 2 popisuje 4 logické funkce, jejichž výstupy F0 až F3 nabývají hodnoty '1' jen pro jednu logickou kombinaci vstupů, hlásí tak její přítomnost. Společně tvoří ***one-hot*** **dekodér**,

[4] Ve snaze o maximální přesnost se vyhýbáme tvrzení, že se X (don´t care) se vždy a všude musí dodefinovat buď na logickou ´0´ nebo na ´1´. Většinou se tak stane, ale existují i jiné možnosti jako již zmíněný stav 'Z' vysoké impedance, více na str. 49, který se potřebuje na obousměrných paralelních počítačových sběrnicích. Dále může výstup být realizovaný i otevřeným kolektorem, třeba na I2C sériové sběrnici nasazované v některé audiotechnice, což již patří do oblasti jiných odborných předmětů.

do češtiny překládaným jako 1 z N, v našem případě 1 ze 4. Jde o velmi důležitý logický konstrukční prvek, který tvoří základ mnoha dalších funkcí. Bude ještě v kapitole 5.1 na str. 79.

N	B	A	F0	F1	F2	F3
0	0	0	1	0	0	0
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	1	1	0	0	0	1

Tabulka 2- Dekodér "One-hot" - 1 z 4

Elegantněji specifikujeme jeho funkce množinou vstupních hodnot, v nichž nabývají logické '1', což nazveme ***on-set***. Kombinace vstupních bitů zakódujeme jako binární číslo *unsigned*. Tabulka 2 se pak redukuje na jeden řádek, na seznam *onset*-ů.

$$F0^{on} = \{ 0 \}, \quad F1^{on} = \{ 1 \}, F2^{on} = \{ 2 \}, F3^{on} = \{ 3 \},$$

V ostatních stavech výstupy F0 až F3 nabývají '0'. Psaní indexů není příliš pohodlné. Pojem minterm jako AND člen se zavedl na Obrázek 5 na str. 13. Víme o něm, že na výstupu má '1' pouze pro jedinou vstupní kombinaci. Můžeme tak sepsat seznam, které mintermy se použijí. Značíme ho malým m a v závorkách uvedeme binární hodnoty vstupů jako *unsigned* číslo.

$$F0 = m(0), \quad F1 = m(1), \quad F2 = m(2), \quad F3 = m(3)$$

Tabulka 3 na další stránce popisuje jiný analogický dekodér, který se nazývá ***one-cold***, protože výstupy F0 až F4 budou právě pro jednu vstupní kombinaci v '0'. Česká terminologie žel nerozlišuje mezi dekodéry *one-hot* a *one cold* a oba překládá jako 1 z N, zde tedy 1 z 4

N	B	A	F0	F1	F2	F3
0	0	0	**0**	1	1	1
1	0	1	1	**0**	1	1
2	1	0	1	1	**0**	1
3	1	1	1	1	1	**0**

Tabulka 3- Dekodér "One-cold" - 1 z 4

Zde popis pomocí *on-setů* není výhodný, lepší je popis pomocí *off-setů*[5], které znamenají množinu vstupů, pro které je výstup v '0'. Funkce dekodéru *one-cold* zapíšeme opět snadno:

$$F0^{off} = \{ 0 \}, \quad F1^{off} = \{ 1 \}, F2^{off} = \{ 2 \}, F3^{off} = \{ 3 \}$$

Opět zde platí, že neuvedené hodnoty jsou v logické '1'. Zde se zase používá zápisově snazší notace pomocí maxtermů, značených jako velké M, tedy členů OR, o nichž víme, že nabývají na výstupu '0' pouze pro jedinou kombinaci hodnot svých vstupů.

$$F0 = M(0), \quad F1 = M(1), \quad F2 = M(2), \quad F3 = M(3)$$

Popisy lze použít i u logických funkcí s *don´t care* stavy, přidáme jen *don´t care* set.

---

[5] Název *off-set* je poměrně zavádějící, protože se tak v technice a matematice obvykle označuje odchylka nebo posun, ale v literatuře o logických obvodech se opravdu používá. Česká terminologie pro *on-set* a *off-set* je v logických obvodech tak různorodá, že ji ani neuvádíme. Matematické označení pro zápisy *on-set, off-set* a *don´t care set* se také liší podle autora. Zde uvedené popisy Fon , Foff a Fdc nejsou ustálené.

Naproti tomu malé m (od minterm) a offset jako velké M (od Maxterm), a dc (don´t care) se běžně používají. Vždyť taky jde taky o mnohem kratší notace☺

N	C	B	A	X	Y
0	0	0	0	0	0
1	0	0	1	0	0
2	0	1	0	X	0
3	0	1	1	X	0
4	1	0	0	1	0
5	1	0	1	1	X
6	1	1	0	1	1
7	1	1	1	1	1

Logické funkce X(C,B,A) a Y(C,B,A) definované tabulkou nahoře, můžeme zapsat takto:

$$X: X^{off} = \{ 0,1 \}, X^{dc} = \{ 2, 3 \} ; Y: Y^{on} = \{ 6,7 \}, \; Y^{dc} = \{ 5 \} ,$$

respektive X: M(0,1) dc(2,3); Y: m(6,7), dc(5)

Pro každou funkci jsme zvolili metodu, která nám dala nejméně práce. Výstup X jsme popsali pomocí *off-setu* a *don´t care* setu, protože výstupních '0' bylo méně než '1', zatímco výstup Y jsme vytvořili pomocí *on-setu* a *don´t care* setu.

Příklad: Napíšeme si základní funkce logiky pomocí mintermů a maxtermů:

Váha vstupu	+2	+1						
Unsigned index	x	y	xor	xnor	and	nand	or	nor
0	0	0	0	1	0	1	0	1
1	0	1	1	0	0	1	1	0
2	1	0	1	0	0	1	1	0
3	1	1	0	1	1	0	1	0

XOR: m(1,2)
XNOR: m(0,3) ale též XOR: M(0,3), XNOR:M(1,2)
AND: m(3)
OR:M(0)
NAND: M(3)
NOR:m(0)

3.3 Karnaughovy mapy

V technické praxi se logické funkce s menším počtem vstupů specifikují Karnaughovou mapou, která je rychlejším a přehlednějším zápisem. Odvozíme si ji z pravdivostní tabulky logické funkce Y=f(D,C,B,A) se 4 vstupy D,C,B a A, kde D má nejvyšší váhu. Její výstup Y nabývá 16 hodnot, které označíme jen logickými konstantami y00 až y15, jejichž indexy naznačí řazení výstupů. V realitě budou samozřejmě mít nějaké hodnoty logické '0', '1' či X (*don't care*).

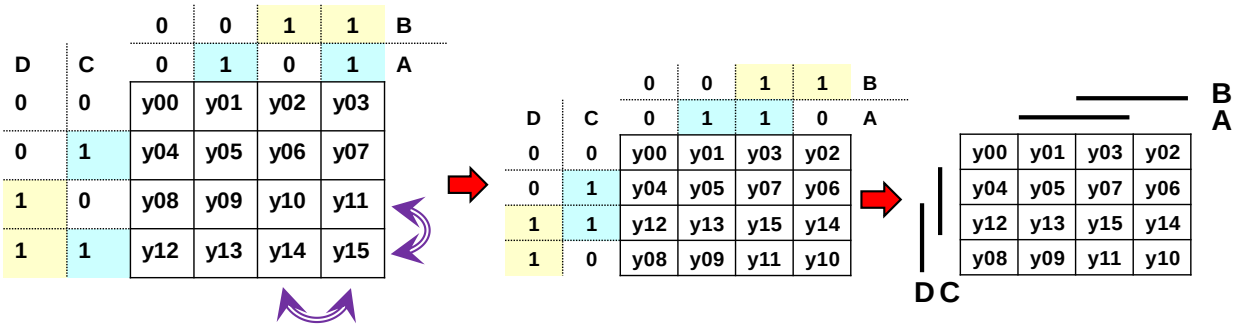
D	C	B	A	Y
0	0	0	0	y00
0	0	0	1	y01
0	0	1	0	y02
0	0	1	1	y03
0	1	0	0	y04
0	1	0	1	y05
0	1	1	0	y06
0	1	1	1	y07
1	0	0	0	y08
1	0	0	1	y09
1	0	1	0	y10
1	0	1	1	y11
1	1	0	0	y12
1	1	0	1	y13
1	1	1	0	y14
1	1	1	1	y15

		0	0	1	1	B
D	C	0	1	0	1	A
0	0	y00	y01	y02	y03	
0	1	y04	y05	y06	y07	
1	0	y08	y09	y10	y11	
1	1	y12	y13	y14	y15	

Obrázek 26 - Pravdivostní tabulka nakreslená v maticovém tvaru

Pravdivostní tabulka vlevo má 16 řádků a čtveřice po sobě jdoucích řádků mají totožné vstupy D a C. S výhodou využijeme zkrácený zápis v maticovém tvaru 4x4, viz vpravo, kde pro každý výstup se hodnoty jeho vstupů určí podle polohy ve sloupci a řádce.

Tabulku vpravo na Obrázek 26 upravíme. Prohodíme její dva poslední sloupce a dvě poslední řádky, čímž dostaneme prostřední tabulku na Obrázek 27 — ta je již Karnaughovou mapou logické funkce. Logické '1' vstupů v ní leží vedle sebe, a tak místo vypisování 0 a 1 se často kreslí jen čára symbolizující, kde má vstup hodnotu '1', viz tabulka vpravo.



Obrázek 27 - Geneze Karnaughovy mapy 4x4

Nejdůležitější vlastností Karnaughovy mapy, a zároveň nutnou podmínkou k tomu, aby se vůbec jednalo o Karnaughovu mapu, je skutečnost, že při jakémkoliv pohybu v ní o jedno políčko svisle nebo vodorovně se **změní pouze jediná vstupní proměnná**.

Například výstup y00 má vstupy DCBA=0000 a výstup y04 o řádek níže 0100. Při přechodu od y00 k y04 se tedy změnil jen vstup C z '0' na '1'. Platí to i přes konce mapy, třeba při posunu z prvního řádku na čtvrtý ve stejném sloupci.

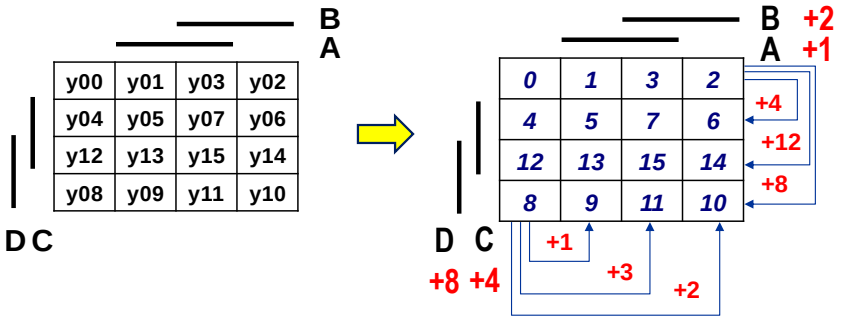
Vezmeme-li si na ukázku třeba poslední sloupec. Výstup y02 má vstupy DCBA=0010 a výstup y10 zase DCBA = 1010. Změnilo se jen D z '0' na '1'. I výstup y14 na konci třetího řádku má hodnoty vstupů DCBA=1110 a výstup y12 na začátku stejného řádku dává hodnotu DCBA = 1100. Změnilo se jen B z '1' na '0'.

Řazení výstupních hodnot logické funkce tak, aby se **měnila pouze jediná její vstupní proměnná** při vodorovném pohybu v řádce či svislém ve sloupci, se nazývá **Grayův kód**. Využívá se nejen v minimalizaci logických funkcí, ale také ve snímačích pozice a přenosech informace, například ke korekci chyb v digitální televizi.

Indexy i výstupů yi v Karnaughově mapě nejdou za sebou. Porovnáme-li jejich čísla v jednom řádku, pak vůči prvkům v prvním sloupci je index ve druhém sloupci na stejném řádku vždy větší o +1 , ve třetím sloupci o +3 a ve čtvrtém sloupci o +2.

Vlastnost vyplývá ze vstupních proměnných. Každý bit má váhu danou mocninou řadou 2^n. Uspořádáme-li vstupy od nejvýznamnějšího D vpravo, tedy DCBA, pak vstup A má váhu $1=2^0$, vstup B má váhu $2=2^1$, vstup C zas váhu $4=2^2$ a vstup D váhu $8=2^3$. Součet vah proměnných (řádek+sloupec) určí hodnotu indexu v příslušném poli Karnaughovy mapy.

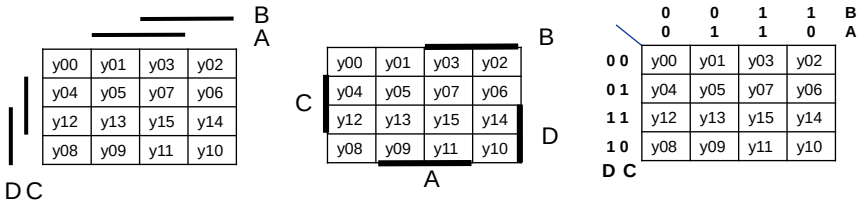
Druhý sloupec má indexy o +1 vyšší než první sloupec, protože se v něm uplatní proměnná A s vahou +1. Třetí sloupec bude mít navíc oproti prvnímu už dvě proměnné A+B, tedy bude mít indexy o +3 vyšší než první sloupec. Analogicky poslední sloupec má navíc oproti prvnímu sloupci jen proměnnou b, a ta má váhu +2.



Obrázek 28 - Závislosti v Karnaughově mapě 4x4

Z uvedené vlastnosti odvodíme i rozdíly mezi indexy ve sloupcích. Druhý řádek má hodnoty indexů vždy o +4 větší než odpovídající prvek stejného sloupce prvního řádku (+C=4). Třetí řádek má indexy větší o +12 (+D+C) oproti prvnímu řádku a čtvrtý řádek o +8 (+D). Stačí nám tak správně přidělit indexy prvnímu řádku a zbylé si už mechanicky odvodíme.

Karnaughova mapa, zkráceně KM, kterou uvádí Obrázek 27, není jedinou možností, jak ji nakreslit nebo jak uspořádat vstupní proměnné. Různí autoři používají mnohdy odlišné styly, dle svého zvyku. Některé možnosti uvádí Obrázek 29.



Obrázek 29 - Některá možná značení proměnných u Karnaughovy mapy 4x4

Rovněž se mohou i jinak uspořádat proměnné, takže získají odlišné váhy, čímž dostaneme i jiné řazení indexů. Možných Grayových kódů existuje opravdu hodně[6], třeba pro čtyři bity je jich již 5712. V logice i programech se kvůli jednoduchému konverznímu algoritmu nejčastěji používá "*binary-reflected Gray code*", má ho i Obrázek 29. Přidržíme se ho v dalším textu.

[6] Přehled dalších Grayových kódů uvádí Wikipedia: *https://en.wikipedia.org/wiki/Gray_code*

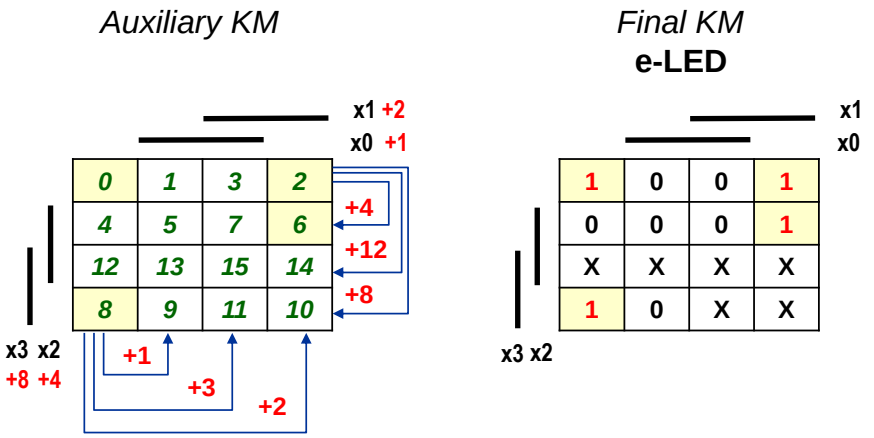
Lze však vytvořit i jiná řazení, která též budou **Grayovými kódy**, pokud splní podmínku, že **vždy se změní hodnota jen jedné vstupní proměnné při jakémkoli posunu vodorovně či svisle o jeden sloupec či řádek, a to včetně přechodů přes konce mapy**.

Příklad: Nakreslete Karnaughovu mapu (KM) pro e-LED 7segmentového displeje.

Řešení: Obrázek 25 na straně 30 popisuje pravdivostní tabulku 7segmentového displeje, z níž vezmeme hodnoty pro e-LED. Zatím nemáme velké zkušenosti s kreslením KM, tak raději postupujeme přes mezikrok, čímž se vyhneme zbytečné chybě☺.

Napřed si nakreslíme pomocnou KM, ve které vyplníme čísla indexů jednotlivých polí dle našeho řazení vstupních proměnných. Podle ní pak zapíšeme hodnoty do finální pravdivostní tabulky e-LED.

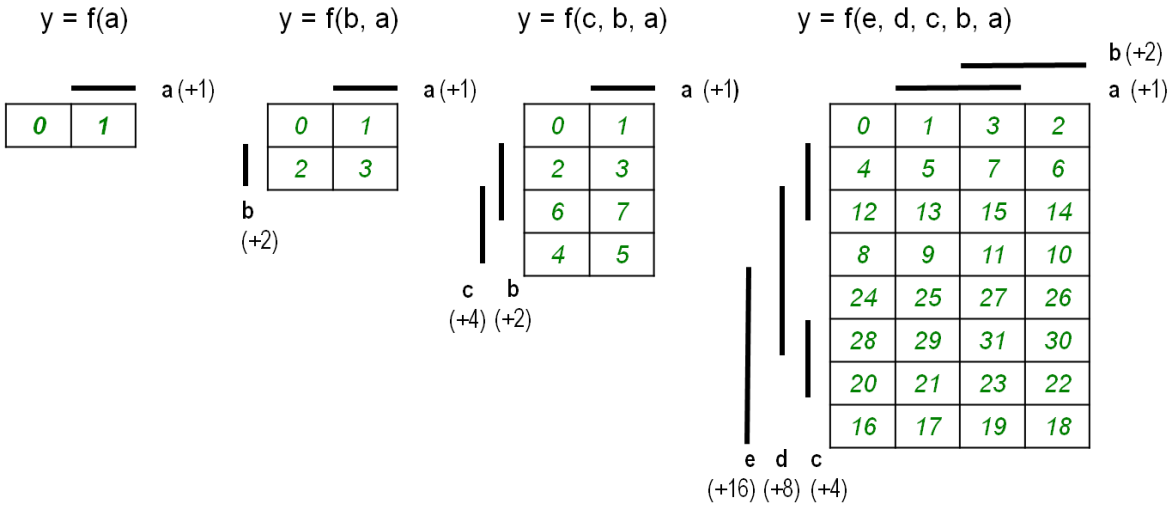
N	x:3210	e
0	0000	1
1	0001	0
2	0010	1
3	0011	0
4	0100	0
5	0101	0
6	0110	1
7	0111	0
8	1000	1
9	1001	0
10-11	101-	X
12-15	11--	X



Obrázek 30 - Karnaughova mapa e-LED 7segmentového displeje

3.3.1 Karnaughovy mapy různých velikostí

Karnaughovy mapy se nehodí ke zpracování v počítači a používají jen k ruční minimalizaci či zápisu logických funkcí s malým počtem vstupů. Složitost KM totiž roste exponenciálně s nárůstem počtu proměnných. Obrázek 31 prezentuje některé varianty (vybrané z mnoha možných), jak sestavit Karnaughovu mapu pro jiné počty vstupů než 4x4, a to včetně výsledného číslování polí. Ve všech je užitý Grayův kód kód typu "*binary-reflected* ".



Obrázek 31 - Karnaughovy mapy pro jiné velikosti než 4x4

Naštěstí lze u logické funkce snížit počet jejích proměnných různými dekompozicemi, brzy si třeba ukážeme Shannonovu expanzi, viz str. 49 a 52. V ručních návrzích tak můžeme vždy vystačit s mapami do velikosti 4x4☺

3.3.2 Princip minimalizace Karnaughových map metodou SoP

Zatím jsme používali pojmy mintermy a maxtermy, které jsme definovali jako členy, které obsahují všechny proměnné z nějaké zadané množiny vstupů a pouze pro jedinou kombinaci vstupních hodnot dávají výstup v '1' (minterm) nebo v '0' (maxterm).

Mějme nějakou logickou funkci f() s N různými vstupními proměnnými. Z těch můžeme vytvářet výrazy s Q odlišnými proměnnými, Q=1 až N, spojenými AND operátory, například výraz x1 and not x2 and x4. Pokud výrazy neobsahují všechny vstupy, jen některé, pak definují více výstupů logické funkce. Zavedeme si obecnější pojem AND implikant. Analogicky definujeme OR implikant pro spojení pouze operátory OR, např. x1 or not x3.

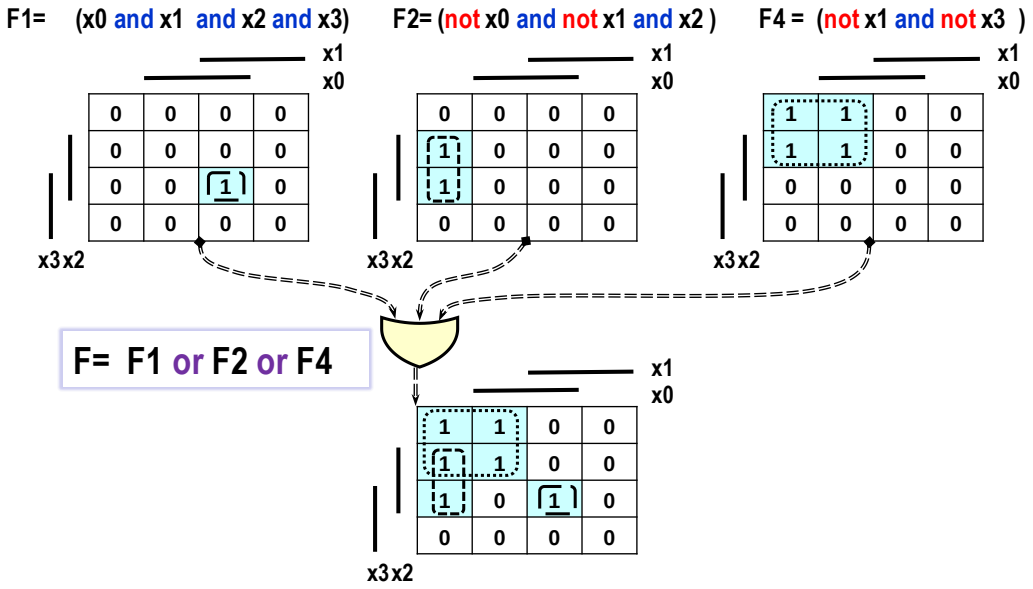
- AND implikant s Q členy určí 2^{N-Q} logických výstupů f() v '1', ostatní budou v '0'.
- OR implikant s Q členy definuje 2^{N-Q} logických výstupů f() v '0', ostatní budou v '1.

Jeden AND či OR implikant tak specifikuje výstupní hodnoty f() v počtu 1, 2, 4, 8, 16, ...

Pokud **Q=N** AND implikant určí v f() **jedinou** '1', bude současně i **mintermem.**
OR implikant naproti tomu popisuje **jedinou** '0', bude také i **maxtermem**.

Pozn. Pojem implikant se pojí k zadané množině N vstupů. Jde o přesný termín, ale v některých publikacích se AND implikanty pro jednoduchost nazývají vždy mintermy a OR zase maxtermy. Místo pojmu implikant se také zavádí i skupina (group) či jiné značení.

Uvedeme příklady AND implikantů pro N=4. Implikanty určují 1, 2 a 4 logických výstupů.



Obrázek 32 - Princip metody PoS

Pokud vytvoříme logickou funkci F pomocí spojení AND implikantů z F1, F2 a F4 operátory OR, pak její výstup bude daný logickým součtem.

Při minimalizaci rekonstruujeme členy tohoto výrazu. V Karnaughově mapě hledáme AND implikanty, a to největší možné, které již nelze více rozšířit, tzv. **primární implikanty**. Určují počet logických výstupů určený mocninou 2. Ty se implikantem pokryly.

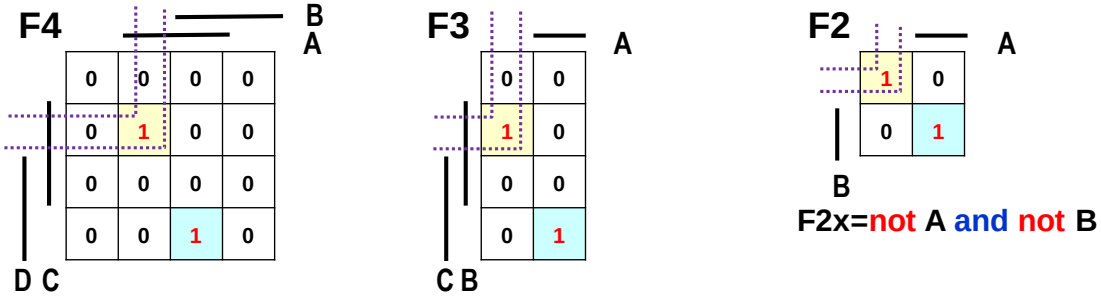
Až nalezneme AND implikanty, které plně pokrývají Karnaughovu mapu, pak jejich logické výrazy spojíme funkcí OR. Odtud dostala metoda svůj název **SoP - *Sum of Products*.**

Ukážeme si postup názorně na jednotlivých případech pokrytí 1, 2, 4 a 8 prvků.

3.3.3 Demonstrace situací při SoP

Pokrytí 1 prvku

Předpokládejme, že má tři funkce F4(A,B,C,D), F3(A,B,C) a F2(A,B) zadané jako Karnaughovy mapy. Výraz začneme budovat od žlutě zvýrazněných horních jedniček.



F4x=A and not B and C and not D **F3x=not A and B and not C**

Implikanty, jimiž jsou zde mintermy, napíšeme přímo z poloh '1' v KM. Každý dávající jen jednu '1' bude obsahovat právě tolik členů, kolik má příslušná funkce vstupních proměnných.

Vidíme, že žlutá '1' **je pod** A, **není pod** B, **je vedle** C a **není vedle** D. Slovo "není" zde implementujeme přidáním NOT před proměnnou:

F4x = A and not B and C and not D

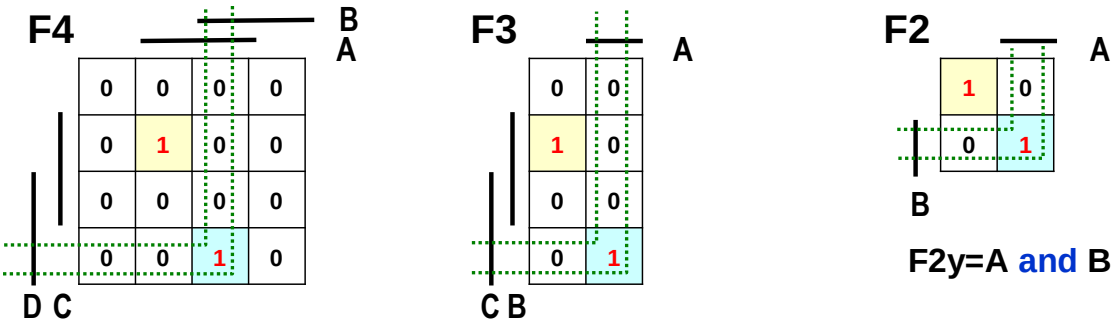
Analogicky se popíše minterm F3x: **není pod** A, **je vedle** B a **není vedle** C.

F3x = not A and B and not C

Stejným způsobem vyjádříme F2x: **není pod** A a **není vedle** B.

F2x = not A and not B

Podobný postup aplikujeme i na další zeleně zvýrazněnou 1 a dostaneme F4y, F3y a F2y



F4y=A and B and not C and D **F3y=A and not B and C**

Výsledné funkce obdržíme spojením jejich členů operací OR. Jakmile bude kterýkoli minterm v '1', i OR (výběr maxima) bude v logické '1':

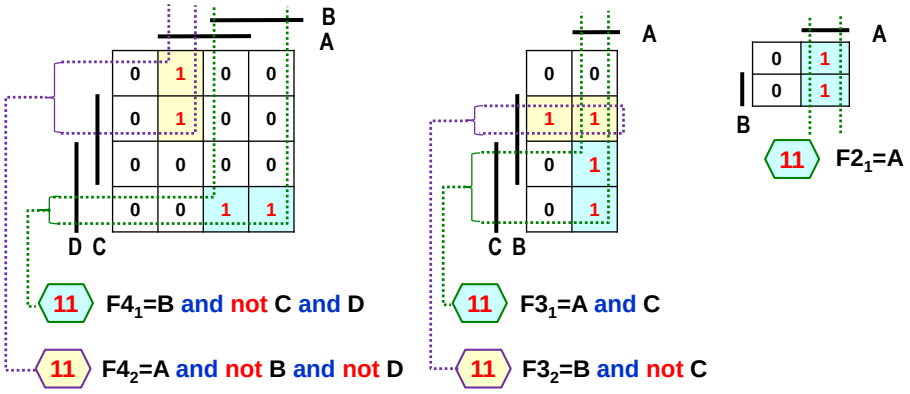
F4=F4x or F4y=(A and not B and C and not D) or (A and B and not C and D)

F3=F3x or F3y=(not A and B and not C) or (A and not B and C)

F2=F2x or F2y=(not A and not B) or (A and B)

Pokrytí 2 prvků

Máme-li dvě '1' u sebe, pak je pokryjeme AND implikantem, který bude mít o jednu proměnnou méně než počet vstupů dané funkce.



Obrázek 33 - Implikanty dvou '1'

Kdybychom každou zeleně zvýrazněné dolní '1' napsali pomocí dvou mintermů (AND implikantů pokrývajících jediný prvek), pak bychom obdrželi:

(levá dolní '1') $F4_{1L}$=A and B and not C and D

(pravá dolní '1') $F4_{1R}$=not A and B and not C and D

Po spojení obou mintermů operací OR dostaneme

$F4_1$ = $F4_{1L}$ or $F4_{1R}$= (A and B and not C and D) or (not A and B and not C and D)

Z výrazu vytkneme (B and not C and D)

$F4_1$ = (A or not A) and (B and not C and D)

a uplatníme pravidlo komplementarity, viz Obrázek 10 na str. 17.

$F4_1$ = **(B and not C and D)**

AND implikant se třemi členy, který jsme dostali, ale můžeme sestavit přímo z KM podle polohy obou zeleně zvýrazněných jedniček. Základní pravidlo Karnaughovy mapy totiž říká, že při pohybu o políčko kolmo či vodorovně se změní hodnota výhradně jediné vstupní proměnné. Díky tomu se v ní dá graficky aplikovat pravidlo komplementarity.

Zapíšeme, že obě zeleně zvýrazněné '1' **jsou pod** B, **nejsou vedle** C a **jsou vedle** D, jako

$F4_1$ = B and not C and D

Analogicky sestavíme i tříčlenný AND implikant i pro žlutě zvýrazněné horní '1', které se obě nacházejí **pod** A, ale **nejsou pod** B a **nejsou vedle** D:

$F4_2$ = A and not B and not D

Spojíme je operátorem OR, čímž dostaneme výslednou logickou funkci, kterou popisuje KM:

F4= $F4_1$ or $F4_2$ = (B and not C and D) or (A and not B and not D)

Prostřední KM, Obrázek 33 nahoře, popisuje logickou funkci se třemi proměnnými, která má 4 jedničky, ovšem uspořádané tak, že se nedají popsat společným implikantem. Uspořádání čtyř logických '1', u nichž to lze, si ukážeme dále. Zde musíme využít dvojicí implikantů se dvěma členy, tedy o jeden méně než počet proměnných funkce.

Obě zelené zvýrazněné dolní '1' leží **pod A** a **jsou vedle C,** tedy **$F3_1$=A and C**

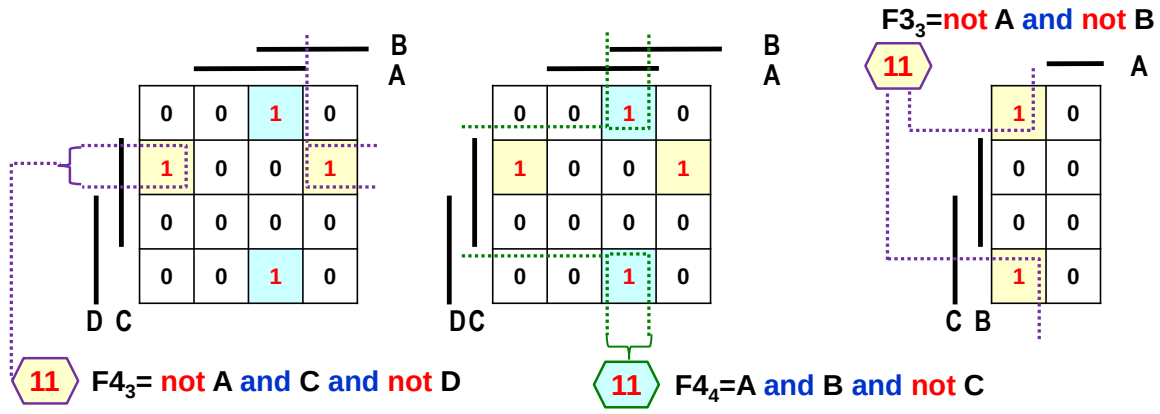
Obě žlutě zvýrazněné horní '1' se nacházejí **vedle B** a **nejsou vedle C, $F3_2$=B and not C**

Výsledná logická funkce bude

$$F3 = F3_1 \text{ or } F3_2 = (A \text{ and } C) \text{ or } (B \text{ and not } C)$$

Dvouvstupová logická funkce tvoří triviální případ. Obě jedničky jsou pod A, tedy $F2_1$=A.

Mají-li obě políčka na začátku i konci mapy '1', můžeme je pokrýt stejným implikantem, neboť Karnaughova mapa zachovává pravidlo změny hodnoty jen jedné vstupní proměnné i při vodorovném či svislém pohybu přes svůj okraj. Ve stejném sloupci sousedí horní políčko s dolním, což analogicky platí i v řádku.



Žlutě zvýrazněné jedničky $F4_3$ **nejsou pod** A, **jsou vedle** C a **nejsou vedle** D.

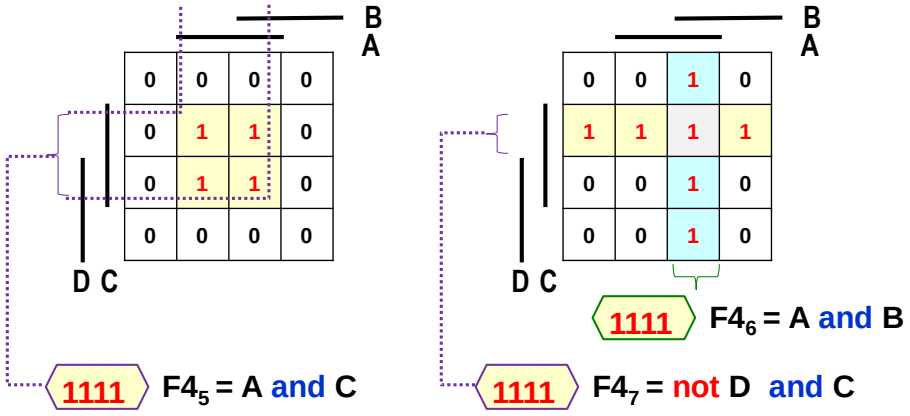
$$F4_3 = \text{not A and C and not D}$$

Zeleně zvýrazněné jedničky $F4_4$ zas leží **pod** A a **pod** B, ale **nejsou vedle** C.

$$F4_4 = \text{A and B and not C}$$

Pokrytí 4 prvků

Pokud vezmeme implikanty, které budou o 2 proměnné kratší než počet vstupních proměnných funkce, pak se k KM zobrazí jako 4 jedničky vedle sebe.

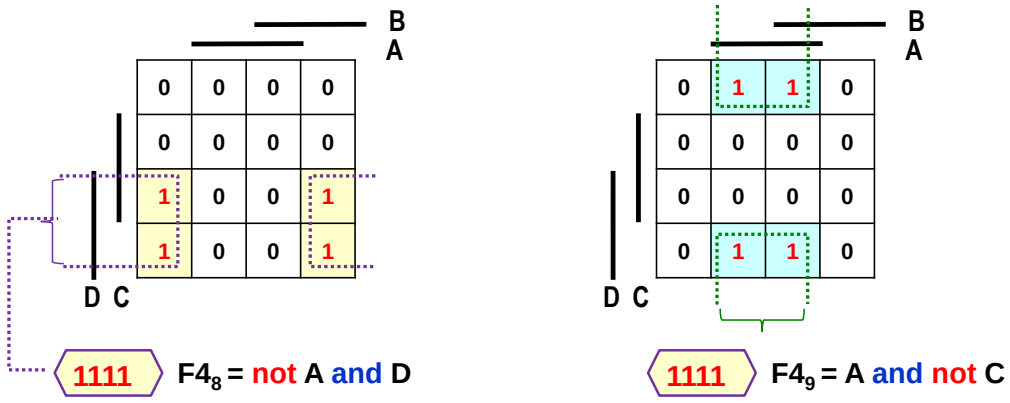


Opět implikanty sestavíme podle polohy, u $F4_5$ je **pod** A a **vedle** C, tedy F_5=A and C.

KM vpravo pokrytá dvojicí implikantů se sestaví stejným způsobem a výsledná funkce:

$$F4_{67} = F4_6 \text{ or } F4_7 = (A \text{ and } B) \text{ or } (\text{not } D \text{ and } C)$$

Šedě vyznačené políčko hodnot A='1', B='1', C='1' a D='0' leží v obou AND implikantech $F4_6$ a $F4_7$. Pokrývají ho oba. Funkce OR, výběr maxima, bude však v '1' při jednom i více svých vstupech v '1', a tak každou '1' lze zahrnout do více AND implikantů, jak se nám to hodí. I zde může nastat pokrytí přes hranu, jak ukazuje obrázek nahoře. Sestavíme opět z poloh.

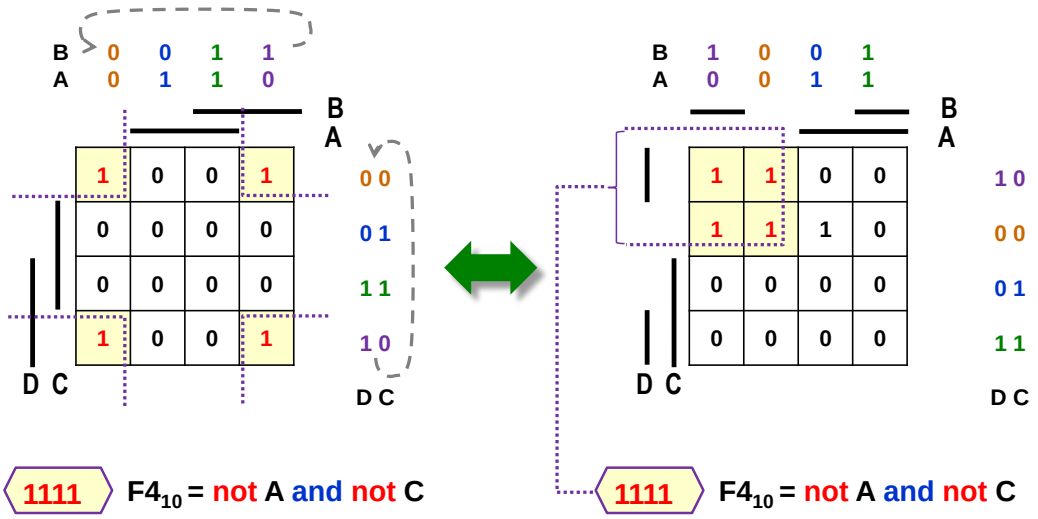


Obrázek 34 - Pokrytí 4 prvků přes hranu

Nejzajímavější případ nastává u pokrytí rohů, kdy funkce F_{10} (na obrázku dole vlevo) uplatňuje souvislost přes hrany. Čtyři jedničky v rozích **nejsou pod** A a **nejsou vedle** C, tedy

$$F_{10} = \text{not A and not C}$$

Pravidlo bude zřejmě, pokud si KM posuneme kruhově o 1 řádek a o 1 sloupec.



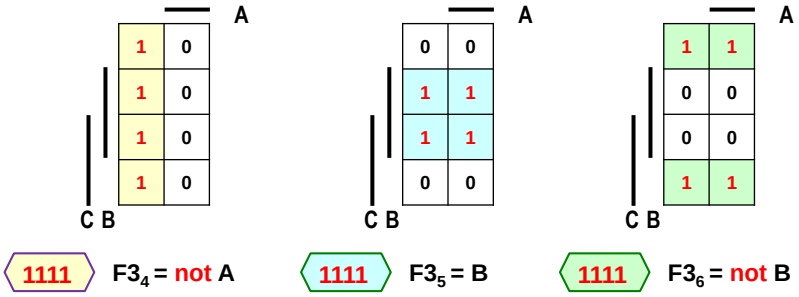
Obrázek 35 - Pokrytí rohů Karnaughovy mapy

Obrázek 35 definuje stejnou logickou funkci, a to jak v Karnaughově mapě vlevo, tak vpravo.

Pravá KM jen nepoužívá Grayův kód typu "*reflected binary*", ale jiný, nicméně stále Grayův, neboť splňuje požadavek na změnu hodnoty jedné proměnné při pohybu v řádku či sloupci včetně přechodů přes konce tabulky. A čtveřice '1' v níž již leží u sebe[7].

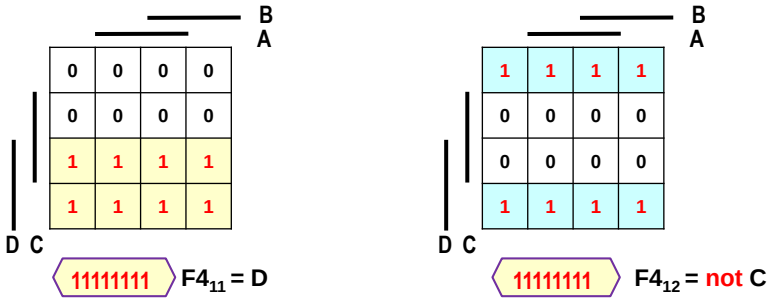
U KM závislé na třech vstupech se AND implikant menší o dva členy redukuje na pouhou jednu proměnnou a primitivní případy pokrytí.

[7] Na přednáškách se dozvíte, že Karnaughova mapa čtyř proměnných reprezentuje graf logické funkce na čtyřrozměrné krychli. Každý její vrchol má své čtyři sousedy. U KM tří proměnných jde pak o třírozměrnou krychli, tři sousední vrcholy. Z toho vyplývá i spojení přes pravý konec řádku k jeho levému začátku, či ve sloupci seshora dolů. Plášť krychle nemá konce a začátky, ty vzniknou až jeho rozvinutím do roviny. Poloha '0' a '1' v KM pak závisí na tom, od kterého vrcholu začneme krychli rozbalovat.



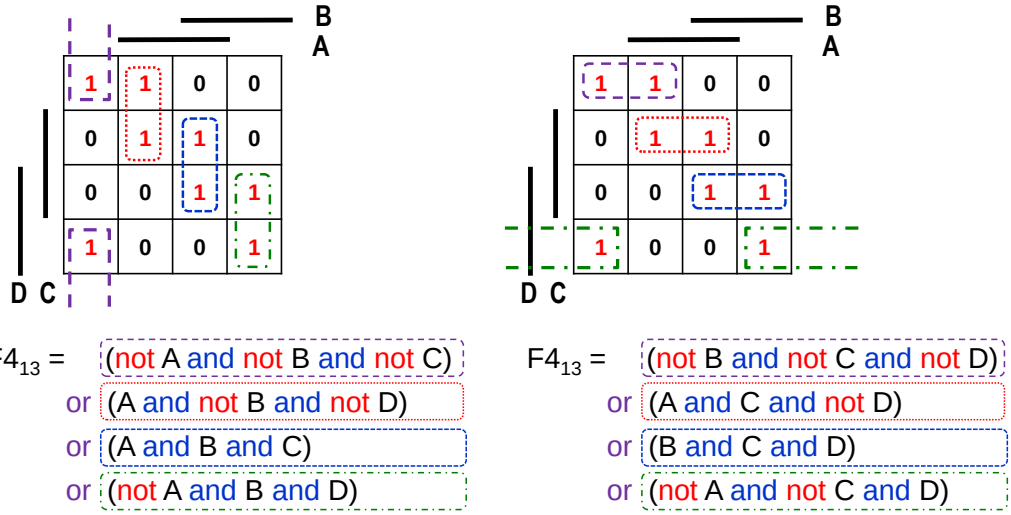
Pokrytí 8 prvků

U logické funkce se čtyřmi vstupy se použije AND implikant kratší o tři členy, tedy pouhá jedna proměnná, jak jsme odvodili v kapitole 3.3.2 na str. 37 (zde máme $8=2^{4-1}$).



Příklad: Pokrytí KM více implikanty:

Komplikovanější Karnaughovy mapy se pokrývají více implikanty, které opět volíme jako největší možné, tedy **primární implikanty**. Může existovat i více možností pokrytí.



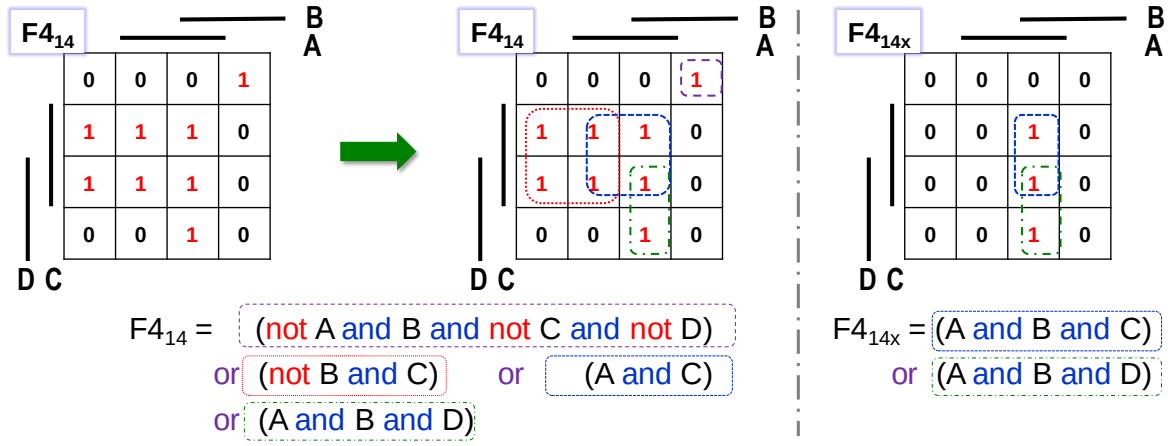
KM na obrázku nahoře se dá pokrýt různými způsoby, které jsou rovnocenné a mají stejný počet použitých členů.

F4$_{13}$ =(not A and not B and not C) or (A and not B and not D) or (A and B and C) or (not A and B and D)

=(not B and not C and not D) or (A and C and not D) or (B and C and D) or (not A and not C and D)

Zde vidíme nevýhodu logických výrazů. Mohou mít odlišný tvar, a přesto popisují stejnou logickou funkci. U KM platí, že stejné logické funkce mají totožné KM[8].

[8] Dalo by se tedy říci, že k logickým funkcím stačí sestrojit jejich KM coby důkaz jejich shody. Žel složitost KM roste s 2^N, kde N je počet vstupních proměnných. Lze je snadno použít u funkcí do 4 či 5 proměnných, kdy je



Obrázek 36 - Pokrytí více implikanty

Logická '1' v levém horním rohu v $F4_{14}$ je izolovaná a poryjeme ji tedy AND implikantem obsahujícím všechny vstupní proměnné, díky čemuž bude také mintermem. Prostřední blok se šesti '1' pokryjeme jako dva AND implikanty, každý z nich zahrne čtyři '1'. Dolní dvojici zapíšeme AND implikantem.

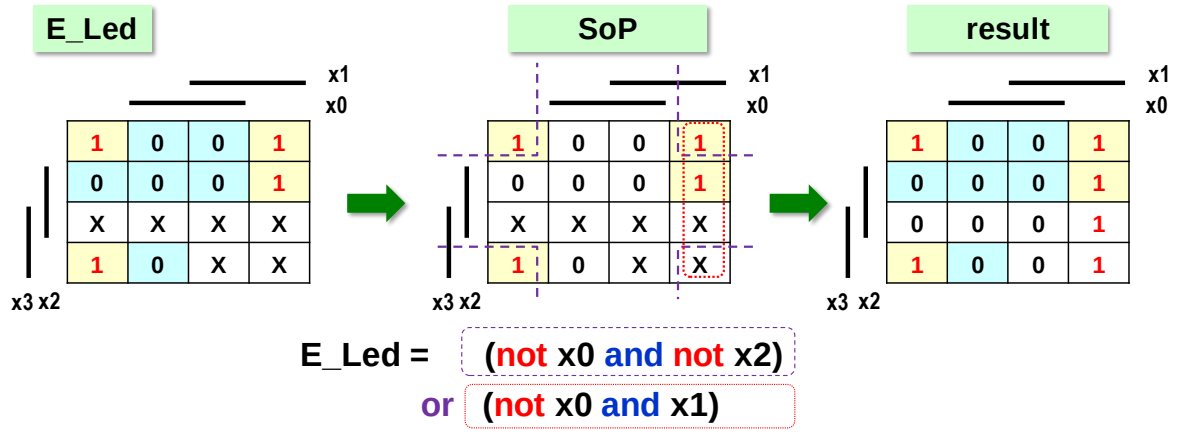
Pravá $F4_{14x}$ demonstruje situaci, kdy vedle sebe leží tři '1'. Jelikož implikanty pokrývají výhradně počty '1' dané mocninou 2, musíme využít dva překrývající se AND implikanty, každý z nich zahrne dvě '1'.

minimalizace KM rychlejší než vložení údajů do nějakého programu. Náročnější funkce se častěji zpracovávají s využitím dekompozic, což bude dále v kapitole o kombinačních obvodech.

Příklad: Využití *don't care*

Obsahuje-li logická funkce znaky *don't care*, pak máme svobodnou volbu, které z nich zahrneme do pokrytí, a jaké ne. Všechny, které pokryjeme metodou SoP, dodefinujeme na logické '1', ostatní budou '0'. Právě zde rozhodujeme totiž o hodnotě *don't care*.

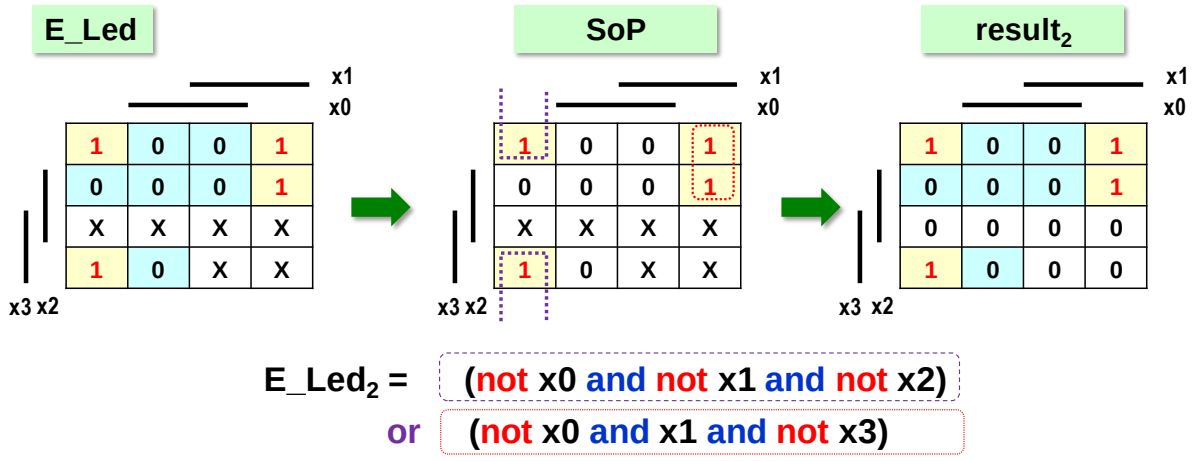
Obrázek 30 na str. 36 obsahuje Karnaughovu mapu e-LED 7segmentového displeje. Napíšeme ji jako logickou funkci pomocí dvou implikantů, přičemž každý zahrne čtyři logické '1'. V prvním využijeme dříve uvedené pokrytí rohů, viz Obrázek 35 na str. 41. V druhém zabereme celý pravý sloupec.



Obrázek 37 - Příklad na využití don't care

Naše pokrytí současně **dodefinovalo** všechna *don't care* (odložená rozhodnutí o hodnotě). Přiřadilo jimi '0' a '1', viz tabulka vpravo. Pokrytým '1', nepokrytým '0'.

Co se stane, když nezahrneme *don't care* do našich implikantů? Dostaneme akorát trochu složitější výrazy s více členy.



Všimněte si tady, že E_Led$_2$ logická funkce dává jiný výstup než předchozí E_Led, ovšem obě se shodují ve všech požadovaných hodnotách, tedy tam, kde KM předepisuje '0' a '1' .

Lze se ptát, zda **bude návrh E_Led$_2$ chybou?** Nevyužili jsme v něm přece možné pokrytí rohů a *don't care*!

Odpověď závisí na způsobu fyzické realizace logické funkce. V počátcích logiky, kdy se vše zapojovalo pájkou a drátky, by se návrh E_Led$_2$ nazval hrubou chybou, protože vyžaduje více hradel.

Dnes se rovnice logických výrazů vloží do návrhového prostředí, které samo provede propojení. Pokud bude cílovou fyzickou realizací FPGA, pak se v něm využijí konfigurovatelné logické elementy, LE, které využívají LUT, *Lookup Table*. Ty rozebereme ve stati o nitru FPGA, a to v kapitole 5.4 začínající na str. 86. Každý LE obvykle umí logickou funkci se čtyřmi i více vstupy. Spotřebuje se tedy právě jeden LE, a to jak na E_Led, tak na E_Led_2.

Návrh E_Led_2 dnes není chybou. Vždyť poskytl námi požadovaný výstup, což klademe za hlavní podmínku. Můžeme ho jen označit za neoptimální návrh, jelikož se zbytečně píší delší výrazy.

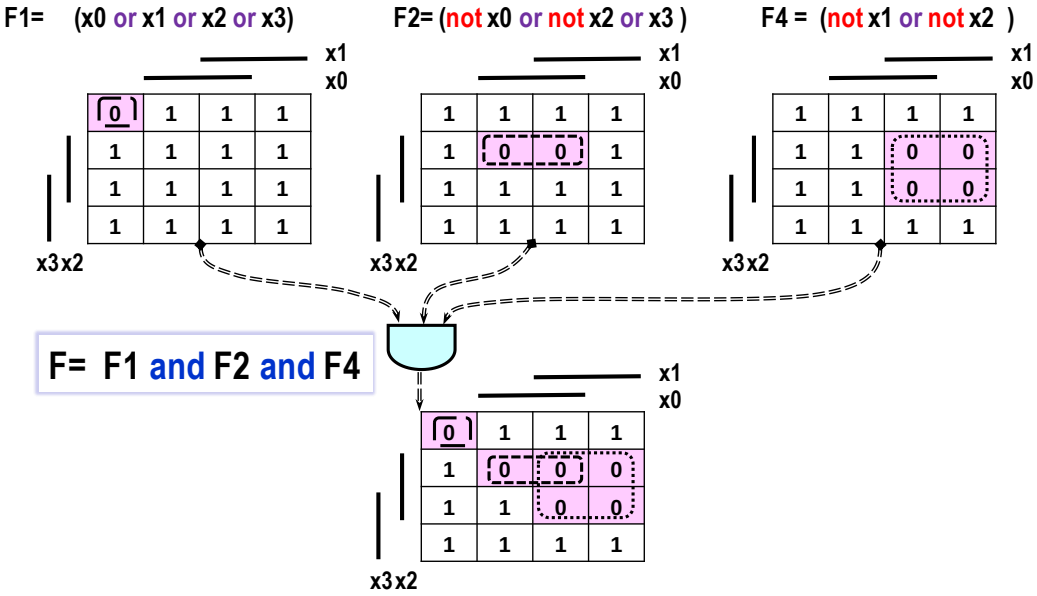
Otázka: *A lze návrhovému prostředí obvodu zadat E_Led logickou funkci i bez pokrývání Karnaughovy mapy?*

Ano. V HDL jazycích pro návrh obvodů lze logickou funkci definovat nejen logickým výrazem, ale i výčtem hodnot výstupů, a dalšími prostředky. Logický výraz však bývá mnohdy kratší a přehlednější cestou v řadě případů. Kvůli tomu vysvětlujeme minimalizaci.

3.3.4 Minimalizace Karnaughových map metodou PoS

Obsahuje-li Karnaughova mapa hodně jedniček, pak jejich pokrývání nemusí přinášet optimální výsledky. Pokud obsahuje mnohem méně logických '0', rychleji se pokryje OR implikanty. Ty spojíme AND operací (výběrem minima), vůči níž je '0' (minimum) agresivním prvkem. Je-li kterýkoli OR implikant v '0' i výstup bude '0'.

Metoda se kvůli tomu nazývá **PoS** - ***Product-of-Sum*** a demonstruje ji obrázek dole. Postup je stejný. Hledáme primární OR implikanty, tedy největší možné,. Uplatníme vše, co známe.

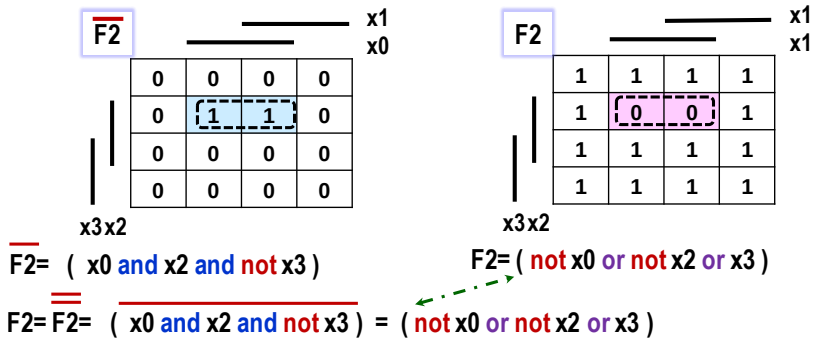


Obrázek 38 - Princip metody PoS

OR implikanty ale odvozují operátory NOT opačně oproti AND implikantům.

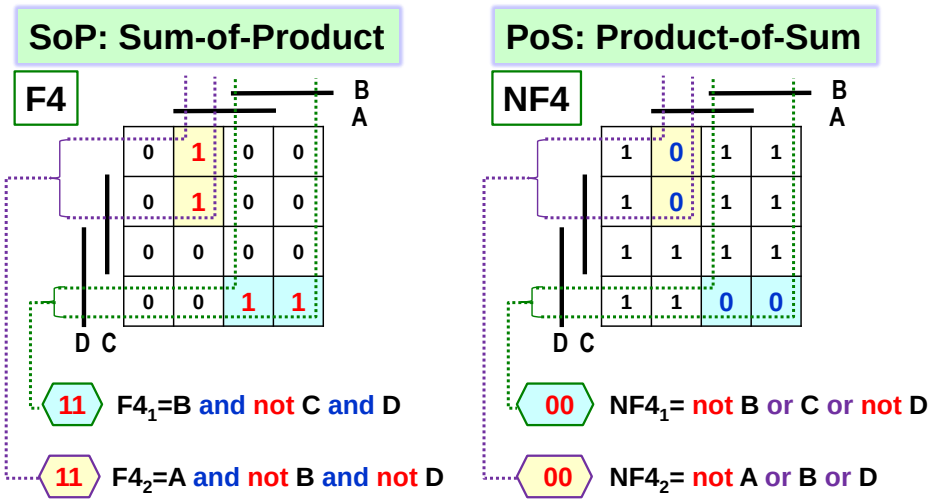
Například dvojice '0' pokrytá OR implikantem F2= (not x0 or not x2 or x3) je pod x0 a vedle x2, což se v OR implikantu zapisuje s operátory NOT. Naproti tomu vedle x3 nemá NOT před x3.

Rozdíl vyplývá z De Morganova teorému (str. 19). Pokryjeme-li negovanou F2 metodou SoP, pak výsledný výraz můžeme negací převést na původní F2, čímž dostaneme výraz shodný s pokrytím OR implikanty.



Obrázek 39 - Srovnání pokrytí AND a OR implikantem

Ukážeme si ještě analogii k pokrytí demonstrovanému na Obrázek 33 na str. 39.



Obrázek 40 - SoP, pokrytí '1', versus PoS, pokrytí '0'

OR implikant funkce NF_1 se vytvoří dle jeho pozice **je pod** B, **není vedle** C a **je vedle** D, přičemž unární NOT píšeme před proměnnými v opačných situacích než u AND implikantů:

$$NF4_1 = \text{not } B \text{ or } C \text{ or not } D$$

Podobně vyjádříme i druhý OR implikant jako **je pod** A, **není pod** B a **není vedle** D

$$NF4_2 = \text{not } A \text{ or } B \text{ or } D$$

Výslednou funkci pak sestavíme s obou implikantů jejich spojením AND:

$$NF4 = NF4_1 \text{ and } NF4_2 = (\text{not } B \text{ or } C \text{ or not } D) \text{ and } (\text{not } A \text{ or } B \text{ or } D) \qquad (n1)$$

Znovu si ukážeme souvislost s De Morganovým teorémem (str. 19). Napíšeme si NF4 jako negaci F4 pokryté implikanty.

$$NF4 = \text{not } F4 = \text{not } (F4_1 \text{ or } F4_2) \qquad (n2)$$

$$= \text{not } ((B \text{ and not } C \text{ and } D) \text{ or } (A \text{ and not } B \text{ and not } D)) \qquad (n3)$$

Nyní rozepíšeme not před závorkou dle De Morganova teorému, čímž operátor or změníme na and a not posuneme před oba členy výrazu. V dalším kroku opakujeme i pro ně.

$$NF4 = \text{not } (B \text{ and not } C \text{ and } D) \text{ and not } (A \text{ and not } B \text{ and not } D) \qquad (n4)$$

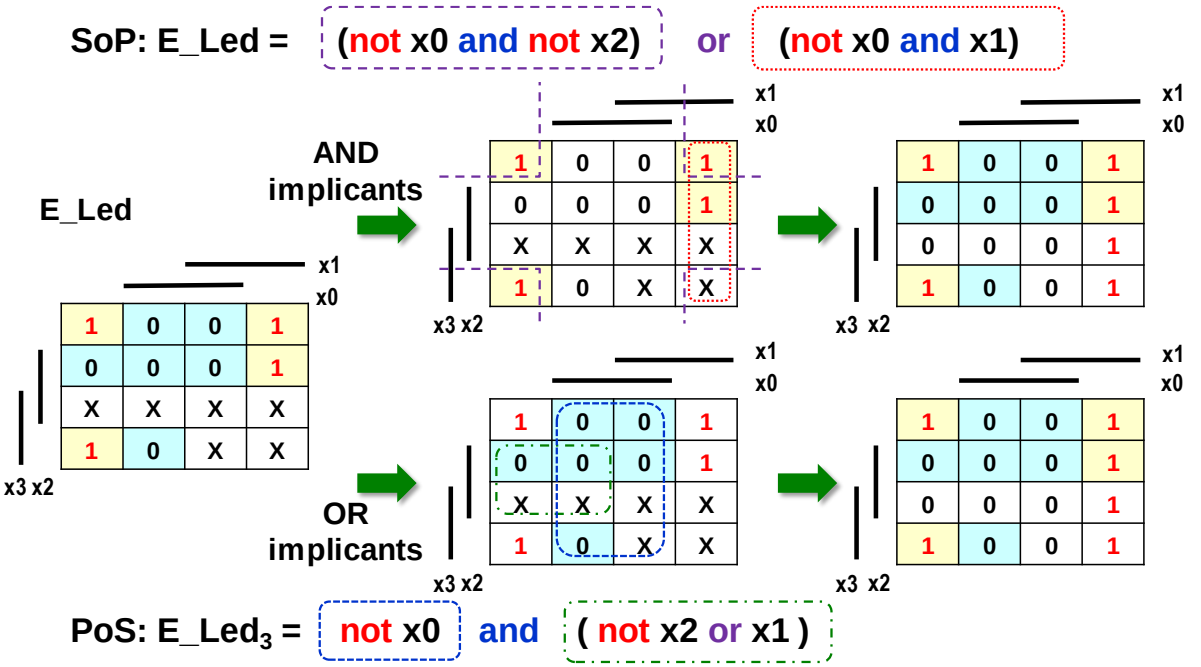
$$= (\text{not } B \text{ or } C \text{ or not } D) \text{ and } (\text{not } A \text{ or } B \text{ or } D) \qquad (n5)$$

Vidíme, že výraz (n5) je shodný s (n1). DeMorganův teorém má tedy svou analogii v pokrytí negované KM pomocí PoS metody a následné negace výsledku.

3.3.5 Srovnání pokrytí s užitím *don't care*

Pokrytí AND implikanty, tedy pomocí '1', musí zahrnout všechny '1' a navíc může některé *don't care*, ty se pak dodefinují na logické '1' ostatní budou v '0'. Při práci s OR implikanty musíme pokrýt všechny '0' a můžeme přidat i vhodná *don't care*, které se tím dodefinují na '0', zatímco ostatní budou '1'.

Srovnáme uvedené způsoby pokrytí E_Led 7segmentového displeje:

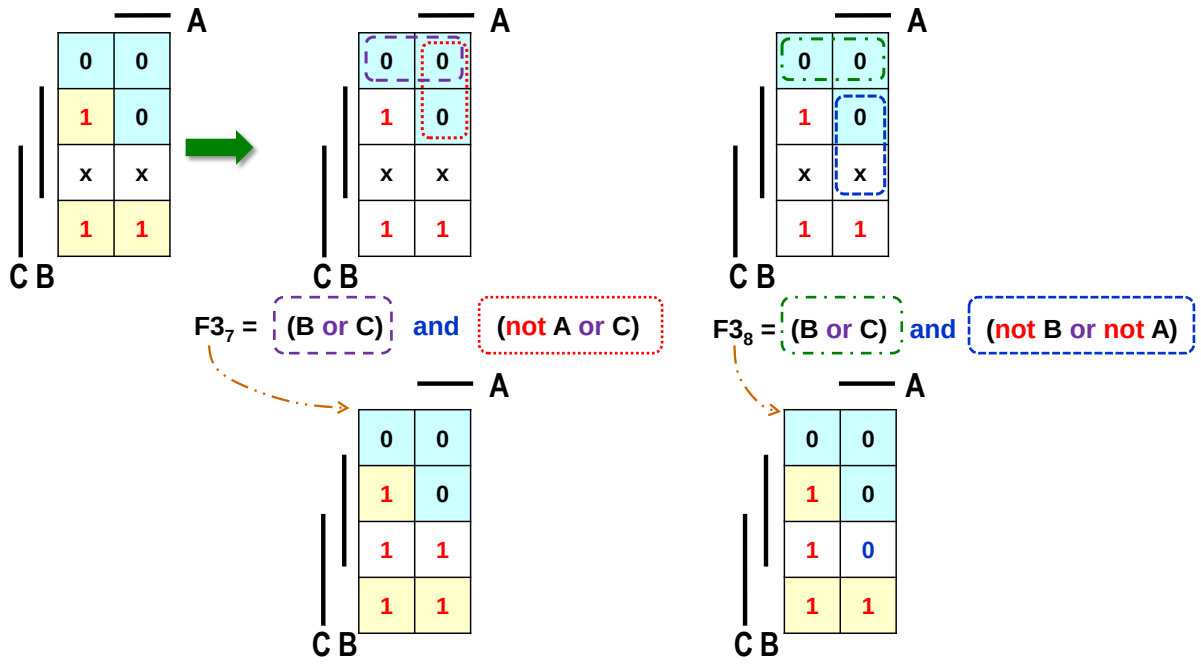


Logická funkce E_Led_3 potřebuje dva OR implikanty, jeden leží pod x0, bude tedy not x0, zatímco druhý je vedle x2 a není pod x1, což dává not x2 or x1. Výsledná funkce:

$$E\_Led_3 = \text{not } x0 \text{ and } (\text{not } x2 \text{ or } x1)$$

Nezahrnutá *don't care* se dodefinovala na '1', čímž jsme dostali stejnou výslednou funkci jako při pokrytí '1', což prokážeme, když členem not x0 roznásobíme závorku.

Další příklad ukazuje různá pokrytí '0', tedy OR implikanty. Obě vyjádření jsou správná, avšak odlišným dodefinováním *don't care* vedou na různé logické funkce, avšak shodné v předepsaných '0' a '1':



Obrázek 41 - Dodefinování don't care při pokrytí '0' (PoS)

3.3.6 Shannonova expanze Karnaughovy mapy

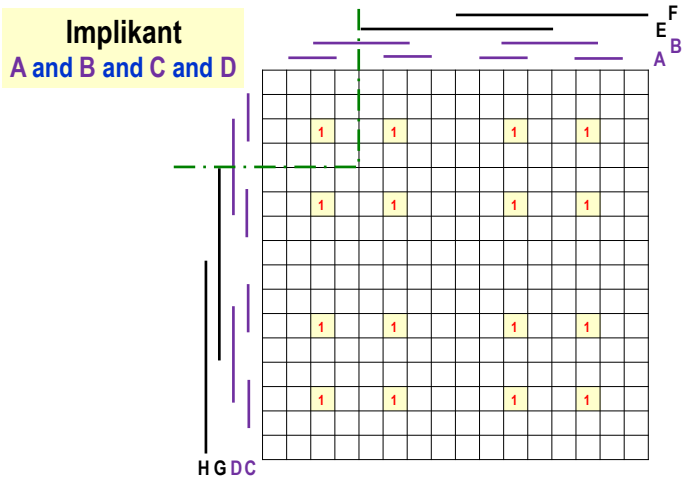
Karnaughova 4 proměnných je největší, u níž se logické '1' všech AND implikantů (či '0' OR implikantů) objeví u sebe, uvažujeme-li i přechody přes konce tabulky.

U větších KM budou rozházené. Obrázek dole ukazuje mapu 8 proměnných, mapa 4 proměnných je jejím výsekem.

Implikant s Q=4 členy určí v logické funkci N=8 proměnných

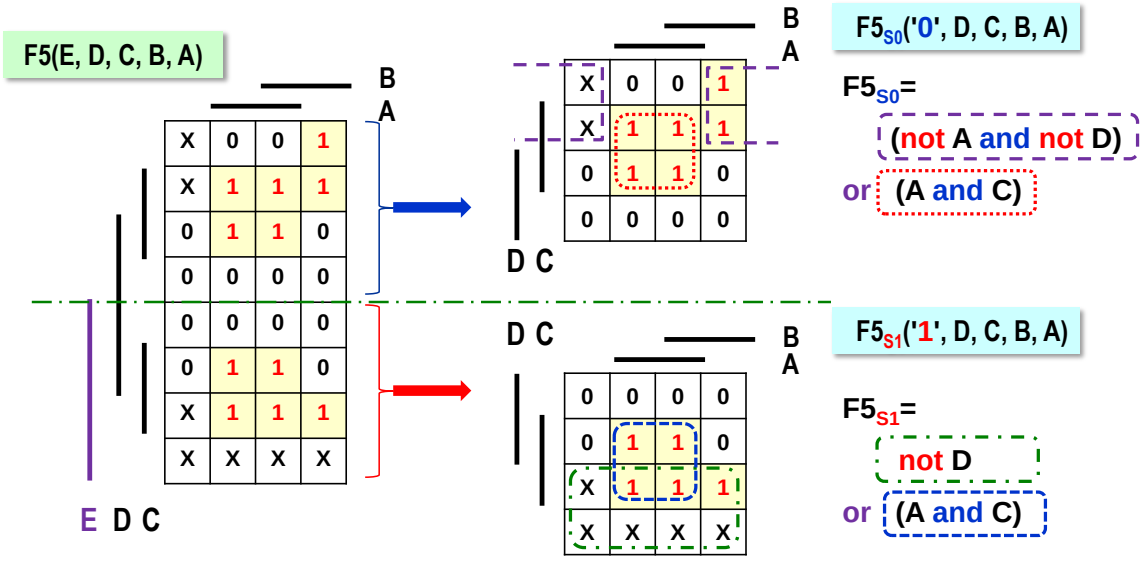
2N-Q =28-4 =24=16

logických '1'. Mapa vpravo ukazuje logické '1' od AND implikantu A and B and C and D. Ty již neleží vedle sebe, což komplikuje pokrývání.



Obrázek 42 - Karnaughova mapa 8 vstupních proměnných

Co když však máme větší KM a zrovna nám chybí vhodný program? Pak ji můžeme roztrhnout na poloviční KM podle jedné proměnné. Na obrázku dole jsme si zvolili E. V horní KM bude E='0', zatímco v dolní E='1'. Obě dílčí KM rozměru 4x4 snadno minimalizujeme.



Výsledek spojíme tak, aby se výstupní hodnoty $F5_{S0}$ se uplatnily jedině při E='0', zatímco $F5_{S1}$ jedině při E='1', čehož dosáhneme operací or a přidání not E a E před dílčí funkce.

$$F5_S = (\text{not } E \text{ and } F5_{S0}) \text{ or } (E \text{ and } F5_{S1}) \quad \text{(F5-1)}$$

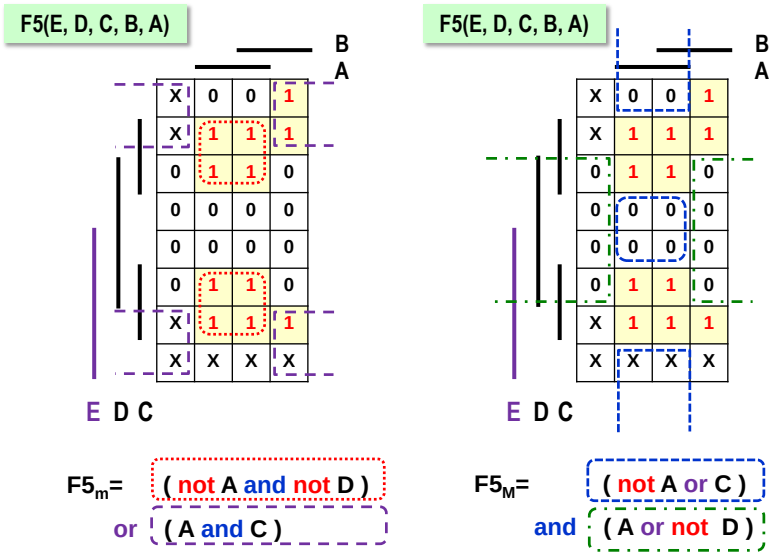
$$F5_S = (\text{not } E \text{ and } ((\text{not } A \text{ and not } D) \text{ or } (A \text{ and } C))) \text{ or } (E \text{ and } (\text{not } D \text{ or } (A \text{ and } C))) \quad \text{(F5-2)}$$

Je výsledek optimální? Není. Přímá minimalizace celé KM ukazuje, že člen $F5_{S0}$ by sám pokryl všechny '1'.

$$F5_m = (\text{not } A \text{ and not } D) \text{ or } (A \text{ and } C) = F5_{S0} \quad \text{(F5-3)}$$

Můžeme ještě zkusit aplikovat OR implikanty:

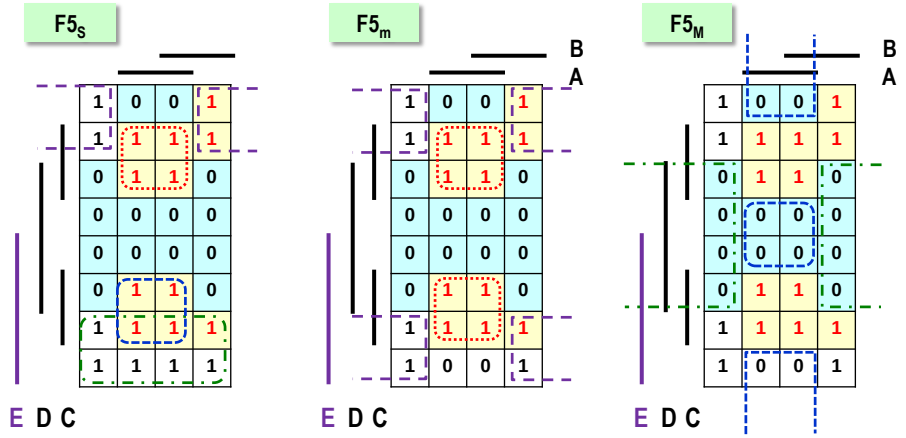
$F5_M$ = (not A or C) and (A or not D) (F5-4)



Obrázek 43 - Přímá minimalizace F5

Získané logické $F5_m$ z (F5-3) i $F5_M$ daná (F5-4) se logicky rovnají. SoP metoda $F5_m$ (pokrytí '1') dodefinovala zahrnutá *don't care* na '1' ostatní na '0'. PoS metoda $F5_M$ (pokrytí '0') naopak specifikovala pokrytá *don't care* na '0', nezahrnutá na '1', ale se shodným výsledkem.

SoP v $F5_S$, rozkládající tabulku Shannonovou expanzí, dává odlišný výsledek, protože pokryla jiná *don't care*. Všechny tři logické funkce se však shodují v předepsaných '0' a '1', což je nejdůležitější.



Obrázek 44 - Srovnání výsledků F5

U větších KM lze dělení podle zvolené proměnné opakovat tak dlouho, dokud se nedostaneme na rozměr 4x4, tedy na KM logické funkce 4 proměnných.

Pokud si však žádáme zcela dokonalou optimalizaci, logické funkce s pěti a více proměnnými raději svěříme počítači. Sice ztratíme čas vkládáním hodnot '0', '1' a *don't care* do programu či návrhového prostředí, ale výsledek bude bez chyb, které nám již hrozí u rozměrnějších KM díky nespojitému rozmístění jejich implikantů.

Další použití velmi důležité **Shannonovy expanze** si ukážeme ještě na str. 52, v úkolu 3.

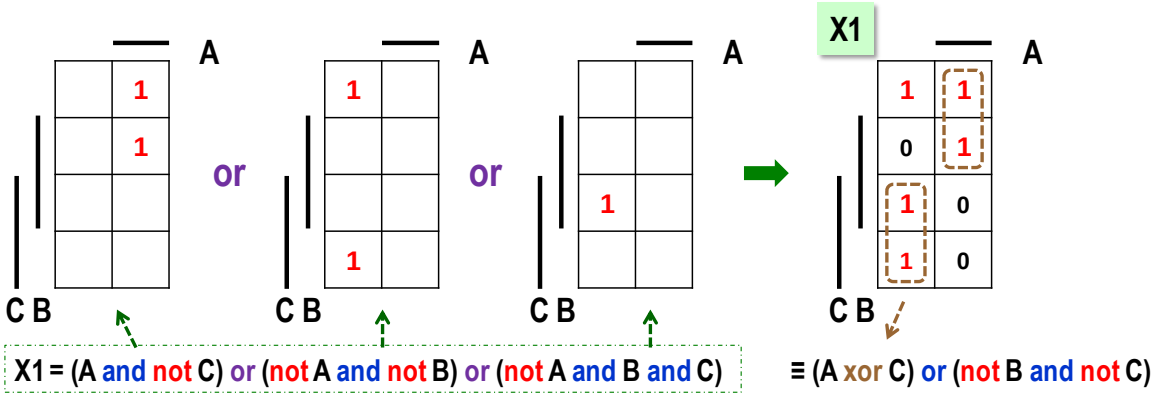
3.4 Použití Karnaughových map k vyčíslení logické funkce

3.4.1 Úkol 1: Využijte SoP ke stanovení KM logické funkce

X1 = (A and not C) or (not A and not B) or (not A and B and C)

Můžeme dosazovat hodnoty za A, B, C a postupně vyčíslit funkci, což bude ale zdlouhavé a s rizikem nechtěných chyb. Pokud si však všimneme, že výraz X1 má tvar SoP (pokrytí '1'), pak vyřešíme vmžiku. Nakreslíme prázdnou Karnaughovu mapu funkce 3 vstupních proměnný a do ní zaneseme '1' generované jednotlivými implikanty. Ostatní políčka budou '0'.[9]

Logické '1' **(A and not C)** počátečního implikanty obě **leží pod** A a **nejsou vedle** C. Ostatní výrazy se sestaví analogicky. Pokud si ještě vzpomeneme na funkci XOR, pak ve finální mapě vpravo zkrátíme výraz.

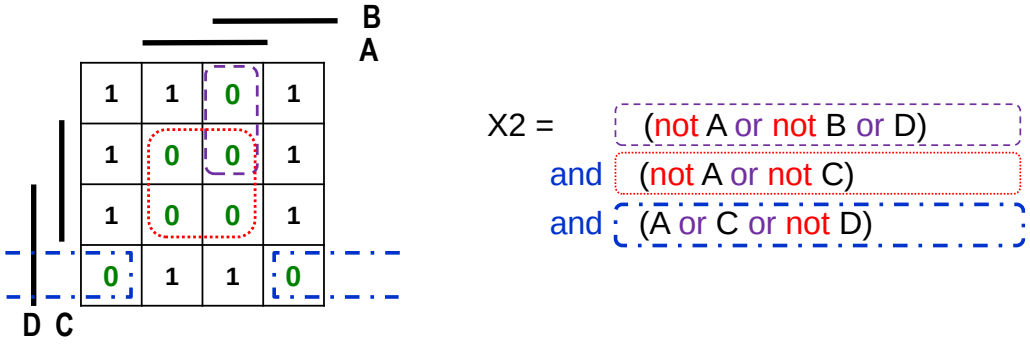


3.4.2 Úkol 2: Využijte PoS k vytvoření KM logické funkce:

X2 = (not A or not B or D) and (not A or not C) and (A or C or not D)

Tentokrát má výraz tvar PoS (pokrytí '0'). Nakreslíme tedy prázdnou KM logické funkce čtyř proměnných a zaneseme do ní výstupy OR implikantů. Zde si dáváme jen bedlivý pozor na to, že operátor not stojí před proměnnými v opačné situaci než u AND implikantů.

První implikant (not A or not B or D) je tedy **pod** A, **pod** B a není vedle D. V obrázku má fialové orámování. Další opět sestavíme analogicky. Nevyplněná políčka budou logické '1', neboť PoS metoda pokrývá '0'.



[9] Připomínáme, že symboly *don't care* se tady nemohou objevit na výstupu logické funkce, jelikož znamenají odložené rozhodnutí o hodnotách. K jejich volbě však již došlo během sestavování logického výrazu. V něm je o nich již dávno rozhodnuto.

3.4.3 Úkol 3: Shannonovou expanzí vyčíslete logickou funkci

Y5(A,B,C,D,E) = (A and D) or (A and not B and E) or (not A and E) or (not C and not E)

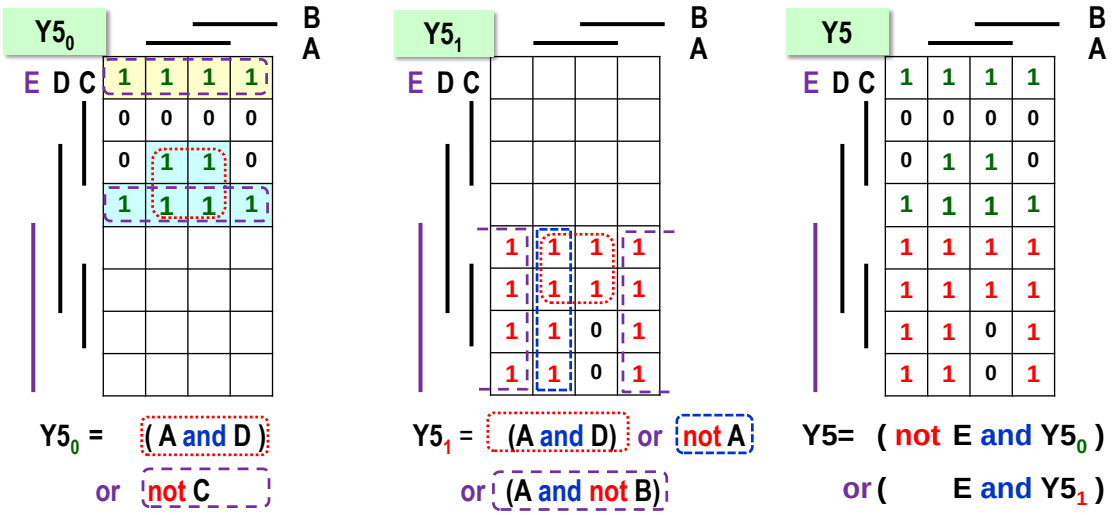
Funkce má 5 vstupních proměnných a volá již po počítačovém řešení. Pokud z nějakého důvodu provádíme její ruční zpracování, lze provést její redukci **Shannonovou expanzí**.

Vytvoříme si dvě funkce, první dosazením '0' za E a druhou pak substitucí '1' za E, čímž roztrhneme pravdivostní tabulku na dvě poloviny.

$Y5_0$=Y5(A,B,C,D,'0') = (A and D) or (A and not B and '0') or (not A and '0') or (not C and not '0')
= (A and D) or (not C and '1')
= (A and D) or not C

$Y5_1$=Y5(A,B,C,D,'1') = (A and D) or (A and not B and '1') or (not A and '1') or (not C and not '1')
= (A and D) or (A and not B) or not A or (not C and '0')
= (A and D) or (A and not B) or not A

Obě funkce mají SoP tvary. Implikanty nalezené logické funkce $Y5_0$ vyplníme horní část Karnaughovy mapy 5 proměnných, v níž je E='0'. Pomocí implikantů $Y5_1$ vytvoříme dolní část, kde má E='1'. Na obrázku rozkresleno kvůli přehlednosti, ale lze psát přímo do výsledné KM.



Obrázek 45 - Použití Shannonovy expanze

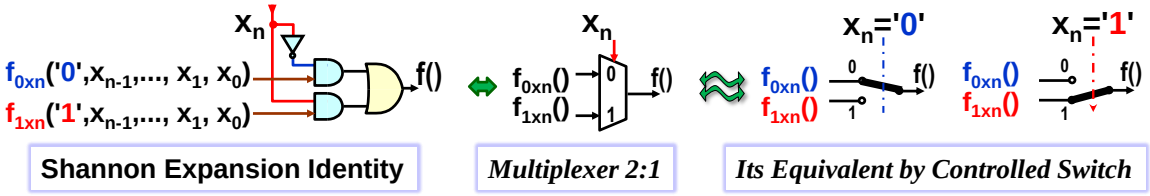
Funkce Y5 se složila jako **Y5 = (not E and $Y5_0$) or (E and $Y5_1$),** tedy stylem SoP pokrytí. Představuje zápis ve formě Shannonovy identity.

Máme-li logickou funkci $f(x_n,x_{n-1},..., x_1, x_0)$, pak postupujeme analogicky k příkladu nahoře. Vybereme si vhodnou proměnnou, například x_n, a funkci f() rozložíme Shannonovou expanzí, tedy pouhým dosazením '0' a '1' za x_n. Dostaneme $f_{0xn}('0',x_{n-1},..., x_1, x_0)$ a $f_{1xn}('1',x_{n-1},..., x_1, x_0)$, které se nazývají Shannonovy kofaktory podle x_n, angl. *Shannon cofactors of f() with respect to* x_n.

Původní funkci lze pomocí nich vyjádřit Shannonovou identitou jako:

$$f(x_n,x_{n-1},\ldots, x_1, x_0)= (\text{not } x_n \text{ and } f_{0n}('0',x_{n-1},\ldots, x_1, x_0)) \text{ or } (x_n \text{ and } f_{1n}('1',x_{n-1},\ldots, x_1, x_0))$$

Pokud si identitu nakreslíme ve formě schématu, pak vidíme, že ve skutečnosti popisuje multiplexor 2:1. Multiplexory budou tématem pozdější kapitoly 5.3 na str. 82.



Obrázek 46 - Shannonova expanze

Kofaktory lze následnými expanzemi rozkládat na ještě jednodušší kofaktory s méně rozsáhlými KM, tedy na menší pravdivostní tabulky, třeba až o jedné proměnné, čímž provedeme vyčíslení logické funkce.

Ve výpočetní technice se Shannonovou expanzí vytvářejí **BDD**, *Binary Decision Diagrams* [10] , mocný nástroj například při verifikaci programů, a jinde, kde se opakovaně vyčísluje mnoho logických výrazů. Existuje i výběr *freeware* BDD knihoven pro běžné programovací jazyky.

Poznámka: Shannonova expanze dává výsledky, které závisí na výběru proměnné, podle níž rozkládáme. Heuristicky se kvůli tomu začíná od takové proměnné, aby se vyřadilo maximum členů logické funkce. I tak nemusí vždy klesat složitost kofaktorů. Některé kombinační logické funkce ani nemají žádný vhodný rozklad, třeba sčítačky či násobičky, ač právě u nich by se nám hodil:-) Shannonovu expanzi lze tedy zjednodušit jenom ***podmnožinu*** *logických funkcí.*

[10] Blíže viz například: https://en.wikipedia.org/wiki/Binary_decision_diagram

3.4.4 Úkol 4: Zjednodušte výraz

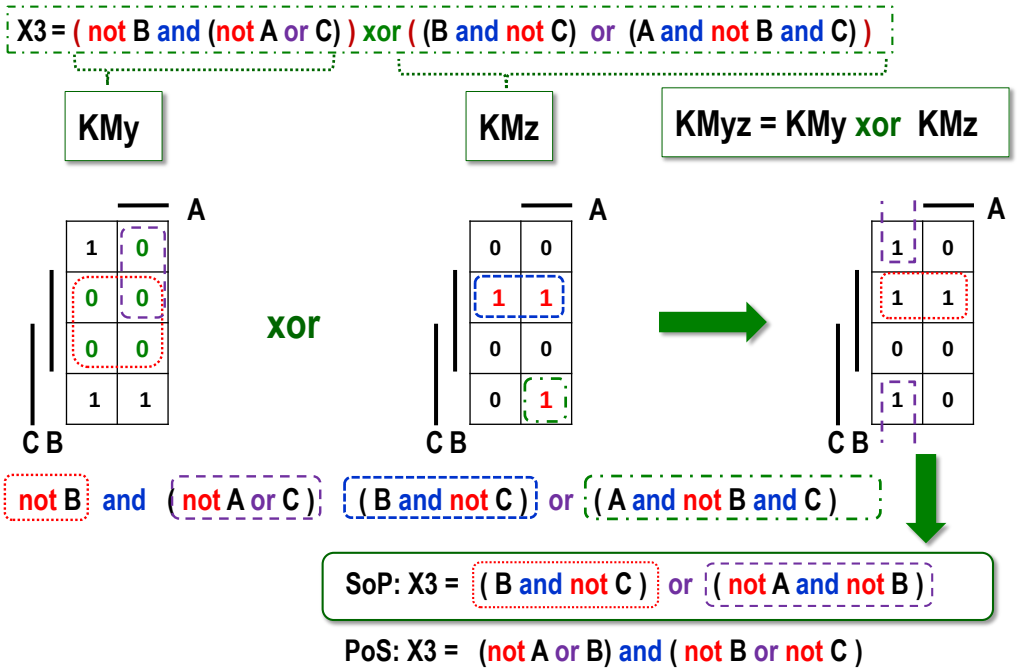
X3 = (not B and (not A or C)) xor ((B and not C) or (A and not B and C)) (xr1)

Funkce xor komplikuje operaci. Lze ji sice rozepsat podle vztahu z kapitoly 2.3.1 na str. 23:

Y xor Z = (Y and not Z) or (not Y and Z) (xr2)

ale za členy Y a Z bychom dosazovali výrazy, což by jen komplikovalo rovnici.

Zkusíme raději KM-trik. Vytvoříme si Karnaughovy mapy KMy a KMz členů Y a Z funkce xor. Z nich pak vypočteme výslednou KMyz. Výraz KMy (levý ve funkci xor), má tvar PoS (pokrytí '0'). Zapíšeme tedy '0' podle OR implikantů a zbylá políčka doplníme na '1'. Pravý výraz xor odpovídá SoP (pokrytí '1'), a tak podle AND implikantů vyplníme '1', ostatní doplníme na '0'.



Výslednou KMyz sestavíme ryze mechanicky políčko po políčku. Víme přece, že xor dává výstup '1' při lichém počtu '1' na svých vstupech, a tak píšeme '1' do těch políček KMyz, v nichž bude KMy různá od KMz, při shodě jejich hodnot vložíme '0'.

Finální KMyz převedeme na výraz třeba metodou SoP (pokrytí '1'), ale lze použít i PoS.

3.5 Počítačové minimalizační algoritmy

V částech o minimalizaci jsme několikrát zmínili počítačové algoritmy. Nebudeme detailně rozebírat jejich přesnou algoritmizaci, které se zasvěceněji věnují jiné publikace, ale uvedeme jen vlastnosti nejznámějších nástrojů.

- **Metoda Quine-McCluskey** funguje analogicky k pokrývání logických '1' postupem SoP, akorát v numerické formě. Vyjde z výčtu indexů '1' a členů *don't care*, tedy z popisu ve stylu *on-set*, viz str. 31, který ve své podstatě určuje mintermy pokrývající jediný prvek KM. Mezi nimi se hledají členy, které se liší jen v negaci jedné logické proměnné, tedy postupem ze str. 39. Z nich se sestaví všechna možná pokrytí dvou prvků, která se využijí k pátrání po pokrytí čtyř členů, a tak dále, dokud se daří najít nějaké možné sloučení. Výsledkem je seznam **primárních implikantů**, tedy maximálních možných. V závěru se z nich vybere vhodné pokrytí všech zadaných '1'. Doba běhu metody závisí

na počtu členů ve výrazech, které se zjednodušují. Maximální složitost metody může být až $O(3^N / \sqrt{N})$, kde N je počet vstupních proměnných[11].

- Profesionální nástroje používají třeba algoritmus **Espresso**[12], který heuristikami manipuluje s logickými krychlemi. V drtivé většině případů poskytne výsledek za zlomek času oproti běhu Quine-McCluskey metody. U malých logických funkcí najde jejich optimální tvar, avšak u velkých, o stovkách proměnných, předloží leda nějaké řešení, a to ještě někdy. U komplikovaných zadání už neskončí jeho běh v rozumném čase.
- Dalších algoritmem je například **Boom**[13] , který pracuje se stromovými strukturami a dokáže často vypátrat řešení ještě rychleji než *Espresso*, avšak také ne vždy.

Proč se rozebírala minimalizace logických funkcí? Vše zvládne počítač!

Existuje hned několik důvodů:

- Naši představu o logické funkce lze samozřejmě zadat návrhovému prostředí i pomocí seznamu výstupů její pravdivostní tabulky, což se někdy dělá i v profesionálních návrzích, třeba u dekodérů pro 7-segmentový displej. Jednodušší funkce se snáze vyjádří logickými výrazy, z nichž se často lépe pozná jejich chování. Sekvence výstupů '0' a '1' v jejich pravdivostních tabulkách málokdy něco naznačí.
- Pravdivostní tabulky logických funkcí rostou exponenciálně s počtem proměnných. Sebelepší algoritmus nezmění podstatu minimalizace, jejíž složitost patří mezi NP-úplné problémy[14]. Ani přidané heuristiky nezvládnou ve všech případech vyřešit mimořádně komplikované funkce v přijatelném čase. Takové se musí nutně dekomponovat na menší dílčí části, což si žádá porozumění dostupným stavebním prvkům, téma kapitoly 5.
- Zde musíme ještě zmínit, že existují i kombinační logické funkce, které mají vesměs izolované implikanty neslučitelné s jinými, a tak jejich minimalizace nemá smysl, například jde o pravdivostní tabulky sčítaček. Všechny takové případy musíme rozložit na menší bloky, což si žádá znalost logických funkcí coby vhodných stavebních prvků.

---

[11] Popis metody stručně uvádí *Wikipedia*, detailně ji rozebírá i s C kódem článek Banerji. S.: *Computer Simulation Codes for the Quine-McCluskey Method of Logic Minimization*, 2014, dostupný na *https://arxiv.org/pdf/1404.3349*. Lze najít i Open Source kódy, třeba *Github*.

[12] Viz https://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer Jeho zdrojové kódy se uvádějí i na Github.

[13] J. Hlavicka and P. Fiser, "BOOM-a heuristic Boolean minimizer," *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, 2001, pp. 439-442, doi: 10.1109/ICCAD.2001.968667

[14] Specifikace NP-úplného problému se zmiňuje třeba na wiki: https://en.wikipedia.org/wiki/NP-completeness

4 Realizace logických hradel

Zatím jsme předpokládali, že disponujeme ideálními logickými hradly. Ta skutečná se budují rozličnými technologiemi, třeba i pneumatickými systémy nebo s využitím relé, ale jde spíš o výjimky určené do náročného prostředí, v němž nejde vyloučit silné elektromagnetické rušení, např. některé průmyslové výroby. Logická hradla se jinak tvoří transistory.

Ojediněle se v některých částech obvodů dodnes používají bipolárními transistory, pak se mluví o TTL (*Transistor-Transistor-Logic*) či LVTTL (*Low voltage TTL*). Například FPGA řada Cyclone II a IV tvoří vnější vstupy a výstupy s LVTTL kvůli její vyšší odolnosti proti elektrostatickému průrazu a úrovni napájení. Jedná se však o méně běžné řešení.

V drtivé většině případů se logika realizuje unipolárními transistory CMOS, *Complementary Metal–Oxide–Semiconductor*, výslovnost "sí-mos". Budeme-li pouzí jejich uživatelé, pak potřebujeme znát leda některé jejich vlastnosti. Na prvním místě zde stojí časové zpoždění, které je kritickým parametrem ve většině návrhů. A hodí se zvažovat i složitost zapojení.

Zběžně tedy nahlédneme do nitra CMOS hradel, která se budují na bázi polovodičů. Ty čtenáři nejspíš už znají, ale snad neuškodí, když připomeneme některé jejich vlastnosti důležité pro vysvětlení CMOS transistory.

4.1 Připomenutí vlastností polovodičů

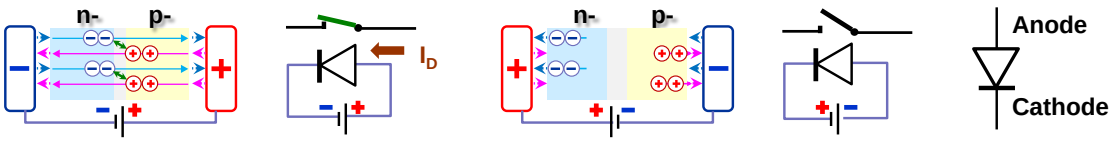
Základem polovodičů bývají prvky, které mají čtyři valenční elektrony, dnes často křemík (Silicon Si), ale i GaAs (Arsenid gallitý). Jejich normální nevodivost se změní přidáním stopového množství jiného látky, tzv. dopováním.

Chceme-li polovodič typu N, pak přimísíme prvek s pěti valenčními elektrony. Čtyři z nich se naváží na křemík, ale pátý zůstane volným elektronem. Ten nese záporný náboj a může se podílet na vedení elektrického proudu.

Polovodič typu P vznikne přidáním prvku se třemi valenčními elektrony, ty se navážou na atom polovodiče, ale na jeho čtvrté pozici bude jeden elektron chybět, čímž vznikne díra nesoucí kladný náboj. A opět se může podílet na vedení elektrického proudu.

Rozdílná síla dopování, tedy podíl příměsi, se zvýrazňuje znaménkem +, je-li silnější, a slabší zas -. Například n+ udává polovodič s větším podílem příměsi než N, zatímco p- polovodič ji má v sobě méně než P.

Pro CMOS bude důležité, že volné elektrony v N a díry v P polovodičích jsou jejich majoritními nosiči, ale díky nečistotám v nich existují i **minoritní opačné nosiče**.



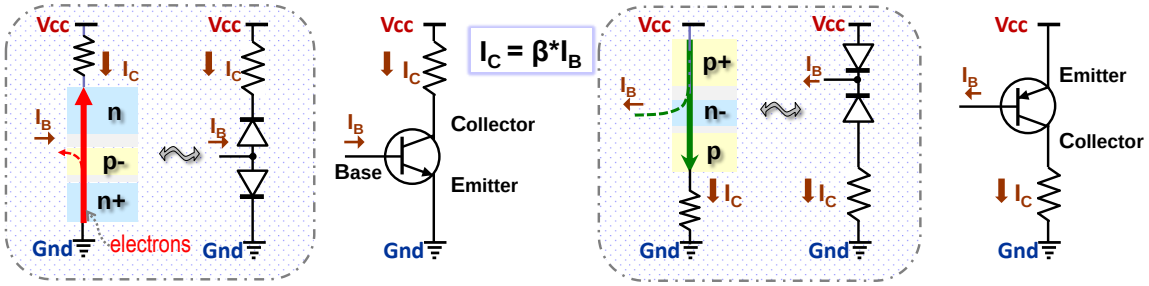
Obrázek 47 - Princip diody

PN dioda vznikne spojením P a N polovodičů se slabým dopováním. Mezi nimi existuje tenký přechod, depletiční zóna (vyprázdněná), angl. *depletion*, v níž nejsou ani volné elektrony ani díry. Zhruba řečeno se v ní navzájem vyrušily, takže zóna od sebe izoluje P a N polovodič.

Bude-li kladný pól připojený na P polovodič, anodu diody, pak odpuzuje díry a přitahuje k sobě volné elektrony. Záporné napětí, které je na N polovodiči, katodě diody, zase odpuzuje

volné elektrony a přitahuje díry. Zvýší se koncentrace nosičů poblíž oblasti přechodu mezi polovodiči, depletiční zóna se ztenčí, díry a volné elektrony se vyměňují a pohybují se. Dioda vede. Když se napětí na diodu připojí obráceně, pak jsou díry přitahované k zápornému pólu a elektrony ke kladnému. Nevodivá depletiční zóna se rozšíří a dioda nevede.

Bipolární transistor se vytvoří vhodným spojením polovodičů NPN či PNP a můžeme ho zhruba aproximovat dvojicí diod, které interně sdílejí své anody či katody. V klidu nevede.



Obrázek 48 - Princip bipolárního transistoru

Pokud však elektrický proud protéká mezi jeho bází (B) a emitorem (E), pak u NPN transistoru se ochuzená báze přeplní volnými elektrony, které proudí ze silně dopovaného emitoru[15]. Jakmile se oslabí depletiční bariéry mezi polovodiči, elektrony pronikají z emitoru přímo do kolektoru. U PNP transistoru jde naopak o díry, a ty se pohybují ve směru proudu.

Z pohybu majoritních nosičů pochází i název vývodů transistoru. Emitor vysílá elektrony, které sbírá kolektor. Všimněte si, že NPN transistor má svůj kolektor, při svém běžném zapojení, orientovaný k Vcc, zatímco PNP transistor tam má svůj emitor.

4.2 Princip CMOS

Existují dva jejich základní typy, a to NMOS (*N-channel MOSFET*) a PMOS (*P-channel MOSFET*). **Oba využívají vodivý kanál na bázi minoritních nosičů!** Polovodič typu P, který je substrátem NMOS, má díry jako své majoritními nosiče, ale vodivý kanál se v něm vytvoří pod elektrodou G, k níž se napětím přitáhnou minoritní nosiče, jimiž jsou elektrony. PMOS má substrát typu N a v něm jsou majoritními nosiči elektrony a díry minoritními.

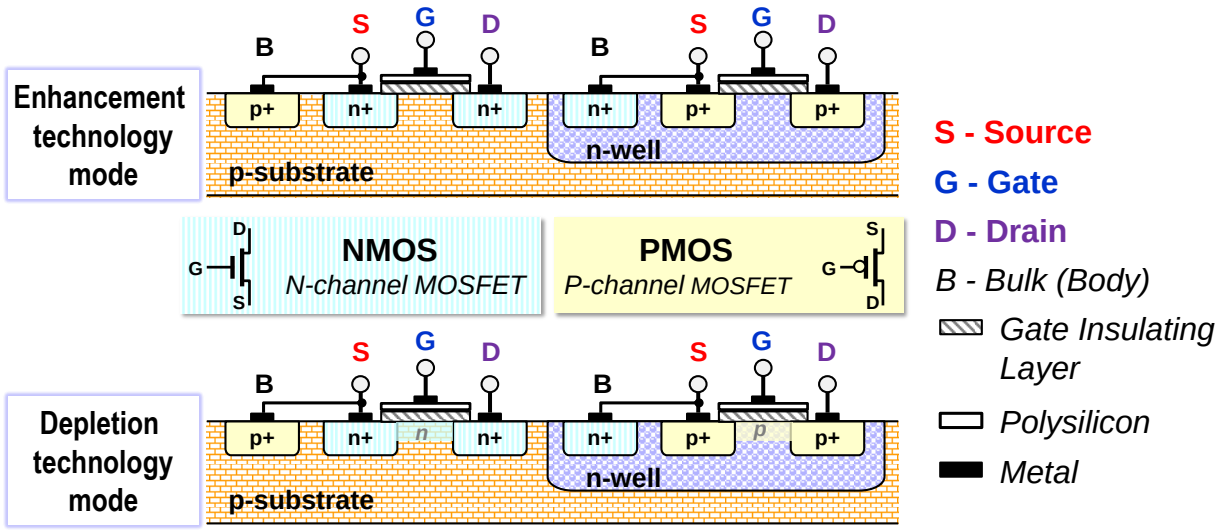
Existuje řada CMOS technologií rozlišených dle geometrie transistorů a aplikovaných příměsí. Jejich popis a rozbor jevů uvnitř nich leží mimo zaměření naší učebnice, a tak zmíníme pouze základní fakta. S vyšším dotováním klesá jednak odpor polovodiče a jednak pohyblivost nosičů v něm, a tak se CMOS vytvářejí na substrátech, které mají velmi slabé dotovaní, tedy velký odpor a značnou pohyblivost svých majoritních i minoritních nosičů.

Všechny technologické módy lze v základu rozdělit na *enhancement* a *depletion*[16], které se od sebe liší jen tím, že při *depletion* se navíc vytvoří částečně vodivý kanál mezi elektrodami S a D, a to pomocí slabé příměsi typu N, respektive P. Transistor je tedy ve výchozím stavu trochu **vodivý**, nikoli naplno, ale jen napůl. Technologie *enhancement* vodivý kanál nevytváří, takže CMOS je v klidu **nevodivý**. Původ názvů vysvětlíme v dalších odstavcích.

[15] Připomínáme, že elektrony se pohybují proti směru pomyslného elektrického proudu. V roce 1752, dlouho před jejich objevem, Ben Franklin zvolil opačný směr. Ten se nechal kvůli řadě již existujících pouček.

[16] Někde se uvádí české termíny *enhancement* = indukované a *depletion* = vestavěné. Oba české termíny nejsou ani přesné, viz dále, ani ustálené. Mají i další velmi odlišné významy, a tak se raději zůstalo u anglických pojmů.

Názvy elektrod CMOS transistorů vycházejí z pohybů nosičů. **Elektroda *Source*, S**, bude jejich zdrojem podobně jako emitor bipolárních transistorů, zatímco ***Drain*, D**, bude jejich příjemce, tedy analogicky ke kolektoru. Jelikož v NMOS jsou ve vodivém kanálu nosiči záporné volné elektrony, elektroda S potřebuje nižší napětí než D, zatímco PMOS chce na S vyšší napětí než na D, neboť nosiči v jeho kanálu jsou kladné díry.



Obrázek 49 - Základní technologie CMOS

CMOS elektroda **G, *Gate***, ovlivňuje vodivost mezi S a D elektrickým polem. Minoritní nosiče se přitahují elektrostatickou silou pod G. U *enhancement* technologie se pod G už vytvoří vodivý kanál, když napětí na ni bude vyšší než prahové napětí (*threshold voltage*). Jeho vodivost pak nelineárně závisí na napětí. Posiluje se jím, od toho vzniklo **pojmenování** technologie.

Pomocná elektroda **B (*Body*)** je vnějším vývodem leda u CMOS transistorů vyráběných jako samostatné diskrétní součástky. Uvnitř integrovaných obvodů se interně spojuje s elektrodou S (*Source*). Její existence vytváří v substrátu napěťové podmínky k sepnutí až po překročení prahového napětí na G (*Gate*).

První CMOS, objevený v roce 1959, neměl B a choval se jako napětím řízený odpor v celém svém rozsahu. Přidání B zlepšilo jeho vlastnosti coby spínače.

Technologie *depletion* se v roce 1970 zavedla jako vylepšení *enhancement*. Její transistory částečně vedou při 0 V na G. Napětím na ni se vodivost jejich kanálu buď posiluje, jako jako u *enhancement* CMOS. Přibyl však i ***depletion*** mód, kdy opačné napětí na elektrodě G oslabuje výchozí vodivost kanálu a rozšiřuje nevodivou depletiční oblast, odtud **název** technologie. Potřebuje však dvojí napájení, kladné a záporné.

Depletion CMOS vykazují nulový zbytkový proud mezi S a D díky dokonalému zavření a vyšší odolnost proti elektrostatickému průrazu.

V dnešních integrovaných obvodech se však upřednostňují *enhancement* CMOS, neboť spínají rychleji a stačí jim jen kladné napájení. *Depletion* CMOS se pořád vytvářejí a využívá se jako jejich částečné vodivosti při 0 V. Tvoří se jimi i analogově orientované části obvodů, zejména odpory, respektive napětím řízené odpory. Hodí se i na zdroje proudu.

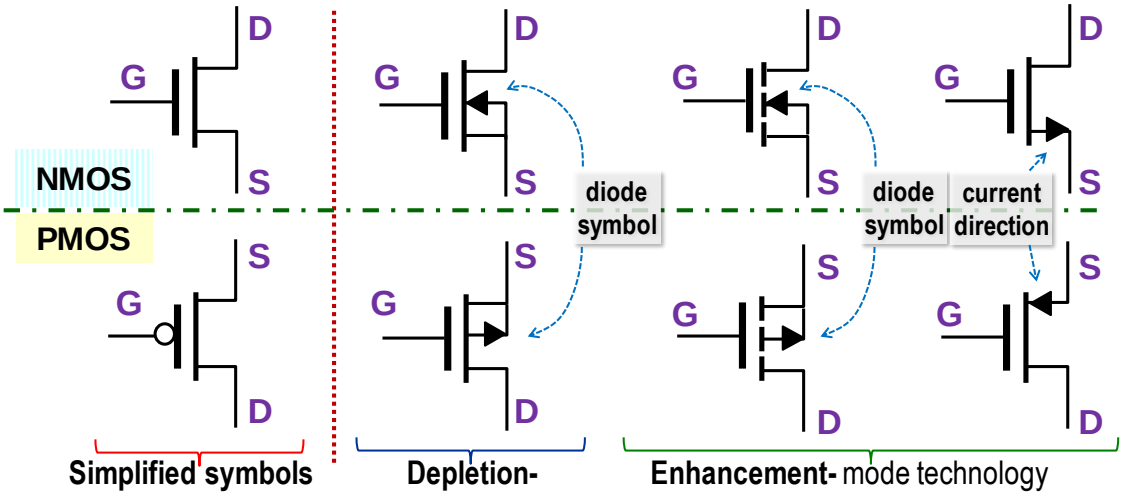
Existuje několik důvodů pro upřednostnění *enhancement* CMOS v hradlech, a to nejen jejich rychlejší klopení. V nanometrových technologiích pod 180 nm klesl zbytkový proud mezi S a

D na zanedbatelnou hodnotu oproti jiným parazitním CMOS jevům, jako jsou kvantové tunelovací efekty v polovodičích.

Depletion CMOS tak přišly o svoji hlavní přednost. V logických hradlech se navíc už nehodí jejich vodivost při 0 V na G, a to jak u NMOS, tak u PMOS, což může nastat i nechtěně a dlouhodobě při náhodném výpadku záporného napájení či oslabení signálu. Obvod se pak přehřeje a zničí. U *enhancement* tohle nehrozí.

4.2.1 Značky CMOS

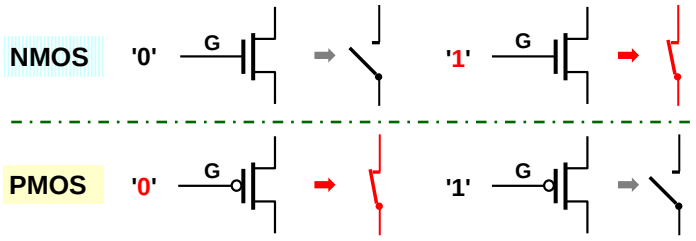
Následující obrázek uvádí nalevo obecné značky a za nimi symboly se specifikacemi technologického módu. Trojúhelníky v symbolech uprostřed neznamenají šipky, ale **diody**. Specifikují dle svého otočení vůči G buď NP přechod, tedy NMOS, či PN u PMOS.



Obrázek 50 - Přehled značek CMOS transistorů

Naproti tomu symbol vpravo se třemi vývody, doporučený normou IEEE, předpokládá již interní spojení mezi S a B elektrodami. U něho šipka specifikuje směr pohybu elektrického proudu, tedy analogicky ke značkám bipolárních transistorů NPN a PNP.

V dalším textu budeme používat pouze zjednodušené symboly, v nichž bublinka na vstupu PMOS indikuje jeho opačné chování oproti NMOS.



Obrázek 51 - CMOS transistory jako spínače

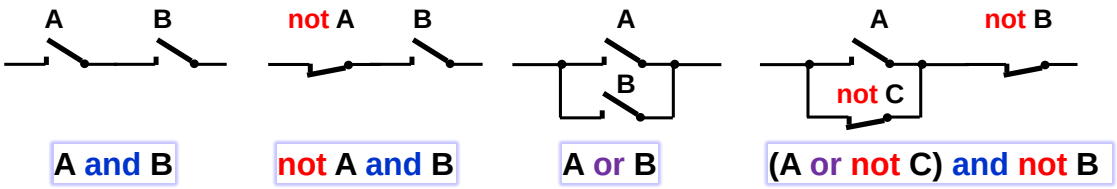
- **NMOS spíná na logickou '1'.** Jeho prefix N naznačuje, že v něm napětím na G vzniká polovodičový kanál na bázi volných elektronů (záporných), a ty, zhruba řečeno, přitáhne kladné napětí na jeho řídicí elektrodě (*gate*) a kanál se stane vodivým.
- **PMOS naopak rozepíná na logickou '1'**, což udává i bublinka invertoru před jeho řídícím vstupem G. Prefix P napovídá, že se v něm napětím na elektrodě G vytváří polovodičový kanál z pozitivních děr. Kladné napětí '1' na G je odpuzuje a kanál se uzavře.

- **Napětí ovládá vodivost CMOS transistorů**. Jejich sepnutí a rozepnutí se řídí elektrickým polem. U ideálního CMOS neteče proud do jeho řídicí elektrody G (*gate)*, ale u reálného ano, a nepříjemně roste s klesajícími nanometry technologie.
- **Řídicí elektroda G se musí vždy zapojit!** Pokud není *G* nikam připojena, je plovoucím vstupem, angl. *floating input*, a zvyšuje se riziko poničení obvodu.

Obecně platí, že nelze "NIC" pokládat za ekvivalent napětí o hodnotě 0 V, respektive logické '0', a samozřejmě ani '1'. Logická '0' i '1' jsou tvrdé zdroje napětí, angl. *stiff voltage sources*. Mají malý vnitřní odpor, a tak jen nepatrně změní svoje napětí se zatížením. Jakýkoli nezapojený vstup má naproti tomu vysoký vstupní odpor. Lehce se na něm indukují rušivé špičky. Dochází pak k náhodnému klopení hradla, což nebezpečně zvyšuje jak jeho odběr ze zdroje, tak i rušivé špičky. Důvod bude probraný na str. 66.

Poznámka: Některé publikace uvádějí, že nezapojený vstup logického hradla se chová jako '1'. Platí to jen u bipolární technologie TTL, ale i u té se radilo zapojit všechny vstupy.

Spínače dovolují realizovat podmnožinu logických funkcí, rozhodně ne všechny, ale lze jimi určitě vyjádřit implikanty. Sériové spojení přepínačů popisuje operaci AND a paralelní OR, jak ukazuje následující obrázek dole, který vychází z představy, že vstupní proměnná o hodnotě logické '0' nemačká na spínač, zatímco při '1' ho stiskne[17]. Negace proměnných se tedy reprezentují použitím spínacího, či rozpínacího tlačítka:

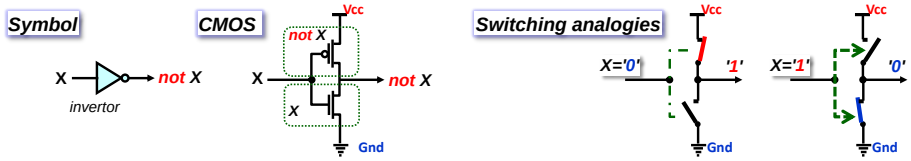


Obrázek 52 - Logika pomocí přepínačů

Musíme však zajistit, aby výstup hradla zůstal pokaždé připojený na napětí, tedy buď na '0' nebo na '1', neboť se povede na vstupy navazujících CMOS, a ty musíme držet v definovaných úrovních. Použijeme tedy dvě skupiny spínačů, čímž akcelerujeme i proces spínání. Horní skupina, při splnění své podmínky, připojuje výstup na Vcc, tedy na '1', a sestaví se podle požadované logické funkce. Dolní je pak její negací.

4.3 Invertor a *buffer*

Invertor má v horní skupině not X, tedy PMOS, ve spodní jeho negaci X, tedy NMOS, jímž se výstup připojí ke k '0' na Gnd při nesplnění horní podmínky. Obrázek ukazuje nejen CMOS realizaci invertoru, ale i její přepínačovou analogii při vstupu X v '0' a '1'.

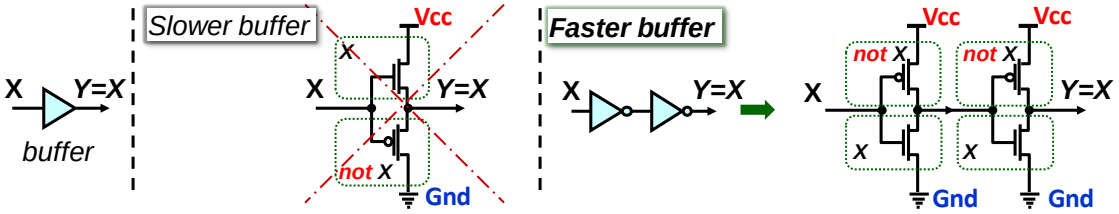


Obrázek 53 - CMOS invertor

Opakem invertoru je hradlo *buffer*, které kopíruje vstup na výstup, třeba kvůli oddělení a proudovému posílení či zvýšení časové zpoždění logické cesty. Při jeho tvorbě už narazíme

[17] Přepínačová analogie se převzala z žebříčkových diagramů, ladder logic, grafického jazyka dnešních průmyslových logických automatů PLCs, *Programmable logic controllers*.

na taje CMOS transistorů. Přímé zapojení totiž nefunguje dobře. Rychlejší variantou jsou dva invertory za sebou, umístěné těsně vedle sebe, tedy spojené vodičem zanedbatelné délky. Paradoxně signál jimi projde dříve než přímo zapojeným hradlem *buffer*!



Obrázek 54 - CMOS buffer

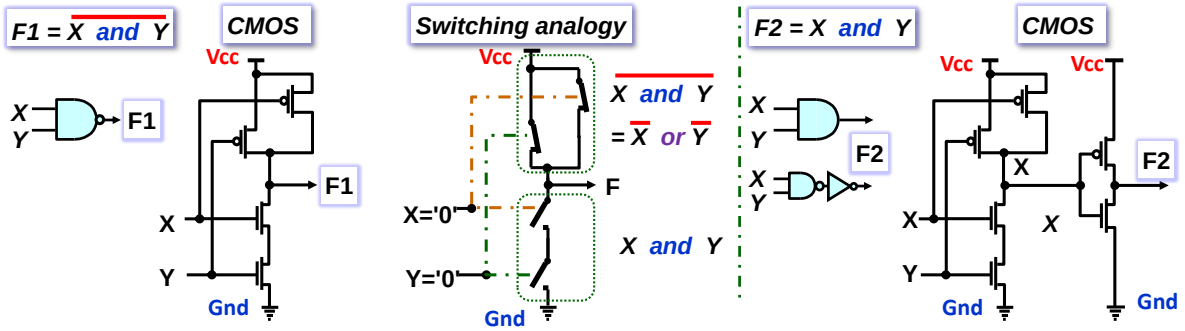
Proč ale? Důvody vyžadují hlubší porozumění fyzikálním charakteristikám CMOS transistorů, což přenecháme jiným publikacím. Pouze lehce nastíníme hlavní důvody.

- Díry v polovodičích zachovávají směr tečení pomyslného proudu, ale elektrony se pohybují proti němu, od záporného pólu ke kladnému. NMOS typy s vodivým kanálem na bázi elektronů mají tak lepší pracovní podmínky v dolní skupině. PMOS s vodivým kanálem z děr, zase v horní. **NMOS je třeba používat jen v dolní skupině transistorů a PMOS naopak jen v horní.**
- Analogová technika zná i zapojení blízká stylu hradla *buffer* uvedenému na obrázku vlevo. Výstup u nich roste/klesá se vstupním napětím a jejich zisk (zesílení) se drží slabě pod 1. Analogovým úrovním signálů to vyhovuje, pohybují se kolem středu napájecího napětí, ale logika potřebuje co nejrychlejší přeběhy ke svým krajním stavům Vcc a Gnd.
- Invertované verze hradel mění napětí svých výstupů v protisměru vůči vstupu, který vyvolal jejich překlopení. Horní či dolní skupiny CMOS se v invertované verzi příznivě ovlivňují. Změna napětí, pokles či nárůst, na jedné z nich ovlivní protilehlou skupinu směrem k urychlení děje, tedy k akceleraci překlopení kladnou zpětnou vazbou.
- Každý invertor má zisk zhruba deset i více během svého přeběhu, měřeno analogovýma očima. Dva invertory za sebou tak nejen oddělí vstup od zátěží za výstupem, ale navíc zlepší strmost hran signálu.

4.4 Logická hradla AND, NAND, OR a NOR

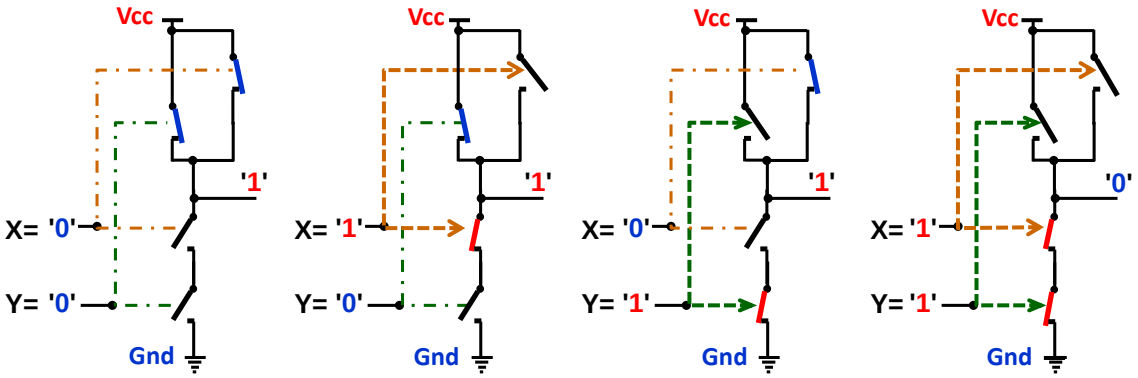
NAND hradlo má logickou funkci not (X and Y) = not X or not Y po rozkladu De Morganovým teorémem. Ta bude v horní skupině a do dolní zapojíme její negaci X and Y.

Hradla AND a OR se na úrovni CMOS tvoří častěji z NAND a NOR, za něž se přidají invertory z důvodu, který se uvedl v předchozí kapitole. Výsledek bude rychlejší než přímé vytvoření AND a OR nedoporučovaným prohozením horní a dolní skupiny CMOS transistorů.



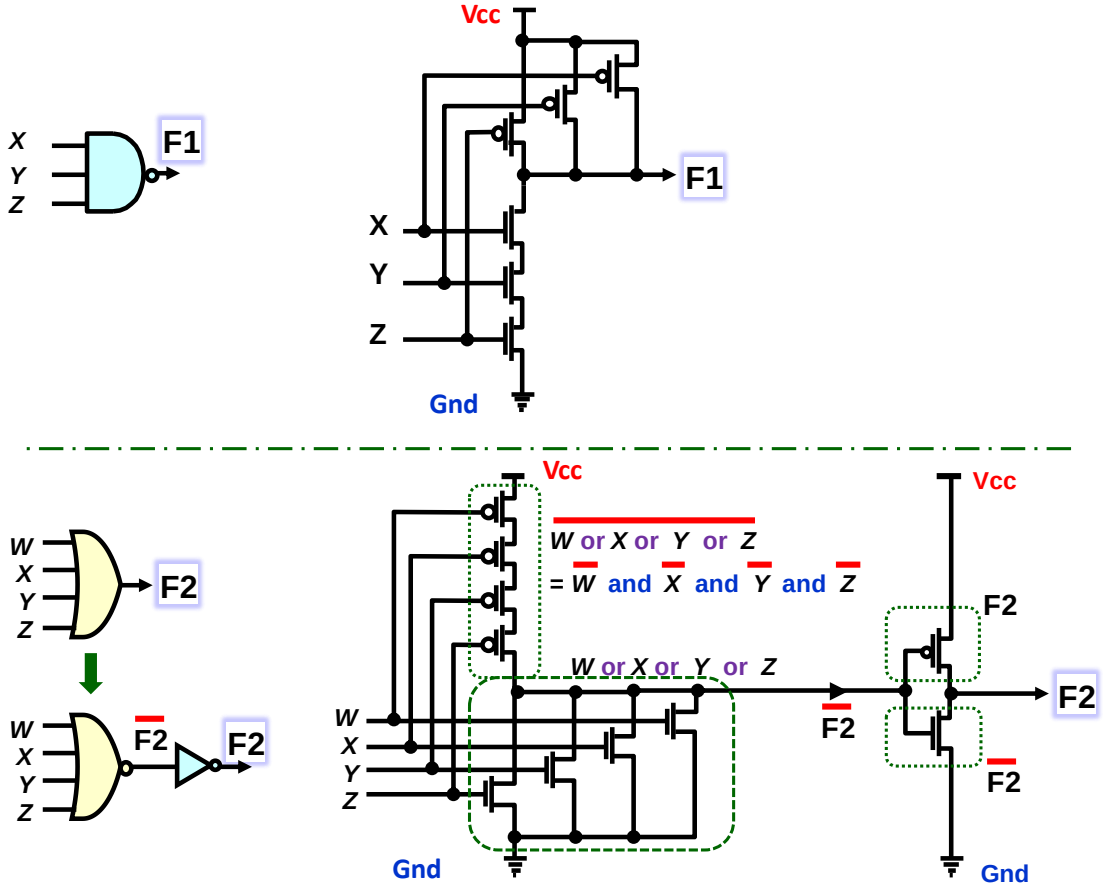
Obrázek 55 - Zapojení hradla NAND a AND

Funkci hradla NAND ukazuje obrázek dole:



Obrázek 56 - Spínačové analogie hradla NAND

Vícevstupová hradla se tvoří dalšími páry CMOS transistorů v horní a dolní skupině.



Obrázek 57 - Vícevstupová hradla NAND a OR

- Hradla NOT, NAND a NOR potřebují dvojici CMOS transistory na každý svůj vstup.
- Hradla AND, OR a BUFFER přidávají ještě výstupní invertor s dvojicí CMOS, pokud se vytvoří ze svých invertovaných protějšků.
- **Vícevstupová hradla budou pomalejší**. U nich se buď v jejich horní, nebo v dolní skupině vyskytuje více transistorů v sérii, a to v počtu vstupů. Zhorší se tím schopnost hradla budit výstup v '1' nebo '0'. Důvod ukážeme v kapitole 4.8.3 na str. 69.

4.4.1 Hradlo AND-OR

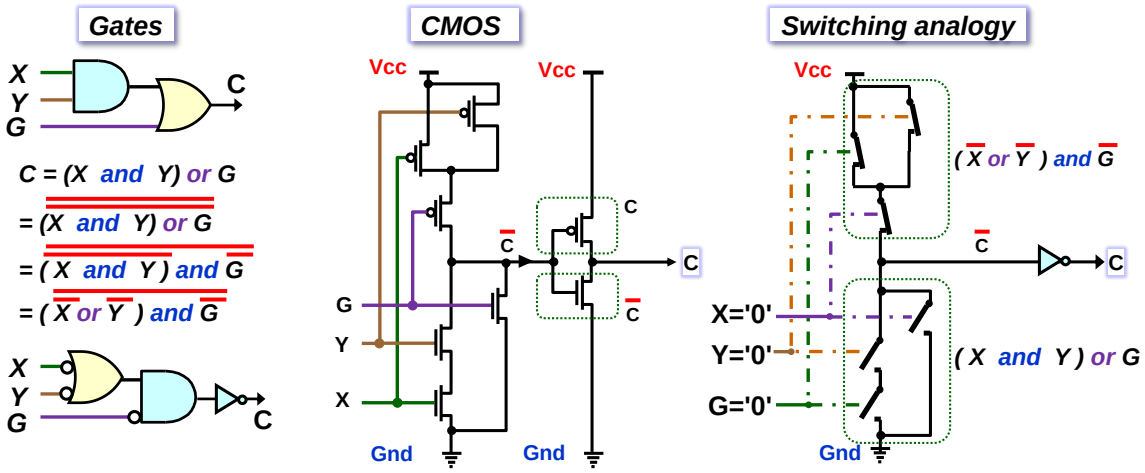
Nemusíme se omezovat jen na základní logické funkce, ale můžeme i složitější výrazy zapojit jako jediné hradlo. Na ukázku si sestavíme AND-OR hradlo, které se nám později hodí u sčítaček k rychlému šíření jejich přenosů.

Jeho rovnici napřed převedeme De Morganovým teorémem na negovanou funkci, které má v horní CMOS skupině jen PMOS a v dolní zase jen NMOS.

C = (X and Y) or G = not not ((X and Y) or G) = not (not (X and Y) and not G)

= not((not X or not Y) and not G)

Horní skupina bude: (not X or not Y) and not G a dolní její negací: (X and Y) or G

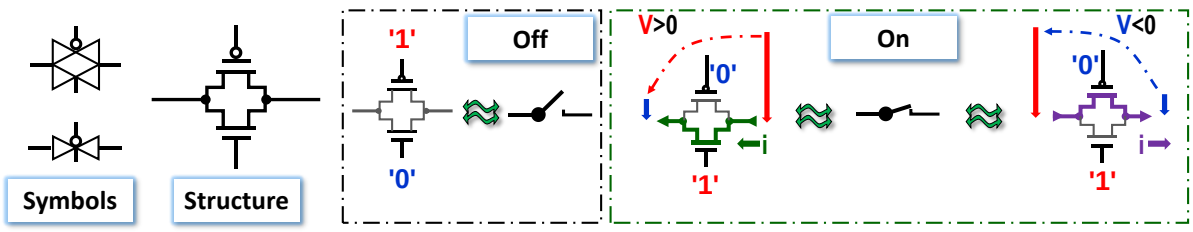


Obrázek 58 - Hradlo AND-OR

4.5 Transmission gate

Termín *transmission gate* by se dal přeložit jako přenosové hradlo, respektive průchozí hradlo, ale dostupné texty naznačují, že se jeho název nechává většinou bez překladu. Má podobnou strukturu jako *pass transistor logic*, PTL, což se někde považuje za jeho synonymum. PTL se však v řadě publikací více pojí ke spínáním analogových signálů. *Transmission gate* naznačuje, že se jeho realizace optimalizovala k přenosu úrovní logické '0' a '1'.

Jde o důležitý stavební prvek integrovaných obvodů, neboť funguje jako obousměrný spínač.



Obrázek 59 - *Transmission gate* respektive *PTL*

CMOS transistory vedou sice o obou směrech, ale jen v jednom dokonale, zatímco v opačném se uzavírají poklesem svého prahového napětí na elektrodě G. Spojí-li se paralelně opačné typy, pak v deaktivovaném stavu jsou oba uzavřené a napodobují rozepnutý spínač. Při aktivaci se uplatňují podle napětí mezi elektrodou G a svým vodivým kanálem.

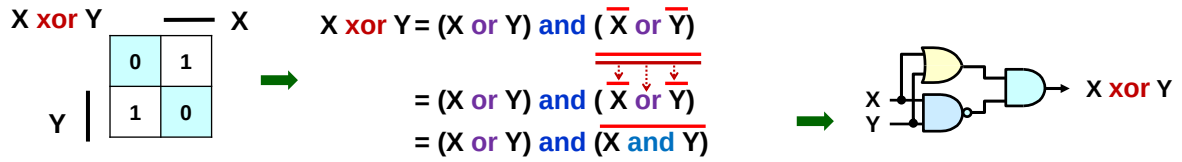
- Při kladném napětí protéká proud přes NMOS, jehož nosiči jsou záporné elektrony, které se pohybují proti směru našeho pomyslného elektrického proudu. Kladný pravý konec si je přitahuje od záporného levého pólu. PMOS se však uzavírá se zvyšujícím se napětím.
- Při záporném napětí mezi pravým a levým koncem převezme vedení proudu hlavně PMOS, v němž jsou nosiči kladné díry, ty proudí ve směru pomyslného proudu k zápornému pólu. NMOS se naopak uzavírá.

Prvek *transmission gate* má v sepnutém stavu odpor závislý na nanometrech své technologie, u malých i v řádu stovek ohmů. Na každém se pak ztratí trochu napětí. Nelze jich řadit mnoho za sebou. Používají se především k budování vnitřní struktury integrovaných obvodů, neboť v té se zná jejich přesné zatížení. Tvoří se s nimi nejen interní multiplexory, ale také synchronní obvody a konfigurovatelné propojky v FPGA.

4.6 Hradlo XOR

Hradlo XOR je složenou funkcí XOR(X,Y) = (X and not Y) or (not X and Y) vzniklou pokrytím logických '1' v její KM. Nehodí se k přímému zapojení v CMOS. Pozitivní a negované členy v jejím výrazu by se daly realizovat jen sériovým spojením NMOS a PMOS, což není přípustné.

Musí se tedy buď přidat invertory vstupů, nebo pokrýt funkci XOR logickými '0' a uplatnit De Morganovo pravidlo, aby hradla AND, OR a NAND vyšla lépe technologicky.



Obrázek 60 - Hradlo XOR metodou PoS

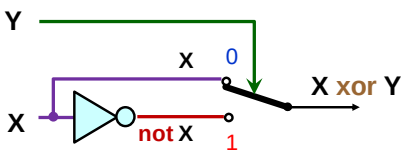
Ještě výhodnější realizaci vytvoříme, využijeme-li vlastnosti XOR coby řízené negace, viz kapitola 2.3.1 na str. 23. Rozklad v ní provedený jsme později zobecnili na Shannonovu expanzi, viz kapitola 3.4.3 na str. 49. Zvolíme třeba kofaktory podle Y:

XOR(X,'0') = (X and not '0') or (not X and '0') = X
XOR(X,'1') = (X and not '1') or (not X and '1') = not X

Samotné XOR lze tedy napsat:

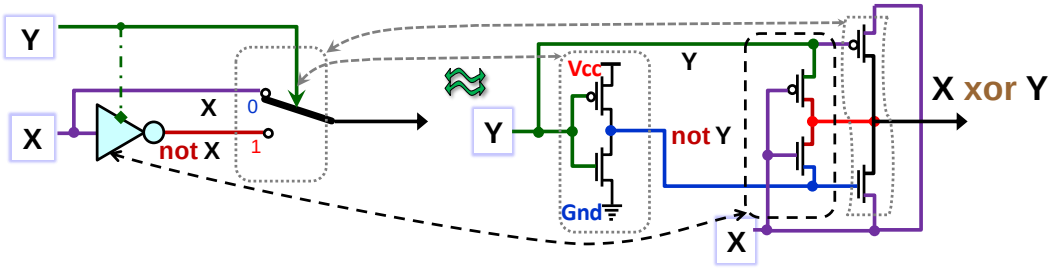
XOR(X,Y) = (not Y and XOR(X,'0')) or (Y and XOR(X,'1'))

Proměnná Y tak ovládá dvoupólový přepínač, který na výstup posílá buď X při vstupu Y='0', nebo not X, při Y='1'. Lze ho realizovat s použitím *transmission gates*.



Výstupní přepínač potřebuje invertor Y ke svému řízení, ale i dvě *transmission gate*. Pokud se využijí napěťové charakteristiky CMOS, lze jedno z nich vynechat a blokovat invertor X signá-

lem Y. Výsledné zapojení se realizuje něčím, což už nazveme obvodovou magií. Její CMOS zaklínadla☺ se vysvětlují v odborném článku autorů zapojení [18].



Obrázek 61 - XOR se 6 CMOS transistory

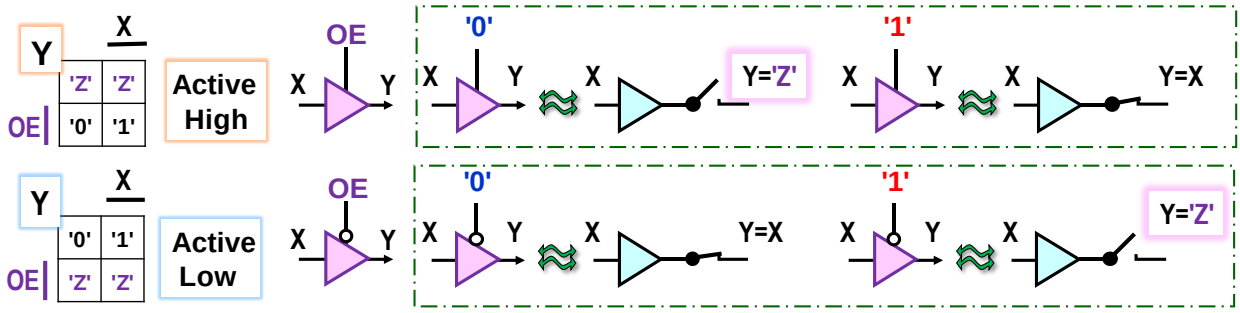
Existují i další šikovné triky, jimiž se XOR vytvoří jen se 4 CMOS transistory, tedy se stejnou složitostí jako AND hradlo, a dokonce se vymyslely i verze jen se 3 CMOS.[19]

Příklad sloužil především k demonstraci širokých možností technologie CMOS, ve které mnohé lze zapojit výhodněji, než vidíme na schématech.

4.7 Třístavové hradlo

Třístavové hradlo je stavební komponentou nejen FPGA obvodů, ale i jiných[20]. Jeho výstup lze uvést do stavu vysoké impedance, pro který se zavedlo označení 'Z' či hi-Z. Jde tak o další logickou hodnotu, která se může objevit na výstupu logického obvodu kromě logické '0' a '1'.

Uvedení do 'Z' řídí přídavný vstup, často označovaný jako OE, *output enable*. Je-li OE aktivní, pak se hradlo chová jako obyčejný *buffer*, v opačném případě přejde do 'Z'.



Obrázek 62 - Třístavový *buffer*

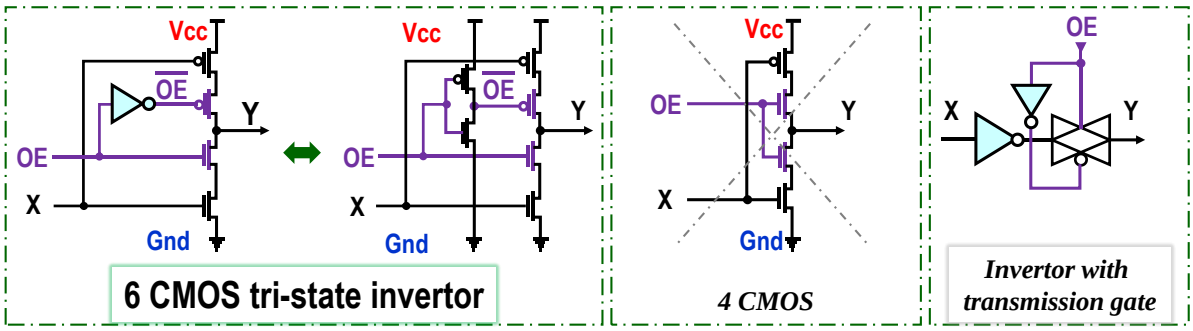
I další typy hradel lze samozřejmě rozšířit i o vstup OE, třeba vytvořit i třístavový invertor. Příklad použití třístavových hradel si ukážeme u logických elementů, kapitola 5.5.6 na str. 90.

[18] N. Ahmad and R. Hasan, "A new design of XOR-XNOR gates for low power application," *2011 International Conference on Electronic Devices, Systems and Applications (ICEDSA)*, 2011, pp. 45-49, doi: 10.1109/ICEDSA.2011.5959039.

[19] Různé způsoby realizace hradla XOR se rozebírají v článku Yann Guidon, Paris, France:
https://hackaday.io/project/8449-hackaday-ttlers/log/150147-bipolar-xor-gate-with-only-2-transistors/

[20] Třístavové hradlo se využívá na obousměrných paralelních počítačových sběrnicích, avšak od těch se dnes již ustupuje. Současné jednosměrné sériové linky sice přenášejí data po jednotlivých bitech, ale ve výsledku paradoxně mnohem rychleji, a to z řady důvodů. U sériové linky nás například nezdržuje časová synchronizace několika paralelně přenášených signálů a lépe se potlačí i vzájemné rušení, tzv. přeslechy, kdy se elektromagnetické pole vytvářené signálem indukuje do sousedních paralelních vodičů. Přenos po sériové lince se pak běžně urychluje tím, že se jich použije několik, přičemž každá z nich vede část dat nezávisle na jiných. Například video výstup Display Port přenáší obraz po čtyřech sériových linkách, další má pak na audio a pomocné informace..

Třístavový invertor se nejčastěji tvoří stylem vlevo, kdy se přidá odepnutí jako horní, tak dolní skupiny řízené invertorem.



Obrázek 63 - Příklady některých vnitřních struktur třístavového invertoru

Obrázek ve středu ukazuje nedovolené řešení. Kdyby se vnitřní invertor signálu OE nahradil užitím NMOS v horní skupině, ušetřily by se sice dva transistory, ale NMOS by se ocitl v horní skupině, v níž nemá vhodné podmínky ke své činnosti, a ještě v sérii s PMOS! Oba typy CMOS mají sice blízké parametry, ale nelze je vyrobit tak, aby byly shodné už kvůli tomu, že používají jiné nosiče. Díry mají v polovodičích zhruba třetinovou pohyblivost než elektrony.

Poslední varianta vpravo je sice přípustná, ale dala by obhájit jenom jako přídavek ke složitější logické funkci, ne u jednoduchého invertoru. Zhoršovala by kvalitu výstupu.

4.8 Model dynamického chování dvou invertorů

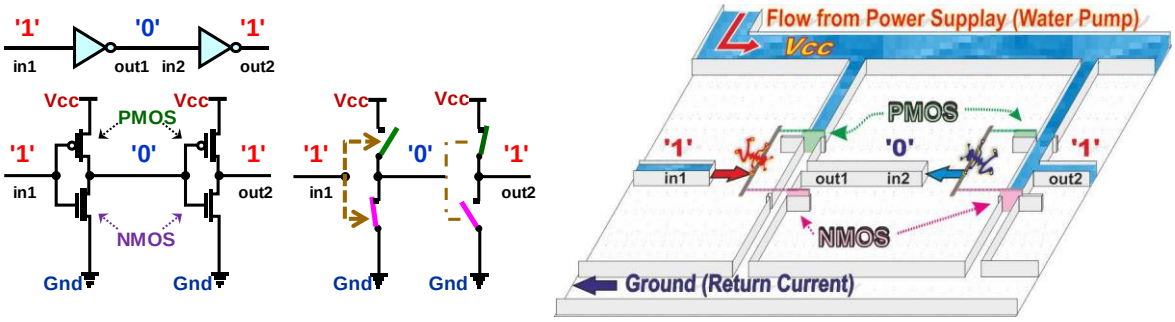
Kaskádu dvou CMOS invertorů si napřed namodelujeme jejich fyzikální analogií, v níž využijeme vodní kanály, neboť šíření elektrického signálu po vedení, angl. po *transmission line,* vykazuje jevy blízké vodě jako postupný nárůst napětí (hladiny) a vznik vln díky odrazům.

Skvělá animace dějů na vodiči se v době psaní této publikace nacházela na konci článku: https://practicalee.com/transmission-lines/ a ozřejmuje fakt, že voda dokáže napodobit jistou podmnožinu elektrických jevů.

4.8.1 Vodní model dvou invertorů

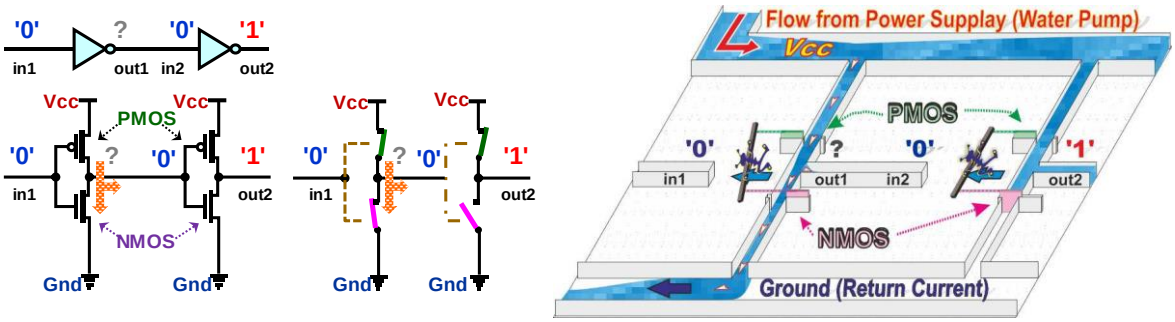
Oba transistory budeme aproximovat pouhými spínači, tedy jejich nejčastějším užitím v logice. Emulujeme je posuvnými vraty, viz obrázek dole. Rozepnutý přepínač odpovídá zasunutým vratů, voda je zastavená a neteče, zatímco sepnutý přepínač otevřeným vratům, tedy uvolněnému průtoku. Odpor transistoru v sepnutém stavu odpovídá průřezu kanálu.

Objem kanálu, který se musí naplnit, zastupuje kapacity vodičů. Máme-li dva invertory za sebou, pak jejich výchozí stav ukazuje obrázek dole.



Obrázek 64 - Vodní model - výchozí stav

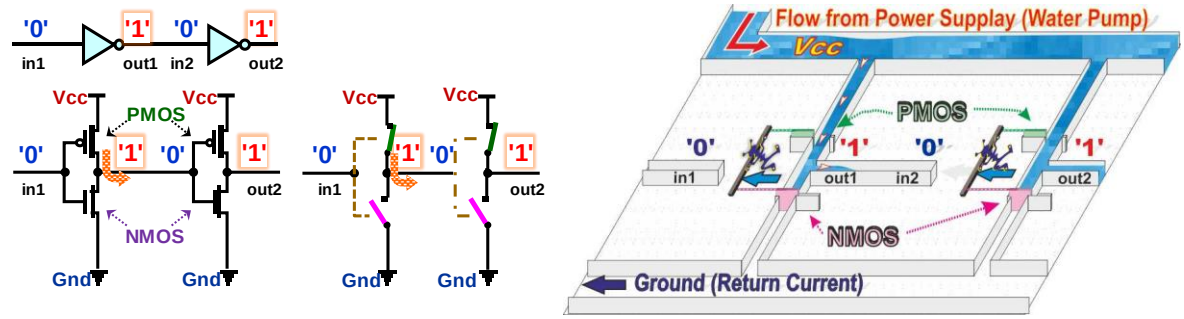
Elektrické pole tlačí na vrata při logické '1', zatímco v '0' je vytahuje. Zatlačená PMOS vrata blokují (rozpojí) kanál a vytažená ho uvolní (sepnou). NMOS vrata se zasouvají na druhou stranu, a tak pracují přesně opačně.



Obrázek 65 - Vodní model - dočasný stav zkratu

Došlo-li k poklesu v kanálu in1, a tak se mění stav levého invertoru, ale nikoli okamžitě. Proud vody urychlí otevření horních PMOS vrat a zpomalí zavírání dolních NMOS. Voda plní výstupní kanál out1, ale víc jí uniká zkratovým proudem. Obě CMOS vrata jsou nyní naplno otevřená (ve stavu své saturace). Ve Vcc přívodu se dočasně snižuje hladina napájení. Výstupní napětí out1 bude teď '?', tedy někde mezi '1' a '0'.

Zkratový proud se objevuje při každém přepínání výstupu hradla, ale u novějších obvodů trvá jen několik pikosekund. Na celkové spotřebě energie se podílí v jednotkách procent.

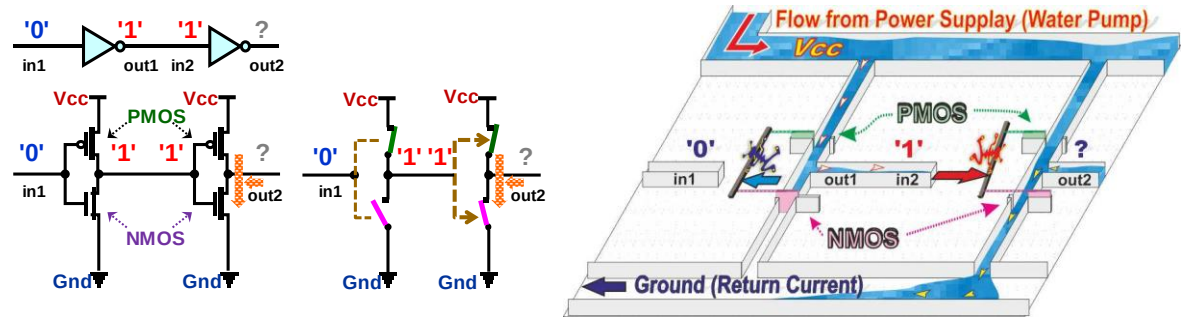


Obrázek 66 - Vodní model dvou invertorů - obě hradla v logické '1'

Dolní vrata se již zcela uzavřela a voda plní kanál od out1 k in2. Všimněte si, že v **logické '1' teče proud směrem ven z výstupu budícího hradla.**

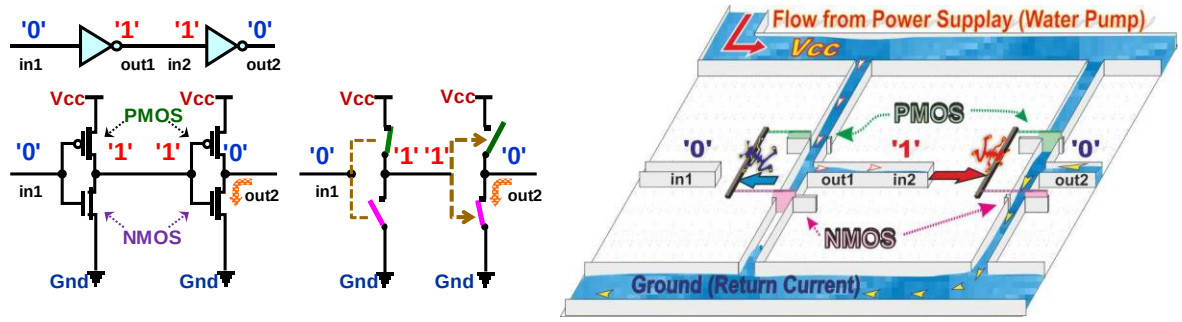
Pravé hradlo má na vstupu stále '0', dosud neví ještě o změně, protože se záplavová vlna teprve šíří kanálem od out1 k in2. Na jeho konci se hladina dosud nezvedla nad rozhodovací úroveň 50 %, takže pravý invertor zatím nepřepnul.

Máme další přechodný stav, kdy **oba invertory mají shodné své výstupy**. Vrátíme se k němu ještě v pozdějším výkladu metastability klopných obvodů.



Obrázek 67 - Vodní model dvou invertorů - pravé hradlo ve zkratu

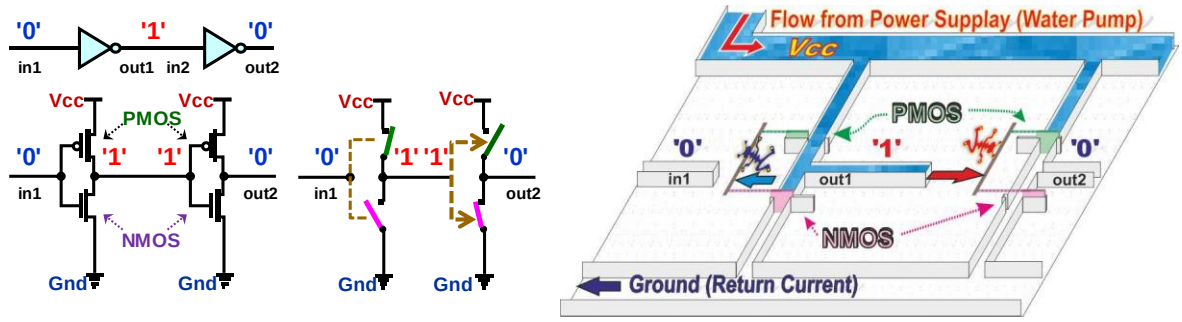
Kanál se již zaplnil a na jeho in2 konci se zvýšil potenciál na úroveň logické '1'. Dolní vrata pravého invertoru se otevřela tlakem vody, ale horní se ještě neuzavřela. Pravý invertor má oba transistory ve stavu saturace, kdy jimi po několik pikosekund protéká **zkratový proud**.



Obrázek 68 - Vodní model dvou invertorů - pravé hradlo přepnulo

U pravého invertoru se již uzavřela horní PMOS vrata a spodními NMOS vraty naplno vytéká voda. Máme již logickou '0' na out2 výstupu pravého hradla, která vybíjí jeho kapacity, změna se tak šíří po navazujícím vodiči. Vyprazdňuje se výstupní out2 kanál do odtoku Ground, ve kterém se hladina krátkodobě zvýší, než se odvede přívalová vlna.

Všimněte si, že **proud v logické '0' směřuje do výstupu budícího hradla.**



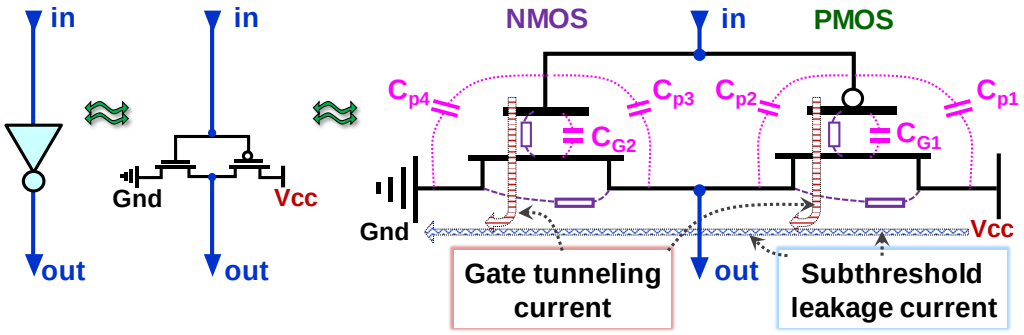
Obrázek 69 - Vodní model dvou invertorů - ustálený stav

Výstupní kanál se již vyprázdnil a nastal ustálený stav, v němž invertory setrvají až do další změny na vstupu in1.

Nepřesnost našeho vodního modelu spočívá především v ovlivnění přepínání pouhou výškou hladiny. Elektrické napětí je rozdílem potenciálů ve dvou bodech, a tak otevření či zavření vrat by správně mělo záviset na diferenci výšky hladiny ve vstupním kanálu vůči stavu v GND odtoku. Podobný model je sice technicky možný, využil by rozdíl tlaků, ale ztrácel by názornost, a tak jsme ho zredukovali a raději nechali nedostatek. Ze stejného důvodu se rovněž nesimulovala vazba mezi horními vraty a spodními vraty, která se v logických hradlech vyskytuje mezi PMOS transistory horní a NMOS dolní skupiny a akceleruje překlopení.

4.8.2 Statický odběr hradla

Vodní model demonstroval, že hradla si řeknou o nárazové odběry ze zdroje při svém klopení. V klidu odebírají jen parazitní zbytkové proudy. Ty vznikají tunelovými efekty v polovodičích, tzv. *quantum tunneling*, a to trvale bez ohledu na stav výstupů hradel.



Obrázek 70 - Parazitní kapacity a proudy v CMOS

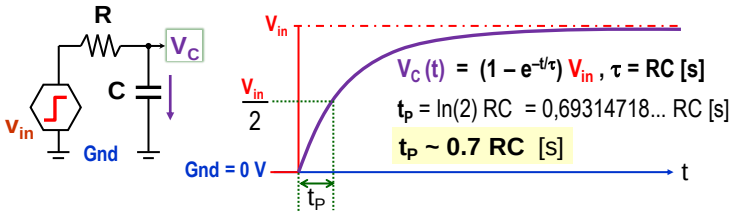
U rozměrnějších *enhancement* technologií nad 180 nm má největší podíl zbytkový proud mezi elektrodami S a D v uzavřeném stavu, angl. *subthreshold leakage current*, ale u menších je zanedbatelný vůči jiným jevům. Ve vodním modelu si ho lze představit jako netěsnosti vrat.

S klesajícím nm se ztenčuje izolační vrstva pod elektrodou G až na tloušťku několika atomů, a tak roste její kvantové tunelování, *gate tunneling current*, zhruba ve stylu prosakování vody z přívodního kanálu do výstupu. Vliv proudu do elektrody G se s poklesem na nm hradla zvyšuje na dominantní odběr. U 7 nm technologie se jeho podíl udává až k 80 % celkové spotřeby obvodu a mikroelektronici intenzivně hledají cesty, jak ho zredukovat.

CMOS transistory mají i parazitní kapacity mezi svými částmi, neboť ty jsou oddělené jen tenkými vrstvami. V našem vodním modelu se při spínání plnil nejen prostor mezi vraty, ale i následující kanál, což v CMOS odpovídá nabíjení parazitních kondenzátorů. Zpožďuje se tím průchod signálu. Jev probereme podrobněji na str. 74.

4.8.3 Odporový model dvou invertorů CMOS

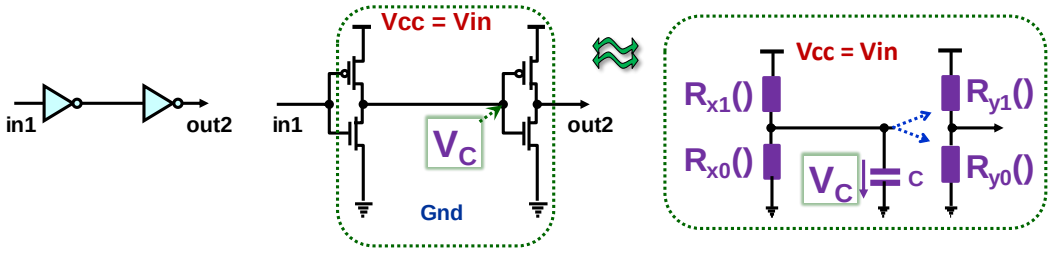
Přesnější rozbor dějů během spínání invertorů by vyžadoval už hlubší zanoření do struktury CMOS transistorů a přesáhl by rozsah naší učebnice. Spínání CMOS transistorů tak jen zhruba aproximujeme kondenzátory, které se nabíjejí přes odpory, tedy analogiemi RC článků, též známými pod názvem integrační články. Jejich chování lze popsat diferenciální rovnicí[21].
Zajímá-li nás doba, za kterou napětí kondenzátoru V_C naběhne na 50 % napětí Vin, pak jejím řešením dostaneme časovou konstantu tp = ln(2) RC [s], která se běžně aproximuje tp= 0.7 RC, neboť málokdy známe odpor i kapacitu s přesností lepší než 10 % až 20 %.



Obrázek 71 - RC článek

Pomocí RC článku namodelujeme zpoždění průchodu signálu skrz levý invertor. Kondenzátor C zahrnuje součet parazitních kapacit na výstupu levého invertoru, tak navazujícího vodiče a vstupu pravého invertoru. Jeho hodnotu můžeme v modelu pokládat za konstantní.

[21] Odvození rovnice najdete třeba zde: https://www.electronics-tutorials.ws/rc/time-constant.html



Obrázek 72 - Odporový model dvou invertorů

Velikosti odporů se však výrazně mění podle momentálních napětí mezi trojící elektrod CMOS transistorů. Při malých hodnotách je lze aproximovat odpory řízenými napětím, a to na něm lineárně i nelineárně závislé. Dochází i k saturaci, kdy se transistory projevují víc jako zdroje proudu. Variabilitu odporů $R_{x0}()$, $R_{y0}()$, $R_{x1}()$ a $R_{y1}()$ jsme naznačili závorkami.

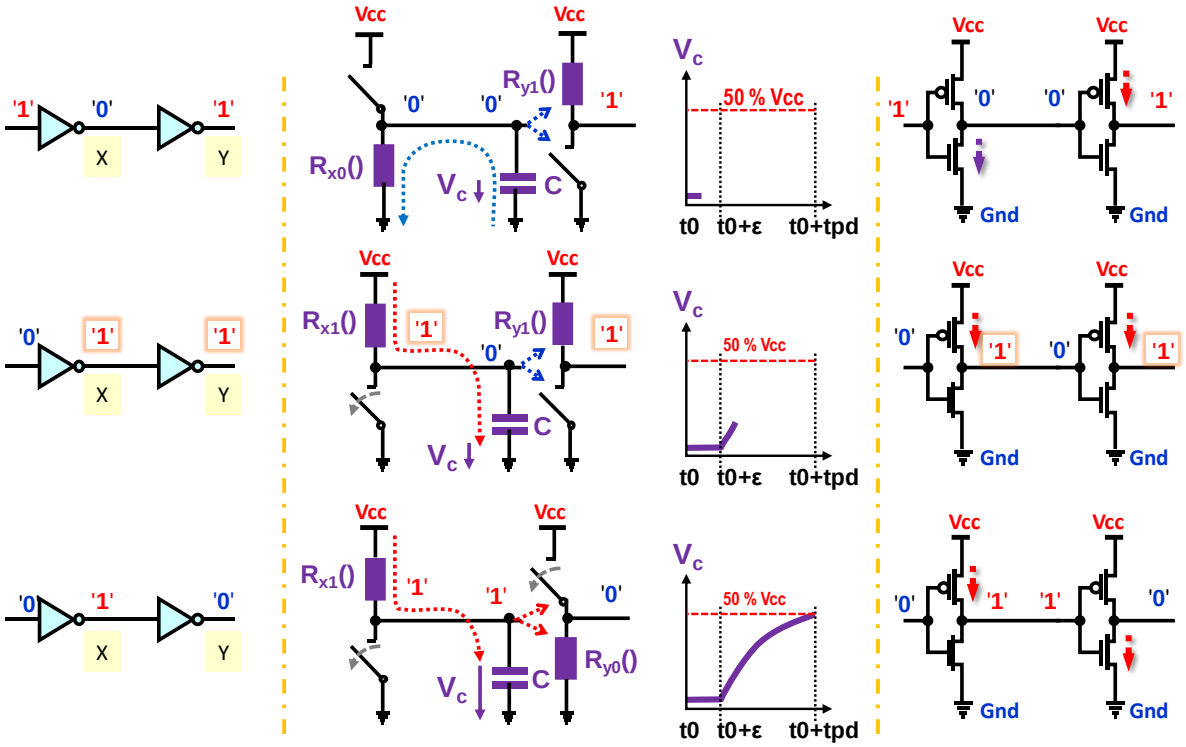
Jaké budou hodnoty časových konstant RC?

Transistory NMOS a PMOS, které se vyrábějí jako diskrétní součástky, mohou mít odpor v sepnutém stavu mít i pod 1 ohm. Hradla ale potřebují větší hodnoty kvůli zkratovým jevům při svém spínání, viz vodní model Obrázek 65 a Obrázek 67.

Vodivost CMOS závisí na řadě parametrů. Jedním z nich je i podíl šířky a délky vodivého kanálu pod elektrodou G. Volí se tak, aby transistory v sepnutém stavu protékal maximální proud odpovídající odporu řádu stovek ohmů, až kiloohmů. Rychlé varianty obvodů se navrhnou s nižšími odpory, tedy s vyššími maximální proudy, aby se kapacity nabíjely rychleji. Součástky určené do aplikací, v nichž se žádá nízký odběr, zase upřednostní vyšší náhradní odpory při sepnutí CMOS, tedy jejich menší zkratové proudy a nárazové odběry.

S poklesem rozměrů technologií CMOS se hlavně zmenšují plochy parazitních kondenzátorů, které zmiňoval Obrázek 70 na str. 69. Jejich kapacita klesá, což redukuje dobu k jejich nabití či vybití a spotřebu energie.

Následující obrázek ukazuje dění na výstupu levého invertoru, na pravém bude obdobné. Vynechaly se pikosekundové okamžiky zkratu, kdy vedou oba CMOS. Mají zanedbatelný vliv na zpoždění. Přepnutí pravého invertoru volíme při 50 % Vcc, což nastane za čas 0.7 RC [s].



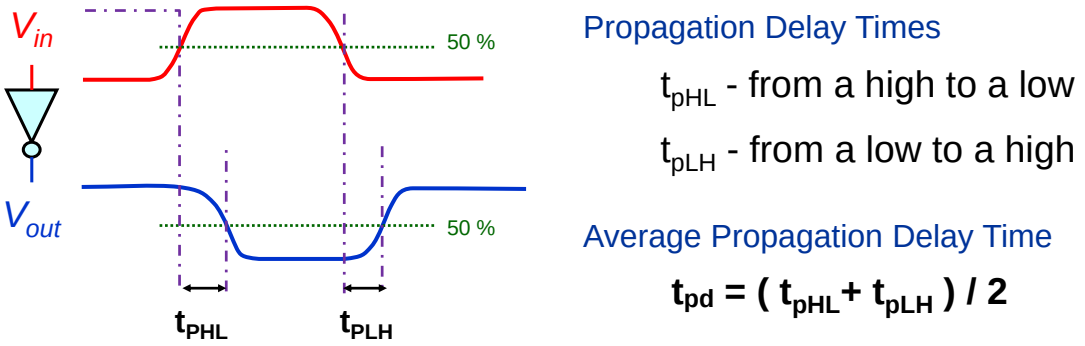
Obrázek 73 - Zpoždění na dvojici invertorů

- **čas t0** - Nechť ve výchozím stavu jsou oba invertory ustálené. Levý má '0' na svém výstupu X. Jeho horní PMOS je uzavřený a budeme ho zhruba pokládat za rozpojený ideální spínač. Sepnutý dolní NMOS nahradíme odporem $R_{X0}()$. Kondenzátor C se vybijí a jeho napětí V_C se asymptoticky blíží k jakési spodní hodnotě. **Proud nyní teče směrem do výstupu X levého invertoru**. Výstup Y pravého invertoru bude v '1'. Jeho sepnutý horní PMOS aproximujeme odporem $R_{Y1}()$ a dolní zavřený NMOS pak otevřeným spínačem.
- **čas t0+ε** - Levé hradlo přešlo z '0' do '1' a odpor $R_{X1}()$ sepnutého horního PMOS nabíjí kondenzátor C na napětí V_C, které je zatím pod rozhodovací úrovní 50 % Vcc. Nepřepnulo ještě pravý invertor, a tak **oba mají hodnoty '1'** na svých výstupech. **Proud u levého invertoru teče nyní směrem z jeho výstupu X.**
- **čas větší či roven t0+tpd**, kde tpd označuje zpoždění, *propagation delay*. Napětí V_C již přesáhlo rozhodovací úroveň 50 % Vcc. Pravý invertor se překlopil. Rozepnul se jeho horní PMOS a sepnul se spodní NMOS, který modelujeme odporem $R_{y0}()$.

O zpoždění tpd lze obecně říct, že u všech typů hradel:

- tpd lineárně závisí na časové konstantě RC článku;
- tpd klesá s růstem napájecího napětí, neboť proudy protékající CMOS se zvyšují s napětími mezi jejich elektrodami. Náhradní odpor sepnutého CMOS se tak zmenšuje;
- tpd se mění s teplotou, kde proti sobě působí několik různých faktorů. U malých technologií se s jejím růstem může i zkracovat a u rozměrnějších se často prodlužuje;
- tpd bývá různé při přepnutí do '1' či do '0'. Obě skupiny nejsou úplně symetrické už kvůli tomu, že do dolní skupiny se dávají NMOS transistory, v nichž se vytváří vodivý kanál na bázi volných elektronů. A ty mají třikrát vyšší pohyblivost než díry, z nichž se formuje vodivý kanál u PMOS. *Pozn. Na pohyblivosti závisí rychlost nosičů v polovodiči. Je-li vyšší, zlepší se proudové a frekvenční charakteristiky.*

V praxi se uvažuje jen průměrné zpoždění, v obrázku bylo označené jako tpd, *propagation delay time*. Katalogy ho udávají pro různé teploty uvnitř obvodu, pro vybrané hodnoty od 0 °C až do 125 °C, a také pro jednotlivá dovolená napájení Vcc.



Obrázek 74 - Zpoždění na invertoru

Pořád nám zůstává otázka, jaké bude zpoždění invertoru? Můžeme uvést akorát hrubé orientační údaje, jelikož zpoždění hodně závisí jak na použité technologii, tak geometrii CMOS transistorů, a není konstantní! Ovlivňuje ho teplota i fluktuace napájení. Vícevstupová hradla jsou obecně pomalejší než invertor, protože v nich bývá v sérii spojeno více CMOS, na nichž se rozloží napětí, na každém klesne, což sníží proud skrz ně. Kapacity se pak nabíjejí/vybíjejí pomaleji. Publikace[22] zmiňují zpoždění invertoru v desítkách pikosekund pro větší technologie (45 nm a více) a v jednotkách pro nižší, u 7 nm CMOS pak i 2.5 ps.

S poklesem nm sice roste hustota integrace, ale hradla se zrychlují již méně výrazně, a to kvůli dalším nepříznivým jevům. U 3 nm technologie se očekává jen nepatrně lepší hodnota[23], v nezatíženém stavu cca 2 ps.

Co omezuje pracovní frekvenci? V dostupných publikacích se liší teoretické odhady, na jaké maximální frekvenci mohou pracovat polovodičová hradla při vhodné volbě jejich materiálů. Nejčastěji se udávají hodnoty někde nad 100 GHz, ale existují i ojedinělé studie, které tvrdí, že logika by mohla pracovat dokonce na frekvencích přes 1 THz.

Procesor Intel Core i9-14900KS měl frekvenci přes 6 GHz a šlo o nejrychlejší běžně prodávaný typ v době psaní této učebnice (rok 2024). Většina procesorů zůstávala na taktech do 4 GHz.

Dílčí části obvodu mohou sice běžet i na několika desítkách GHz, třeba budiče sériových sběrnic, ale obvod brzdí jak teplotní problémy, tak napájení. Už na vodním modelu jsme viděli, že hradlo si při svém klopení buď řeklo o nárazový odběr ze zdroje, nebo vypustilo napěťovou vlnu na zemnicí spoj. A současné technologie nedovedou při vyšších frekvencích dodávat dost energie všem hradlům a odvádět proudové rázy ze zemnících spojů.

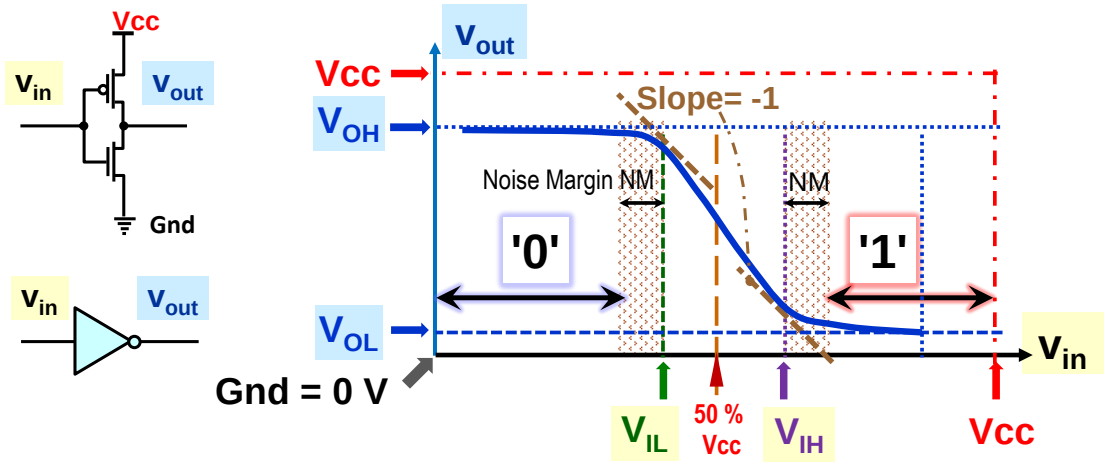
Paralelizace operací nabízí dnes mnohem dostupnější řešení k akceleraci výpočetního výkonu. Přidávají se procesorová jádra a používají se i akcelerátory vytvořené logickými obvody.

[22] Například: Aaron Stillmaker, Bevan Baas, Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm, Integration, Volume 58, 2017, Pages 74-81, link.

[23] Etienne Sicard, Lionel Trojman: Introducing 3-nm Nano-Sheet FET technology in Microwind.2021. hal-03377556

4.9 Zavedení logických '0' a '1'

V předchozím textu jsme kvůli zjednodušení předpokládali nejčastější situaci, a to pozitivní napěťovou logiku (realizaci logické '1' vyšším napětím a '0' nižším napětím). V časových charakteristikách CMOS invertoru jsme však viděli, že přechod od logické '1' k '0', a nazpět, neprobíhá okamžitě, ale napětí se postupně mění, jak se nabíjejí kapacity. Obrázek na další stránce ukazuje jeho výstupní křivku. Jeho výstup Vout nikdy nemá plné napětí Vcc či 0 V.



Obrázek 75 - Zavedení logické '0' a '1'

V průběhu V_{out} existují čtyři důležité hodnoty, které výrobci uvádějí ve svých katalozích.

- **V_{OL}** (*output low*) označuje výstupní napětí hradla při jeho stavu v logické '0'.
- **V_{IL}** (*input low*) specifikuje vstupní napětí, při němž se výstup začíná měnit. U invertoru má tečna průběhu výstupního napětí směrnici -1.
- **V_{IH}** (*input high*) je bod podobný V_{IL}, ale na horním konci průběhu V_{out}.
- **V_{OH}** (*output high*) udává změřené napětí výstupu v logické '1'.

Do průběhu vložíme námi požadovanou šumovou imunitu NM, *Noise Margin*. A v pozitivní napěťové logice prohlásíme za logickou '1' jakékoli hodnoty napětí vyšší V_{IH}+NM, horní rozhodovací úroveň, a za logickou '0' bereme cokoli nižšího než V_{IL}-NM, dolní rozhodovací úroveň. Dostali jsme napěťové rozsahy logické '0' a '1'. Vše mimo ně nazveme nedovolenou, či nechtěnou úrovní. Bude tam sice při každém klopení hradla, ale krátce.

Jak velké jsou rozsahu '0' a '1'? Závisí předně na údajích výrobců vztažených k napájecímu napětí a pak na námi zvolené šumové imunitě NM.

Uvedeme orientační příklad. Vezmeme ho z FPGA řady Cyclone IV E[24], které používáme v našich přípravcích, a uvedeme na nich užité hodnoty. Největší část obvodu, v níž se tvoří logika, označená *FPGA core*, dostává napětí 1.2 V. Vnější vývody z pouzdra obvodu se vedou přes hradla z bipolární logiky LVTTL napájená vyšším napětím 3.3 V, které usnadní připojení navazujících obvodů.

Vcc	V_{OL} [V]	V_{IL} [V]	V_{IH} [V]	V_{OH} [V]
1.2 V (FPGA core)	0.3	0.42	0.78	0.9
3.3 V (In/Out Pins)	0.33	1.0	1.65	3.0

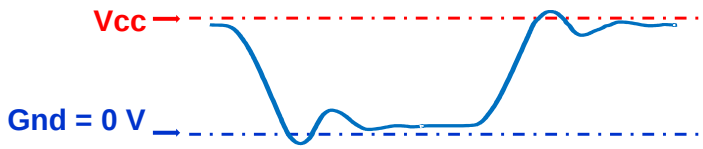
[24] Údaje dle katalogu: Altera: Cyclone IV Device handbook, page 1-12, I/O Standard Specifications, 2016.

Všimněte si, že u vstupů a výstupů máme rozsah logické '1' větší než u logické '0'. Bude-li výstup v '1' (tedy na hodnotě V_{OH}), má vyšší odolnost proti šumu. Právě kvůli tomu se používá tzv. **negativní logika** u některých signálů, které jsou převážně aktivní jen po krátké okamžiky, jako například nulování obvodu po zapnutí napájení, které pak není již aktivní.

V pozitivní napěťové logice se vyšším napětím reprezentuje logická '1' a nižším '0'. V negativní logice jsou prohozené rozsahy '1' a '0'.

V praxi se logické '0' a '1' realizují i různými fyzikálními hodnotami. Na sériových sběrnicích se s výhodou vytvoří proudem, např. 20 mA, kdy logická '0' bude -20 mA, tedy proud tekoucí opačným směrem. Logické '0' a '1' mohou být i změnou fáze signálu, třeba při Manchester kódování, nebo pulzy v případě optických kabelů.

Skutečné průběhy budou ve skutečnosti ještě složitější kvůli odrazům na vedení a mohou se dostat jak nad Vcc, tak pod Gnd do záporných hodnot. Další rozmlžení výstupu přidá všudypřítomný šum vyvolaný vzájemným rušením a špičkami odběru ze zdroje. V logických obvodech neběhají žádné nuly a jedničky, ale komplikované průběhy signálů.



Obrázek 76 - Příklad průběhu reálného napětí na výstupu logického hradla

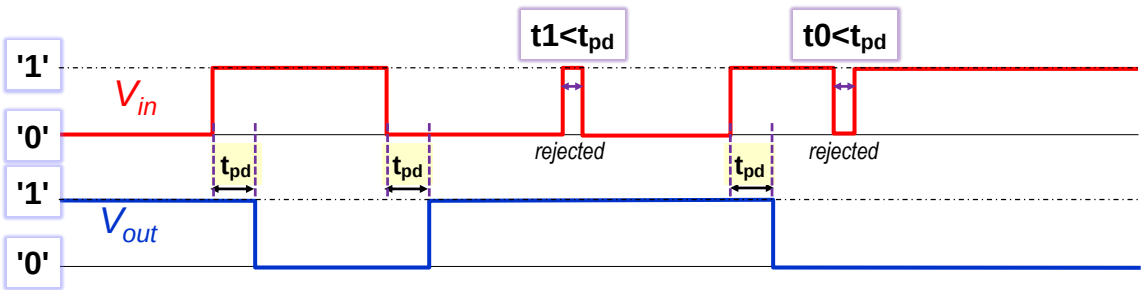
Jak se signály pracujeme v logických návrzích? Jednoduše. Bereme je za logické '0' a '1' a nestaráme o jejich přesnou fyzickou realizaci či napětí. Reálné hodnoty uvažujeme pouze ve výjimečných situacích, např. u přizpůsobení vstupů a výstupů obvodu jeho okolí.

Logické '1' a '0' slouží k usnadnění návrhu — redukujeme jimi složité přechodové děje na abstraktní úrovně.

4.10 Vliv zpoždění na signály

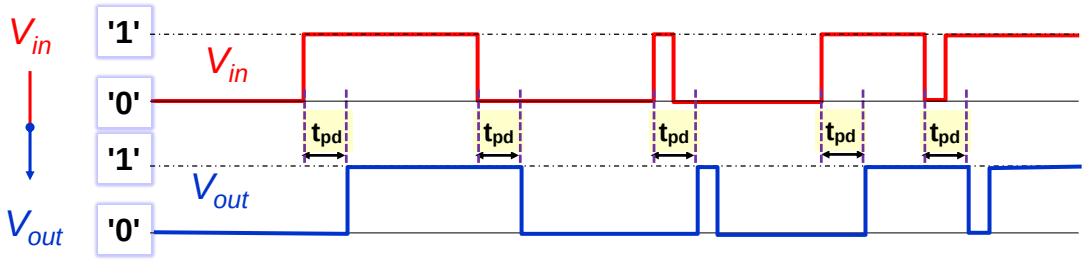
Nebudeme nyní uvažovat skutečné průběhy napětí, ale zjednodušíme-li si pohled na stavy '0' a '1', tedy na situace, kdy V_{in} a V_{out} napětí jsou pod rozhodovací úrovní nebo nad ní.

Lze pak nakreslit jednodušší graf zpoždění hradel, které se řadí do kategorie nazvané ***inertial delay***. Při něm vzniká nejen časový posun výstupu oproti vstupu, ale dojde i ke změně průběhu. Skrz hradla neprojdou kratší pulzy, které nestačily nabít či vybít kapacity, a tak se výstup nezměnil.



Obrázek 77 - *Inertial delay* na hradle

Pouhé časové zpoždění, které nemění průběh signálu, jen ho časově posune, se nazývá *transport delay* (resp. *wire delay*). Mají ho například ideální vodiče.



Obrázek 78 - *Transport delay* na ideálním vodiči

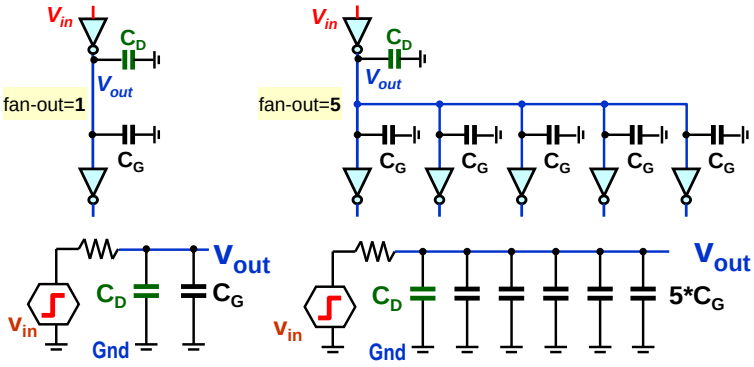
Při zapojení více vstupů se jejich počet označuje termínem ***fan-out*** a se sčítají vstupní kapacity, viz Obrázek 70 na str. 69, jimiž zatěžujeme výstup budícího hradla.

Máme-li změřené jeho zpoždění při buzení jednoho vstupu, lze ho modelovat vztahem, v němž konstanta k_{inv} reprezentuje jeho souhrnný odpor:

$$t_{pd1} = k_{inv} (C_D + C_G)$$

Při rozvedení výstupu invertoru na pět vstupů se nám zpoždění zvýší za předpokladu $C_D \approx C_G$ na:

$$t_{pd5} = k_{inv} (C_D + 5*C_G) = 3 * t_{pd}$$



Obrázek 79 - Vliv zatížení vstupu na zpoždění

Návrhová prostředí budou pečlivě sledovat hodnotu *fan-out* a k jejímu snížení vloží případně oddělující elementy, nám známé prvky *buffer*. Někdy se hodí optimalizovat návrh, pokud to lze, aby se signál nerozváděl na příliš mnoho vstupů. Každý další zvyšuje zpoždění.

Poznámky:

- Jako překročení ***fan-out*** se rovněž hlásí nedovolené spojení více výstupů, tedy analogie zkratu.
- Termín ***fan-in*** udává počet vstupů prvku. Invertor má tedy *fan-in*=1, zatímco čtyřvstupové AND hradlo má *fan-in*=4. Jak jsme se již zmínili, hradla s větším *fan-in* bývají pomalejší, protože v některé jejich části, v horní/dolní skupině spínačů, se musí zapojit více CMOS transistorů do série, což sníží výstupní proud a zpomalí nabíjení kapacit.

4.10.1 Hazardy — přechodové děje v logických obvodech

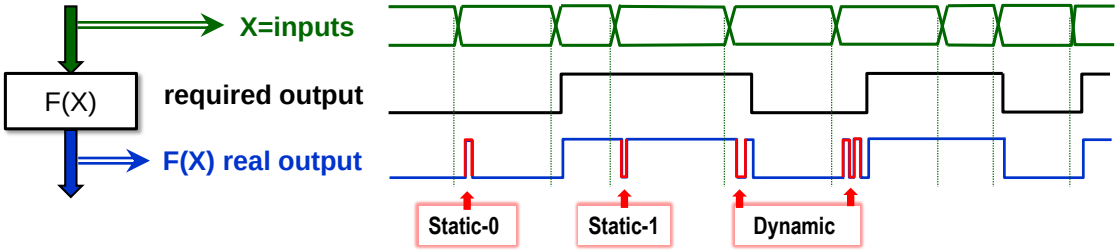
Zpoždění logických hradel, *propagation time delay*, způsobuje přechodové děje v obvodech. Jejich výstup se může vytvářet signály, které mají různé doby šíření uvnitř zapojení, což někdy vyvolá dočasné nechtěné změny, a objeví se nežádoucí pulz *glitch*. Pokud ho logická funkce generuje, pak říkáme, že má hazard.

Pojem „hazard“ pochází etymologicky z arabského slova „*az-zahr*“, které znamená hru v kostky, v níž mnozí přišli o celý svůj majetek. V logických obvodech musíme existenci hazardů vzít v úvahu, jinak náš návrh může rovněž přijít celý vniveč.

Pokud hazardy nastávají v nějakém zapojení, pak se neobjevují vždy, ale jen při určitých přechodech dle vnitřní struktury obvodu.

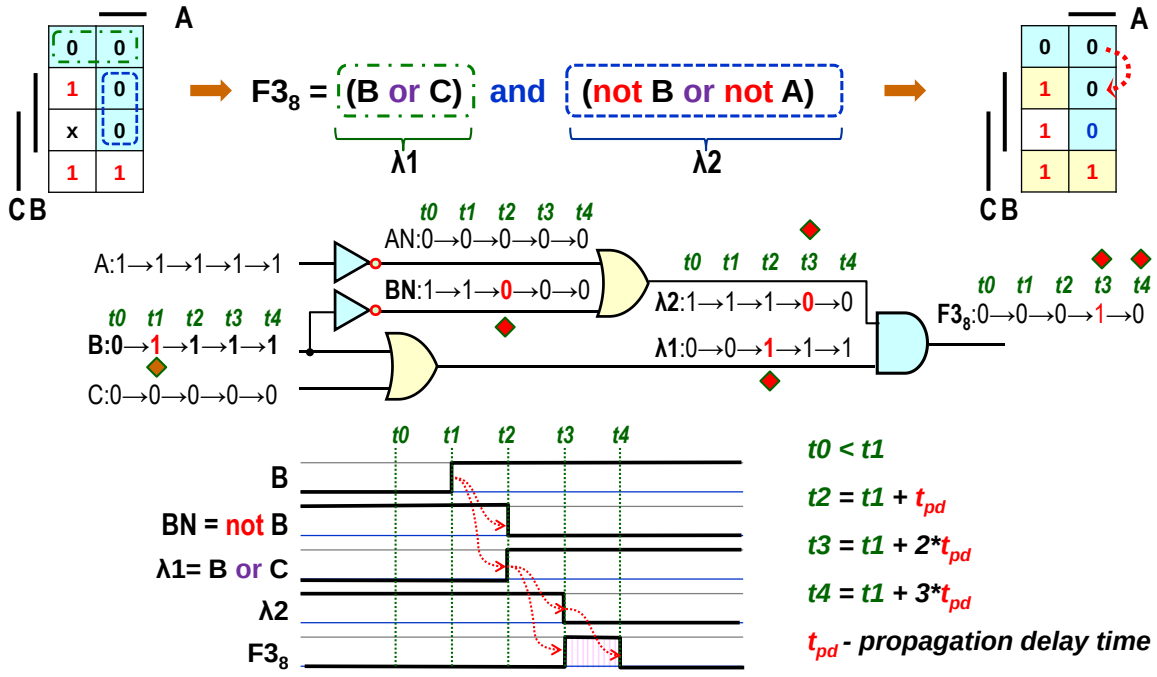
Rozlišují se hazardy:

- **static-0 —** v ustáleném stavu '0' se objeví nechtěný pulz do '1'.
- **static-1 —** v ustáleném stavu '1' se vyskytne nechtěný pulz do '0'.
- **dynamický hazard —** přechod z '0' do '1', nebo naopak z '1' do '0', není hladkou hranou, ale sérií pulzů.



Obrázek 80 - Hazardy

Ukážeme si hazardy na funkci $F3_8$, kterou jsme si vytvořili na str. 48 (Obrázek 41).



Obrázek 81 - Hazardy v logických funkcích

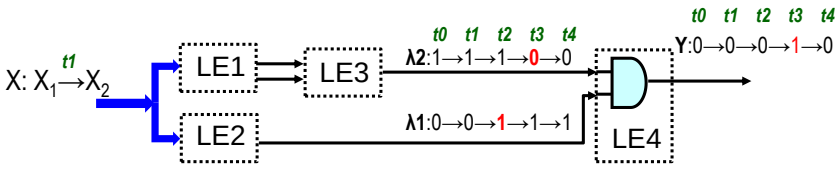
Nechť v čase t0 mají vstupy $F3_8$(A,B,C) hodnoty A='1', B='0' a C=0 a její výstup je ustálený.

1) v čase t1>t0 přejde vstup B z logické '0' na '1'. Jeho změna se bude díky t_{pd} zpoždění hradel lavinovitě šířit zapojením;
2) až v čase $t2=t1+t_{pd}$ ovlivní výstup BN invertoru a λ1 dolního OR-hradla, takže výstupní AND-hradlo $F3_8$ má nyní na obou svých vstupech logické '1';
3) v čase $t3=t1+2*t_{pd}$ se tedy překlopí nejen horní OR-hradlo λ2, ale i AND-hradlo, čímž se změní výstup $F3_8$ na logickou '1', ačkoli by při A='1', B='1' a C=0 měl zůstat v '0'.
4) teprve v čase $t4=t1+3*t_{pd}$ se výstupní AND-hradlo ustálí ve správném stavu '0'.

Při vstupech A='1', B='0' a C=0 bylo totiž AND-hradlo v '0' díky implikant λ1, zatímco při A='1', B='1' a C=0 ho v ní drží implikant λ2, na němž se změna projevila později.

V FPGA se funkce sice tvoří logickými elementy, ale i v nich vznikají rozdílné cesty. Pokud nakreslíme analogii obrázku nahoře, v níž se zamění hradla za bloky LEx, které nám realizují

jakési obecné logické funkce, klidně i složité s násobením a sčítáním, pak při nějaké jejich vhodné vnitřní struktuře může po změně vstupu X z X_1 na X_2 nastat analogická situace.



Dají se hazardy odstranit změnou zapojení v logických kombinačních funkcích?

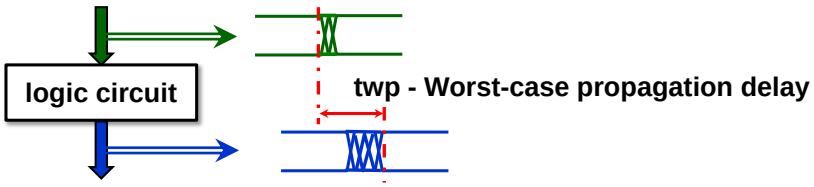
Na běžných FPGA se velmi těžko eliminují. Jejich návrhová prostředí se mohou snažit leda o jejich redukci vyvažováním dob zpoždění na dílčích cestách v logickém obvodu, ale nedokážou je zcela vyloučit. Zpoždění není konstantní, ale mění se s teplotou, která může být uvnitř obvodu různá v jeho odlišných částech.

Hazardy budou též hlavním důvodem, kvůli čemu se v FPGA musíme vystříhat užívání úrovňových klopných obvodů, *latch*, o nichž pojednáme až v kapitole 7.2 na str. 122.

Hazardy lze v kombinačních obvodech vyloučit jenom tehdy, sestavuje-li se obvod přímo z hradel. Navíc je pak nezbytné splnit ještě přísné *Fundamental-mode operation* podmínky, které například předpokládají, že se současně mění pouze jedna proměnné, což lze splnit pouze ve výjimečných případech.

Pokud se totiž změní hodnota několika vstupů v blízkých časových okamžicích, a to dříve než dojde k ustálení výstupu, mohou vzniknout přechodové děje a objeví se *glitch*. Jde o tak zvané funkční hazardy, tedy vycházejí z režimu, v němž využíváme obvod, a ty nelze nikde odstranit pouhou změnou zapojení.

Hazardy lze všude potlačit pomocí synchronních obvodů, jimiž se zapojení taktuje. Bereme vždy v úvahu, že výstup jakéhokoli logického obvodu se nám ustálí až po určité době, tzv. *Worst-case propagation delay*, kterou nám spočítá návrhové prostředí a stanoví i cesty nejpomalejšího šíření změn ze vstupů na výstup.



Obrázek 82 - Worst-case Propagation Delay

Provede-li se změna vstupů, počká se jen na jejich ustálení a za nějaký čas o něco větší než twp, kvůli spolehlivosti, se výstupy vzorkují a zapamatuje se jejich hodnota. Výsledkem bude čistý signál bez hazardů. Synchronní obvody se proberou v samostatné pozdější kapitole.

Kdy se staráme o hazardy?

- Hazardy můžeme zcela ignorovat, pokud se výstup logické funkce přivádí na mnohem pomalejší prvky, třeba na vstup 7-segmentového displeje. Jejich LED diody mají milionkrát pomalejší odezvy.
- O hazardy se musíme starat především u synchronních obvodů, poslední části učebnice. Jejich vstupy hodin a asynchronní nulování zareagují i na krátké pulzy, a tak se na ně nesmí připojit výstup logické funkce, která může generovat hazardy.
- Návrhová prostředí budou někde přidávat i prvky *buffer* k vyvažováním datových cest, zejména v synchronních obvodech.

- Hradla typu **invertor a *buffer* negenerují hazardy**, neboť ty vznikají výhradně při existenci více cest uvnitř logického obvodu. Můžeme je tedy vložit invertor a *buffer* i do kritických signálů

Pozor však na rozvody hodinových signálů! Vložíme-li do jejich cest invertor nebo *buffer*, signál se sice nenaruší hazardy, ale vytvoříme jiný nevítaný efekt. Obě hradla časově zpožďují hodinový signál, takže některé synchronní obvody se klopí o něco později než jiné, což není žádoucí.

Kvůli tomu se doporučuje, aby se do cest hodin nevkládala žádná hradla, je-li možné se tomu vyhnout. Někdy samozřejmě musíme přidat invertor, třeba při změně náběžné hrany na spádovou, nebo *buffer* kvůli oddělení, ale opatrně.

V některých systémech se hodiny běžně zpomalují či se vypínají části obvodu kvůli úspoře odběru, např. v procesorech, tzv. *clock gating*. Jde o regulérní způsob ke snížení spotřeby. Pokud si představíme cestu hodin jako strom, který roste z oscilátoru coby zdroje, tak znepřístupněním momentálně neužívaných větví lze dosáhnout znatelných úspor.

Jde však už o náročné řešení, při němž se složitější synchronní logikou řídí blokování či uvolnění hodin, případně zpomalení jejich frekvence tak, aby se zaručil vhodný časový okamžik změny, ve kterém se nevygenerují rušivé pulzy.

5 Základní kombinační obvody

Kombinační obvody se jen málokdy dají navrhnout jako celek, mnohem častěji se komponují z dílčích stavebních bloků. Kvůli tomu přiblížíme stavební prvky, které se k tomu používají.

5.1 Dekodér 1 z N

Dekodéry 1 z N jsme zmínili již v kapitole 3.2 na str. 31. Mají M vstupů adresy a až N výstupů, kde se $N=2^M$, a existují ve dvou verzích. Čeština je nerozlišuje, ale v angličtině mají odlišné názvy

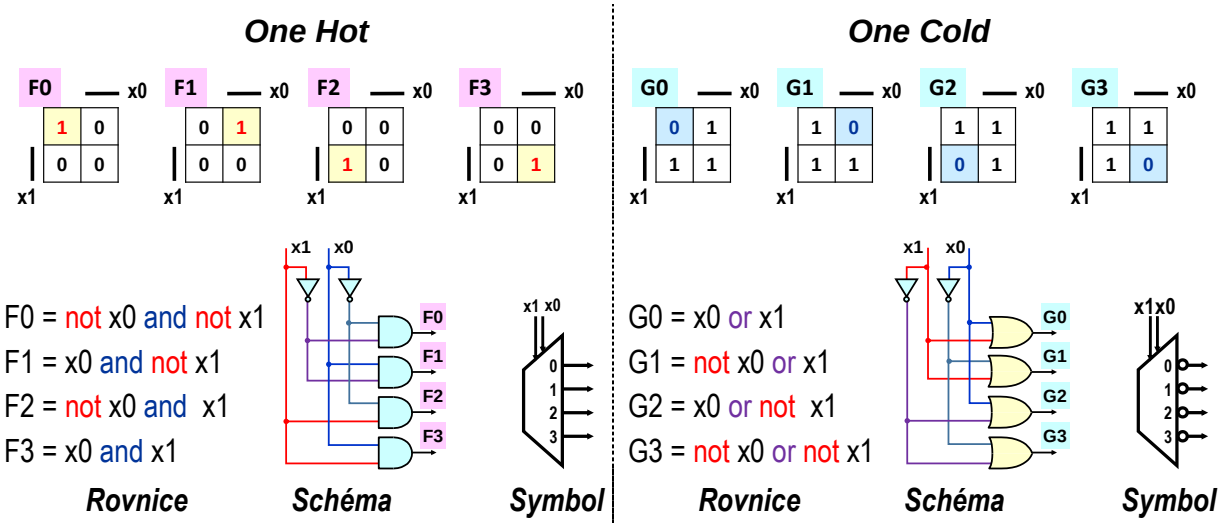
- ***One Hot*** - každý z jeho N výstupů nabývá '1' pouze pro jedinou hodnotu vstupů.
- ***One Cold*** - každý z jeho N výstupů bude v '0' pouze pro jedinou hodnotu vstupů.

	Inputs		*One Hot*				*One Cold*			
N	**x1**	**x0**	**F0**	**F1**	**F2**	**F3**	**G0**	**G1**	**G2**	**G3**
0	'0'	'0'	'1'	'0'	'0'	'0'	'0'	'1'	'1'	'1'
1	'0'	'1'	'0'	'1'	'0'	'0'	'1'	'0'	'1'	'1'
2	'1'	'0'	'0'	'0'	'1'	'0'	'1'	'1'	'0'	'1'
3	'1'	'1'	'0'	'0'	'0'	'1'	'1'	'1'	'1'	'0'

Tabulka 4 - Dekodéry 1 ze 4

Z tabulky nahoře vidíme, že výstupní funkce dekodérů představují mintermy u *One Hot* a maxtermy u *One Cold*. Můžeme si nakreslit jejich Karnaughovy mapy, případně i bez nich rovnou napsat logické rovnice, z nichž pak nakreslíme schéma.

Rovnice Gx funkcí dekodéru *One Cold* lze i odvodit z *One Hot* vztahů pomocí De Morganova pravidla, neboť jsou negacemi Fx, např. G0 = not F0=not (not x0 and not x1)=x0 or x1.



Obrázek 83 - Dekodéry 1 ze 4

Obrázek nahoře uvádí i symboly, jímž se dekodéry vyznačují ve schématech. Značka dekodéru *One Cold* se liší jen přidanými bublinkami invertorů. Výstupy není nutné zapojit, a tak dekodér 1 z N může mít i kratší délku než $N=2^M$, tedy méně výstupních funkcí.

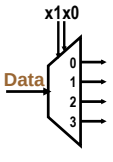
Výstupní funkce dekodéru převádějí vstup ve formě binárního *unsigned* čísla na kódování zvané 1 z N. Máme-li třeba hodnoty v rozsahu 0 až 9, pak je v něm uložíme jako 10 bitové řetězce, v nichž bude v '1' pouze jediný bit. Jeho index udává hodnotu.

Kódování 1 z N se s výhodou používá například v konečných automatech k očíslování jejich stavů. Jejich binární reprezentace má sice větší délku, ale efektivněji zjistíme, zda se dosáhlo požadovaného stavu. Stačí nám testovat pouze výstup jedné Fx či Gx.

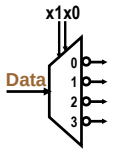
5.2 Demultiplexor

Budeme-li výstupy dekodéru 1 z N spínat datový vstup, takže ho budeme distribuovat na výstup dle zvolené adresy, dostaneme demultiplexor, běžně zkracovaný na Demux. Sám o sobě má omezené uplatnění, ale slouží jako stavební prvek jiných obvodů. Vytvoříme ho, pokud v logických rovnicích, viz Obrázek 83, přidáme jeden člen do každé funkce Fx či Gx. U *One hot* doplníme „and Data“, zatímco u *One cold* „or not Data“, neboť jeho výstupy jsou negované.

D0 = not x0 and not x1 and Data
D1 = x0 and not x1 and Data
D2 = not x0 and x1 and Data
D3 = x0 and x1 and Data

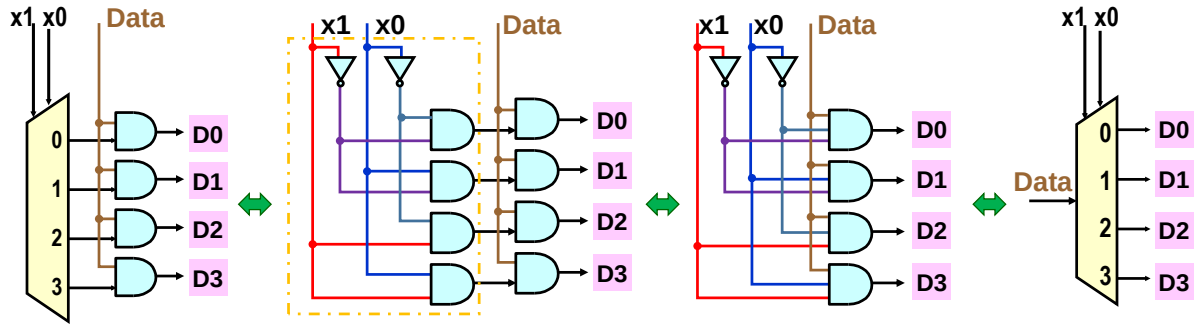


DN0 = x0 or x1 or not Data
DN1 = not x0 or x1 or not Data
DN2 = x0 or not x1 or not Data
DN3 = not x0 or not x1 or not Data



Obrázek 84 - Demultiplexor či Demux 1:4

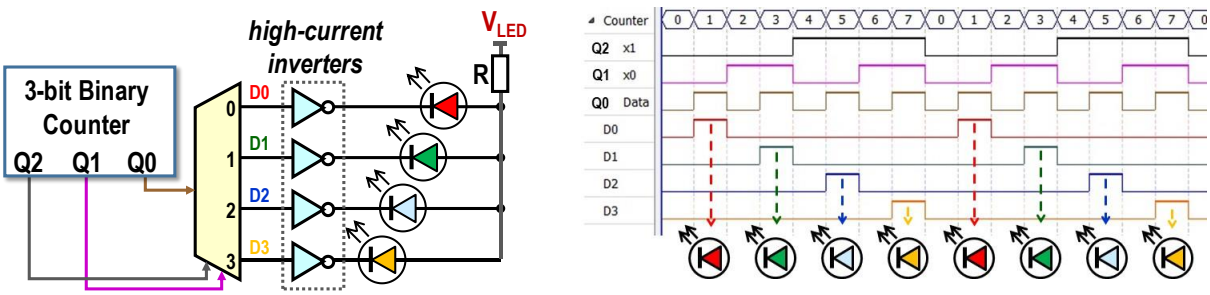
Demux 1:4 rozvádí svůj jediný datový vstup na čtyři různé výstupy. Můžeme ho sestavit přímo z dekodéru 1 z N, nebo také z hradel. Ukážeme si postup na příkladu *One hot* 1 ze 4. V obrázku dole realizují všechna čtyři schémata ekvivalentní funkci. Převod z druhého zapojení zleva na třetí se provedl na základě asociativního zákona, viz strana 16.



Obrázek 85 - Kompozice demultiplexoru 1:4 z dekodéru 1 ze 4

Ukažme si příklad využití Demux k vytvoření blikajícího hada z LED diod. Předpokládejme, že máme tříbitový binární čítač. (Jeho sestavení si ukážeme v 7.5 na str. 138.) Zapojíme jeho výstupy na náš Demux 1:4. Výstup čítače Q0, s nejnižší vahou, povedeme na vstup Data, Q1 připojíme na adresu x0 a nejvyšší bit Q2 na x1.

CMOS obvody malých technologií mají nízká pracovní napětí i proudy a nemusely by naplno rozsvítit LED, které pro maximální jas žádají obvykle kolem 20 mA a napětí 1 V až 4 V, dle jejich velikosti a emitované barvy. Potřebují vyšší napětí, a to označíme V_{LED} a bude 9 V.



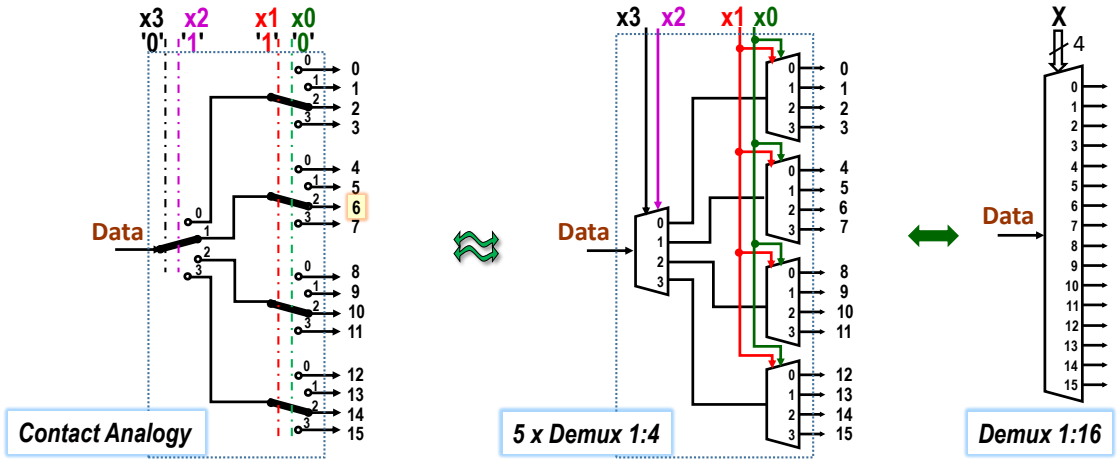
Obrázek 86 - Využití Demux 1:4 na blikajícího světelného hada

Za výstupy Demux připojíme oddělovací invertory určené k převodu úrovní. Ty se vyrábějí jako samostatné součástky. Budou nám spínat LED diody napájené V_{LED} = 9 V. Výsledný obvod vytváří efekt blikajícího světla, které se cyklicky posouvá. Lze použít i Demux s více výstupy a delší čítač, čímž bychom dostali delšího blikajícího hada.

5.2.1 Skupinová minimalizace a Demux 1:16

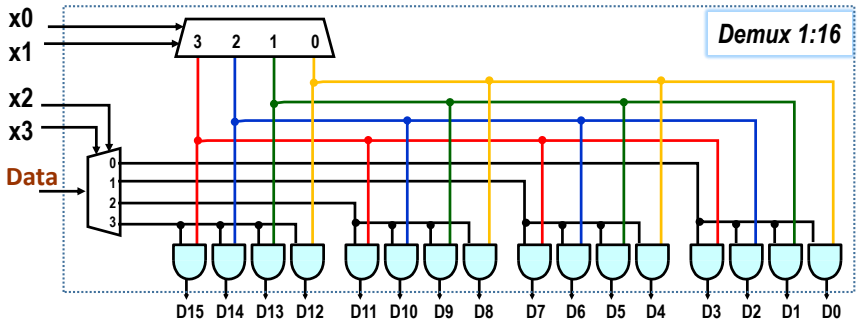
Není šikovné zapojovat Demux 1:16 přímo z jeho rovnic, neboť by vedly na 16 mintermů typu: D0 = not x0 and not x1 and not x2 and not x3 and Data. K nim se musí přidat ještě čtyři invertory adres a *buffer* na Data vstup rozvedený na 16 hradel, aby se redukoval *fan-in* obvodu. Pětivstupová AND hradla budou i pomalejší.

Zkusme jiné řešení. Demux 1:16 si sestavíme z 5 obvodů Demux 1:4.



Obrázek 87 - Demux 1:16 z 5 Demux 1:4

Každý Demux 1:4 obsahuje 4 AND se 3 vstupy a 2 invertory (1 vstup). Počet hradel ještě více snížíme, budeme-li sdílet dekódované adresy, viz obrázek dole.



Obrázek 88 - Optimalizovaný Demux 1:16

Vstup Data demultiplexoru 1:4 se posílá na jedinou čtveřici dvouvstupových hradel AND, která vybere podle dvou horních bitů adresy X. Tři zbylé čtveřice se zablokují logickými '0'. Dekodér 1 ze 4 současně uvolní, dle dvou dolních bitů adresy, pouze jedno hradlo z aktivované čtveřice, a ostatní deaktivuje logickými '0'. Zpoždění odezvy výstupu se u našeho Demux 1:16 zvýšilo o jedno hradlo AND oproti Demux 1:4.

Uvedený postup se nazývá **skupinovou minimalizací**. Při ní neoptimalizujeme přes jednu funkci, ale bereme v úvahu minimální řešení celku.

Jak se řeší skupinová minimalizace při návrhu obvodů? Pokud pracujeme v návrhovém prostředí, můžeme k ní přihlédnout, ale zpravidla se zpočátku o ni příliš nestaráme a necháme

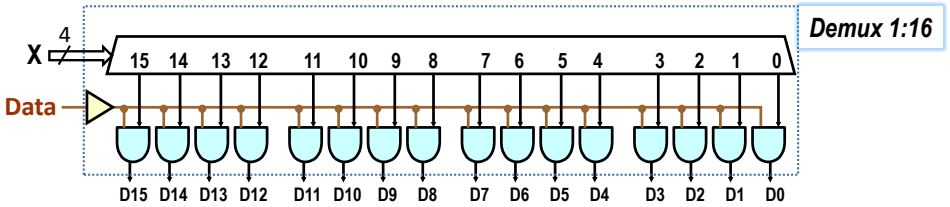
ji plně počítačovým algoritmům. Dbáme jen na to, aby se z našeho popisu obvodu poznalo, co chceme, třeba zmíněný Demux 1:16.

Vylepšování si většinou necháváme až na druhou fázi návrhu, kdy nám už vše funguje. Pokud se nám nějaká dílčí část řešení nelíbí, třeba kvůli jeho odezvě či spotřebě elementů, můžeme zkusit její odlišný popis, jímž návrhovému prostředí vnutíme svou realizaci, třeba ve stylu, který ukazuje předchozí Obrázek 88, či jinou.

Nesmíme se však divit, když si prohlížíme výsledné zapojení automaticky realizované návrhovým prostředím. Můžeme v něm najít různě dekomponovaný Demux 1:16, který vůbec nemusí vypadat jako předchozí, optimalizovaný na použité CMOS.

Pokud se například bude Demux realizovat z jiných komponent, třeba z FPGA logických elementů, může lépe vyjít ze dvou Demux 1:8 a jednoho Demux 1:2.

Popřípadě ho lze vytvořit dekodérem 1 ze 16, jehož výstupy aktivují jedno ze šestnácti AND hradel, což především minimalizuje zpoždění z Data na Dx výstupy.



Obrázek 89 - Jiné řešení Demux 1:16 pomocí dekodéru 1 ze 16

Druhý vstup AND hradel má Data rozvedená z výstupu výkonného hradla typu *buffer* schopného zvládnou *fan-out* 16. Alternativně lze použít i rozvedení Data z několika paralelních prvků *buffer*, třeba ze dvou, každý pak budí jen vstupy dvou čtveřic hradel.

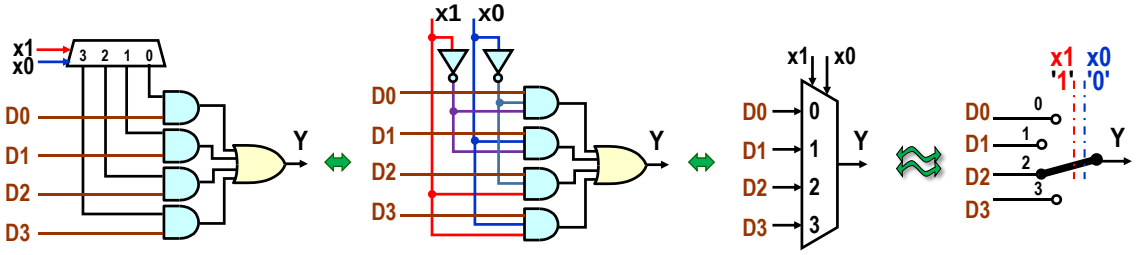
Příklad skupinové minimalizace současně prokázal, že v obvodech nemusí nutně existovat jediné optimální řešení. I tak jednoduchý Demux 1:16 se dal realizovat několika způsoby. Každý z nich přinášel jiné výhody.

5.3 Multiplexor

Multiplexor funguje reverzně k demultiplexoru. Má 2^M datových vstupů, kde M je počet bitů vstupní adresy. Podle její hodnoty posílá vstup Di na výstup. Jeho název se běžně zkracuje na Mux, alternativně se používá i pojmenování „data selektor“, neboť pracuje analogicky jako otočný vícepolohový přepínač.

Multiplexor 4:1 lze sestavit z dekodéru 1 ze 4 a hradel, či přímo z jeho rovnic:

Y = (not x0 and not x1 and D0) or (x0 and not x1 and D1)
or (not x0 and x1 and D2) or (x0 and x1 and D3)



Obrázek 90 - Multiplexor 4:1

U prvku Demux, respektive dekodéru 1 z N, jsme mohli mít až 2^M výstupů. Nepotřebné šlo vynechat, jelikož není nutné zapojit každý výstup. Vstupy se však musí vždy definovat.

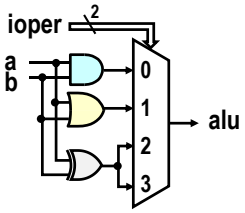
Multiplexor N:1 musí znát hodnoty všech svých N=2^M vstupů. Pokud jich nějaká aplikace tolik nepotřebuje, i tak se musí zadat všechny zbylé, akorát zapojené na logické '0' či '1', aby existovala jasně určená hodnota výstupu pro každou adresu od **0** až do **2^M-1**.

Co se v hardwaru interpretuje multiplexorem N:1?

Na něj se převádějí například přepínací operace, podobné příkazu **switch**() jazyka C, avšak implementace v obvodu vyžaduje, aby vždy existovala default hodnota.

Jazyk C++	Hardware

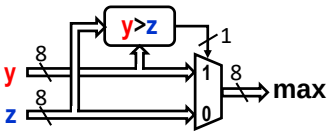
```
bool alu(int ioper, bool a, bool b)
{   switch(ioper)
    { case 0: return a && b;
      case 1: return a || b;
      default: return a ^ b;
    }
}
```



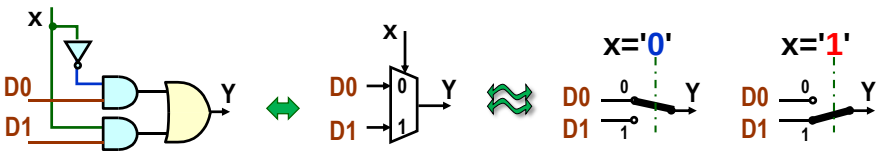
Přepínáním na multiplexorech se též realizují podmíněné příkazy typu **if-then-else**, respektive podmíněná přiřazení, která rovněž vedou na multiplexory 2:1.

Jazyk C++	Hardware

```
byte max(byte y, byte z)
{   return y>z ? y : z;
    //  if(z>y) return z; else return y;
}
```

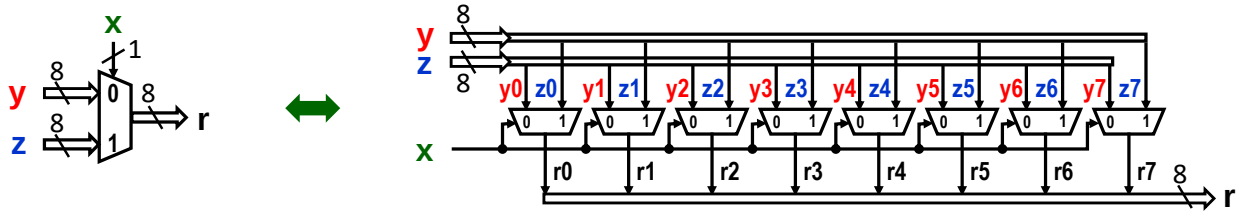


Předchozí obvod využívá komparátor. Ten bude probraný v kapitole 6.1 na str. 97. Výsledek jeho porovnání řídí osmici jednoduchých multiplexorů 2:1, které lze pokládat za analogií přepínače, o čemž jsme se již zmínili na str. 53.



Obrázek 91 - Multiplexor 2:1 jako přepínač

Při přepínání sběrnic se použije pro každý bit jeden, tedy tolik kolik má vodičů. Napojení vstupu x adresy na každý z osmi multiplexorů se úsporněji vyjádří pomocí signálu x vedeného skrz ně, tedy stylem, jímž se nejčastěji kreslí sítě multiplexorů či jiných prvků.



Obrázek 92 - Multiplexor 2:1 osmibitové sběrnice

Mux 2:1 posílá jeden ze dvou svých vstupů na výstup r. Ve skutečnosti tak provádí analogii výběru prvku z pole, zde o dvou prvcích, a to podle indexu, jímž x vstup.

Příklad: Výraz s více xor jsme probrali v kapitole 2.3.1 na str. 23 , v níž jsme matematickou indukcí dokázali, že vrací '1' při lichém počtu vstupních bitů. U třech vstupů má rovnici:

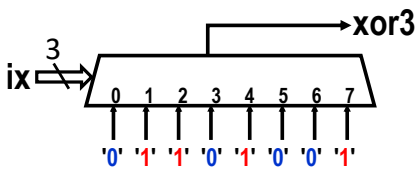
xor3(ix2, ix1, ix0)=ix2 xor ix1 xor ix

Alternativně ji lze popsat seznamem mintermů, viz kapitola 3.2 na str. 31, coby úsporným zápisem jeho pravdivostní tabulky. Pak popíšeme xor3 jen seznamem indexů, při nichž existuje lichý počet vstupních bitů, tedy xor3 dává na výstupu '1', tedy xor(ix2, ix1, ix0): m(1,2,4,7)

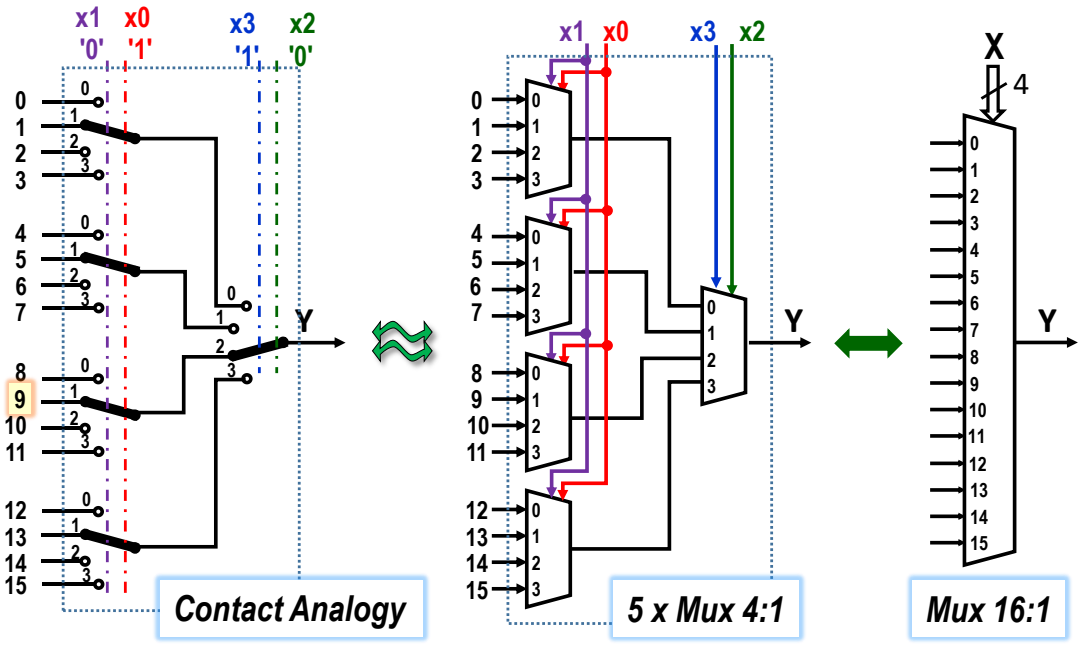
Můžeme funkci realizovat i multiplexorem s adresou X o délce 3 bity, kterou se vybere prvek z pole o délce $2^3 = 8$ prvků, do něhož uložíme '1' na uvedené indexy, a jinde budou '0'. Multiplexory dovolují tak vyjádřit libovolnou kombinační logickou funkci[25].

Jazyk C++	Hardware

```
bool xor3(int ix)
{  // xor(ix2, ix1, ix0): m(1,2,4,7)
    bool pole[8] = { 0, 1, 1, 0, 1, 0, 0, 1 };
    return pole[ix & 0x7];
}
```



Vícevstupové multiplexory lze s výhodou sestavovat z menších. Pokud například potřebujeme multiplexor 16:1, pak ho lze vytvořit třeba z patnácti Mux 2:1, nebo dvou Mux 8:1 a jednoho Mux 2:1, případně z pěti Mux 4:1, jak ukazuje obrázek dole i s přepínačovou analogií, v níž se naznačuje stav voličů při adrese x="x3 x2 x1 x0"="1001 převedené jako *unsigned* na 9.



Obrázek 93 - Multiplexor 16:1

Všimněte si **pořadí bitů adresy**. Levá řada multiplexorů používá nižší bity adresy X, neboť ve výběru hodnoty má menší prioritu než výstupní Mux 4:1. Ten leží až za ní, a tak musí dostávat horní bity adresy. Jedině tak se nám vstupy očíslují v souvislé řadě.

[25] Připomínáme, že Shannonova expanze provede rozklad logické funkce na dva kofaktory, tedy funkce na vstupech multiplexoru 2:1, viz Obrázek 46- Shannonova expanze na str. 42. A kofaktory lze dál rozkládat na menší.

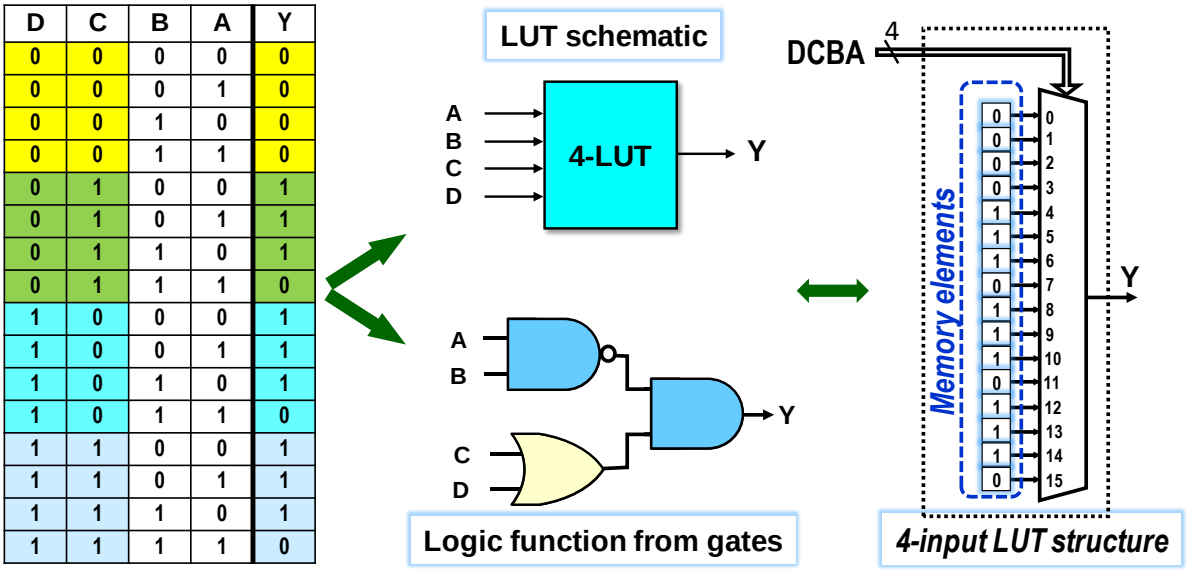
Kdyby se skupiny bity x3 a x2 adresovala levá řada Mux 4:1, zatímco x1 a x0 výstupní Mux 4:1, pak by se dostal chybný převod $x3*2^1+x2*2^0+x1*2^3+x0*2^2$ binárního čísla interpretovaného jako *unsigned*. Při změně X od 0 do 15 by se na výstup Y posílaly vstupy v pořadí:

0, 4, 8, 12, 1, 5, 9, 13, 2, 6,1 0, 14, 3, 7, 11,15.

5.4 LUT tabulky FPGA

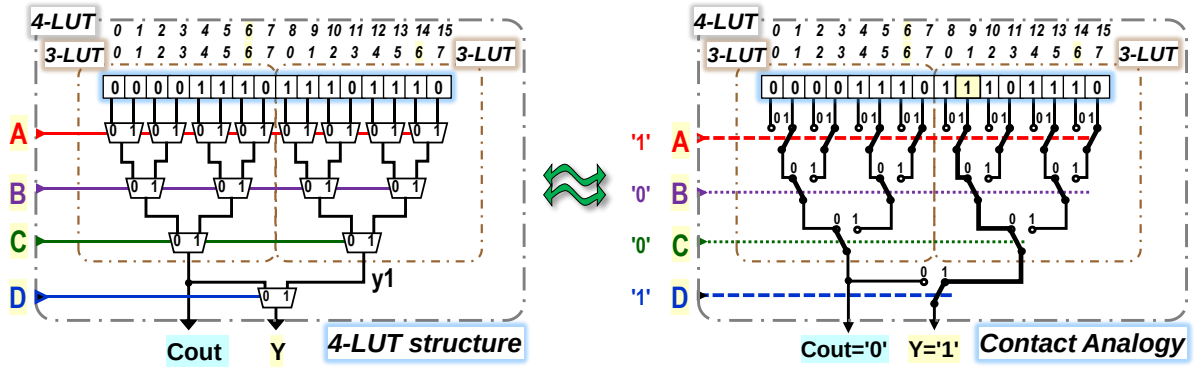
V FPGA se kombinační logika realizuje pomocí LUT. Jejich vstup X vybírá hodnotu logické funkce z její pravdivostní tabulky. Během konfigurace FPGA se hodnoty logické funkce uloží do paměťových buněk, ty se poté chovají jako konstanty, neboť drží své hodnoty až do nové změny celého FPGA na jiné zapojení.

Podobné výběrové logice se říká **LUT, *Lookup Table***, tedy vyhledávací tabulka[26]. V obvodech ji lze elegantně řešit rovněž multiplexorem. Obrázek dole ukazuje typ LUT se čtyřmi vstupy, pro který se často zavádí označení 4-LUT:



Obrázek 94 - 4-LUT - čtyřvstupová LUT

LUT mají různé vnitřní struktury. Lze je například složit z Mux 2:1. Obrázek uvádí i její přepínačovou analogii se zvýrazněnou cestou v případě vstupů DCBA="1001".



Obrázek 95 - Možné řešení 4-LUT

Každý vstup 4-LUT vykazuje jinou dobu zpoždění, *propagation delay*. Nejrychleji se na výstupu Y projeví změna hodnoty vstupu D, kdy dojde jen k výběru jednoho ze dvou výsledků vyšší řady ovládané vstupem C, jenž je volí ze čtyř výstupů řady B. Ze vstupu C pak signálová cesta povede na výstup Y skrz dvě řady multiplexorů. Ze vstupu B se prodlouží na tři.

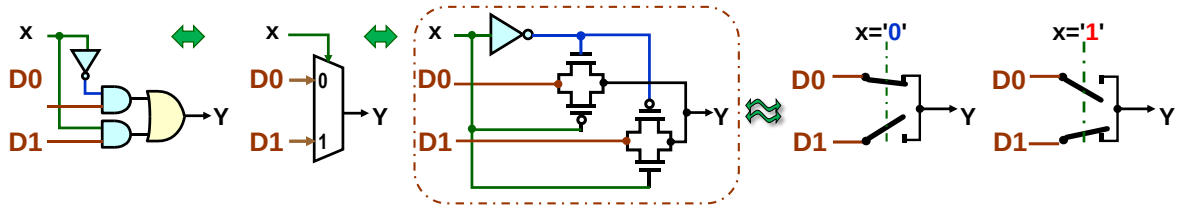
[26] Z matematického hlediska provádí LUT restrikci zobrazení na omezený definiční obor. Realizuje se datovou strukturou, která nahrazuje výpočet nalezením hodnoty. LUT se používají nejen v FPGA, ale i v klasickém programování k vyčíslení složitých funkcí, u nichž se případné mezihodnoty dají stanovit interpolací.

Nejdelší zpoždění má vstup A, a to čtyř multiplexorů. Po změně jeho hodnoty se najednou přepne všech osm horních Mux 2:1. Osmice jimi vybraných hodnot z 16 paměťových členů se pak šíří přes nižší řady, jejichž aktuální stavy určí, která z nich pronikne až na výstup Y.

Má-li jich implementovaná logická funkce méně, upřednostní se rychlejší vstupy LUT. Například dvouvstupové XOR se popíše pravdivostní tabulkou logické funkce Y = C xor D. Nepoužité pomalejší LUT vstupy, zde A a B, se připojí třeba na '1'.

Zpoždění XOR nebude však poloviční oproti čtyřvstupové logické funkci, jen kratší o dva multiplexory. Spoje k LUT vedou pokaždé přes další členy, jejichž zpoždění se rovněž přidá. Máme-li obvod složený z více dvouvstupových funkcí, pak i nevelké urychlení každé z nich se příznivě projeví na celkovém zpoždění.

Ve vnitřní struktuře FPGA se Mux 2:1 zapojují z *transmission gates* probraných v kapitole 4.5 na str. 63. Zpoždění mezi vstupy D0 a D1 multiplexoru a jeho výstupem závisí jen na nabíjení kapacit navazujících vodičů, u krátkých bude mizivé.



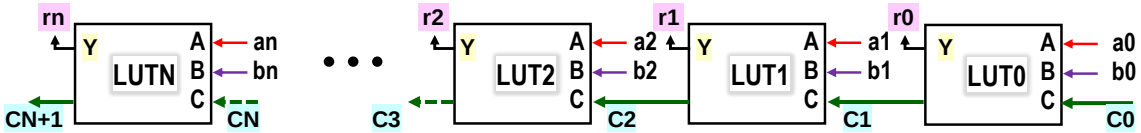
Obrázek 96 - MUX 2:1 z *transmission gates*

Mux 2:1 se složí ze dvou *transmission gates* a invertoru. Ten v sítích multiplexorů sdílejí i další členy se stejnou adresou, třeba celá osmice ovládaná vstupem A.

Pro úplnost dodáme, že existují i úspornější realizace výběrové logiky LUT, například na bázi proudových zdrojů[27], v nichž lze *transmission gates* nahradit jednoduchými NMOS transistory.

Celkovou spotřebu CMOS transistoru na jednu LUT však nejvíce determinují její paměťové buňky. Každá z nich potřebuje rovněž logiku ke svému nastavení během programování FPGA obvodu na nové zapojení. Je-li na bázi CMOS pamětí RAM, pak každá buňka má 8 až 12 CMOS transistorů dle svého konkrétního provedení.

V řadě obvodů, třeba u sčítaček nebo komparátorů, se potřebuje rychlé šíření přenosu, *carry propagation*. Každý dílčí stupeň zpracovává dva bitové vstupy a k tomu ještě přenos z nižšího řádu. Generuje jak bit výsledku, tak svůj přenos určený vyššímu řádu. Výsledky celého řetězce budou definitivní, teprve až po ustálení všech přenosů, což rozhoduje o rychlosti celé komponenty.

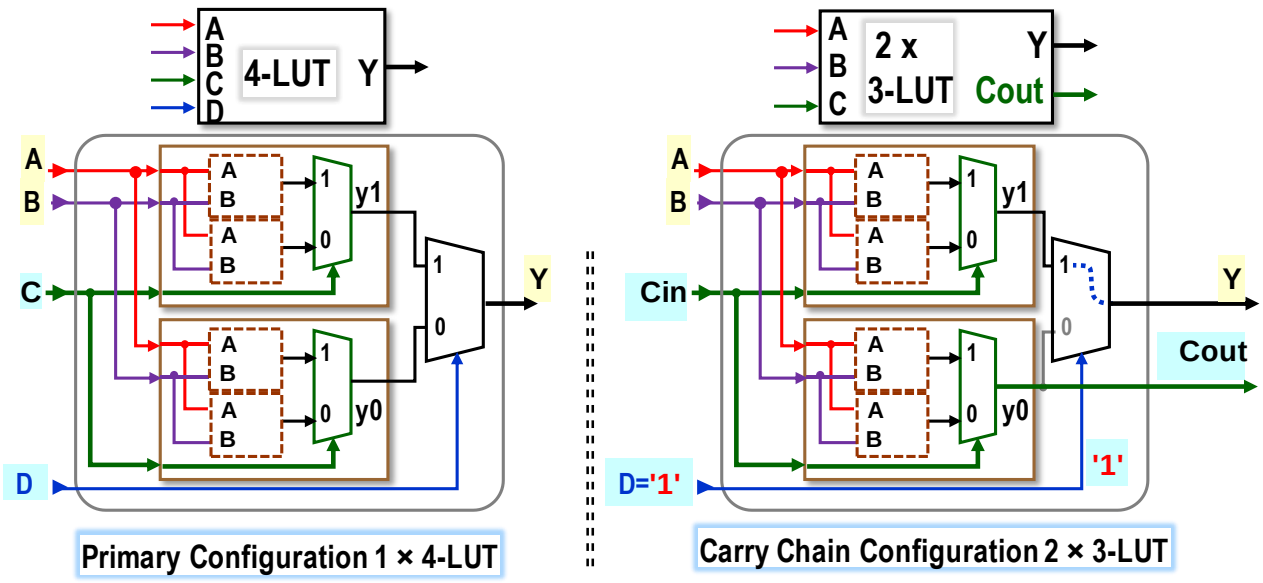


Obrázek 97 - Šíření přenosu

LUT lze nakonfigurovat k urychlení přenosů. V následujícím obrázku je rozkreslena 4-LUT na dvě 3-LUT. V běžné 4-LUT konfiguraci se mezi nimi přepíná vstupem D. Pokud se D interně připojí na '1', lze z dolní 3-LUT vyvést Cout, *Carry Out*. Mezi Cin vstupem a Cout výstupem leží nyní zpoždění jediného dvouvstupového multiplexoru. A ten se řadí mezi rychlé prvky. Obě 3-LUT sdílejí A a B vstupy jako své pracovní a C coby přenos.

---

[27] Suzuki, Daisuke et al. "Area-efficient LUT circuit design based on asymmetry of MTJ's current switching for a nonvolatile FPGA." *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)* (2012): 334-337.



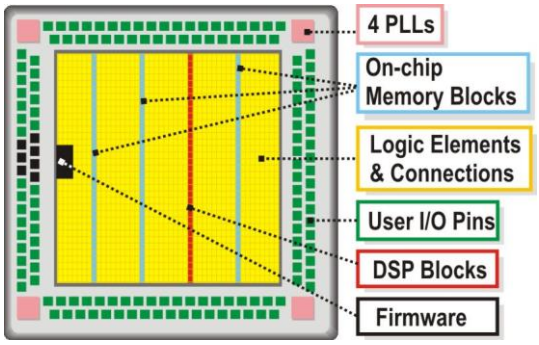
Obrázek 98 - Konfigurace 4-LUT na zrychlené šíření přenosu

5.5 Vnitřní struktura FPGA obvodu

Co se nám vlastně zapojilo? Otázka bude aktuální, sotva popíšeme nějaké zapojení ve zvoleném návrhovém prostředí, které naše zadání optimalizuje. Výsledek pak nahraje do FPGA obvodu. Měli bychom se pak vždy podívat, jak se vše realizovalo, zda náš popis vedl na přijatelnou vnitřní strukturu zapojení, nebo bude nutné zvolit jinou techniku rozkladu problému na dílčí části. Kvůli tomu musíme něco málo znát i o vnitřní struktuře FPGA obvodů.

V katalozích výrobců najdeme řadu vnitřních uspořádání FPGA, dle jejich konkrétního zaměření a typu.

Na ukázku jsme si vybrali starší obvod FPGA Cyclone II typ EP2C35F672 technologie 90 nm, který má sice méně elementů než FPGA Cyclone IV E používaný v našich novějších vývojových deskách (Obrázek 2 na str. 9), ale obsahuje veškeré jeho stavební prvky.



Obrázek 99 - FPGA Intel Cyclone II

Jiné typy FPGA používají podobnou strukturu, a tak výklad platí i pro ně.

5.5.1 User I/O Pins

Pouzdra FPGA mají stovky vývodů, z nichž většina je uživatelská, na něž můžeme nasměrovat naše vnější fyzické vstupy či výstupy, což ulehčí rozvržení plošného spoje. FPGA z obrázku nahoře má 672 vývodů, z nichž 475 můžeme využít dle našich potřeb a struktury obvodu.

Používáme-li FPGA na již hotové vývojové desce, pak pozice vstupů a výstupů pevně určuje její plošný spoj. Jejich rozmístění si jen načítáme ze seznamu zvaného *Pin Assignments*, který obsahuje i volitelná symbolická jména vstupů a výstupů. Lze tak přehledně psát třeba jen CLOCK_50 místo přesného indexu fyzického vývodu.

5.5.2 DSP bloky

Bloky DSP, *Digital Signal Processing*, zabudované v FPGA obvodech, se vyrábějí jako univerzální pro velkou šíři operací. Nejčastěji provádějí rozličná násobení, včetně operací s čísly

v pohyblivé řádové čárce. Další DSP realizují například digitální filtry, či FFT, rychlou Fourierovu transformaci.

FPGA Cyclone II zahrnuje 35 DSP bloků. Všechny obsahují 18x18 bitů integer hardwarové násobičky. Každou z nich lze ale nakonfigurovat na dvě nezávislé 9x9 bitů integer násobičky. Na princip hardwarových násobiček se podíváme v kapitole 6.3.6 na str. 112.

5.5.3 PLL - fázový závěs

Zkratka PLL pochází z *Phase-locked Loop* a označuje obvod, který náleží k běžné výbavě pokročilejších obvodů, a to nejen FPGA, ale i další techniky. Má četná použití, z nichž vybereme jen některá. Rekonstruují se jim například hodiny z přenášených dat po sériových linkách, tzv. *clock recovery*, a existují i analogové PLL k frekvenční demodulaci.

Vybraný FPGA obsahuje čtyři digitální PLL, které umí vstupní frekvenci hodin vynásobit zlomkem, jehož číselná hodnota může být i mnohem větší než 1. Lze je nezávisle nakonfigurovat na tvorbu námi požadovaných frekvencí odvozených od základních hodin.

Krystaly užívané v oscilátorech lze totiž vyrobit jen do kmitočtů desítek MHz a vyšší frekvence se tvoří na PLL. Například i během uživatelského přetaktování procesoru či grafických karet měníte jen hodnotu zlomku, jímž se násobí frekvence krystalového oscilátoru. Princip operace se přiblíží v přednáškách našeho předmětu LSP.

5.5.4 Firmware

Většina FPGA obvodů v sobě obsahuje i veškerou výbavu potřebnou k jejich naprogramování na jiné zapojení[28]. Jejich firmware zahrnuje nejčastěji rozhraní sériové sběrnice JTAG[29], která je zavedeným průmyslovým standardem jak ke konfiguraci obvodů, tak k monitorování jejich stavů. Vývojové desky se běžně prodávají s převodníky mezi USB a JTAG. Stačí nám jen nainstalovat příslušný ovladač do našeho operačního systému.

5.5.5 On-chip Memory

Paměti umístěné přímo v FPGA se též nazývají ***Embedded memory***. Jejich možné využití budou třeba tabulková řešení goniometrických funkcí, nebo vyrovnávacích paměti přijímaných dat typu FIFO, *First In, First Out*. V FPGA se sestavují z paměťových bloků pevné délky v kilobitech. Vždy se použije celý blok, i kdybychom do něj uložili byť jediný bit.

Zde vybrané FPGA obsahuje 472 kilobitů rozdělených do 105 paměťových bloků nazvaných M4K. Každý uchová 4 kilobity plus dalších 512 bitů k vnitřní paritě. Lze z něho vytvořit jak paměť 4kb×1, tedy 4096 bitů, nebo na 2kb×2 potřebujeme-li 2 bitový výstup, respektive v jiných variantách, například paměť 512 bytů s paritou. Rozsáhlejší paměti o větší šířce dat lze sestavit z několika bloků. Jejich požadovaná konfigurace se specifikuje přes vývojové prostředí výrobce, a tak vytvoření paměti i její užití je relativně snadné. Ukážeme si ho na cvičeních našeho předmětu LSP.

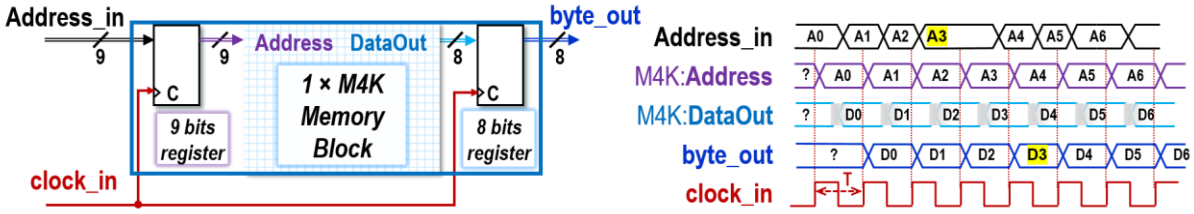
Vnitřně jsou paměťové bloky založené na SRAM typech pamětí, tedy Static RAM, *Random Access Memory*, tedy stejných jako ve většině jiných FPGA. Jejich obsah lze i inicializovat během konfigurace FPGA a využít je i jako ROM, *Read-Only Memory*.

---

[28] Pouze FPGA obvody založené na antifuse paměťových elementech vyžadují externí programovací zařízení kvůli potřebě napěťových pulzů. Jejich konfigurace je již permanentní, nejde ji změnit.

[29] O JTAG se lze dočíst třeba na Wikipedii či na https://www.xjtag.com/about-jtag/what-is-jtag/

Princip SRAM si však žádá uložení vstupní adresy do registru, neboť musí zůstat neměnná po dobu nutnou k vybavení dat. Ta se však objevují na výstupu s variabilním zpožděním, a tak se doporučuje přidávat i jejich výstupní registr. V obrázku dole se po změně adresy na hodnotu označenou A3 objeví data na výstupu se zpožděním téměř dvou period T hodin.

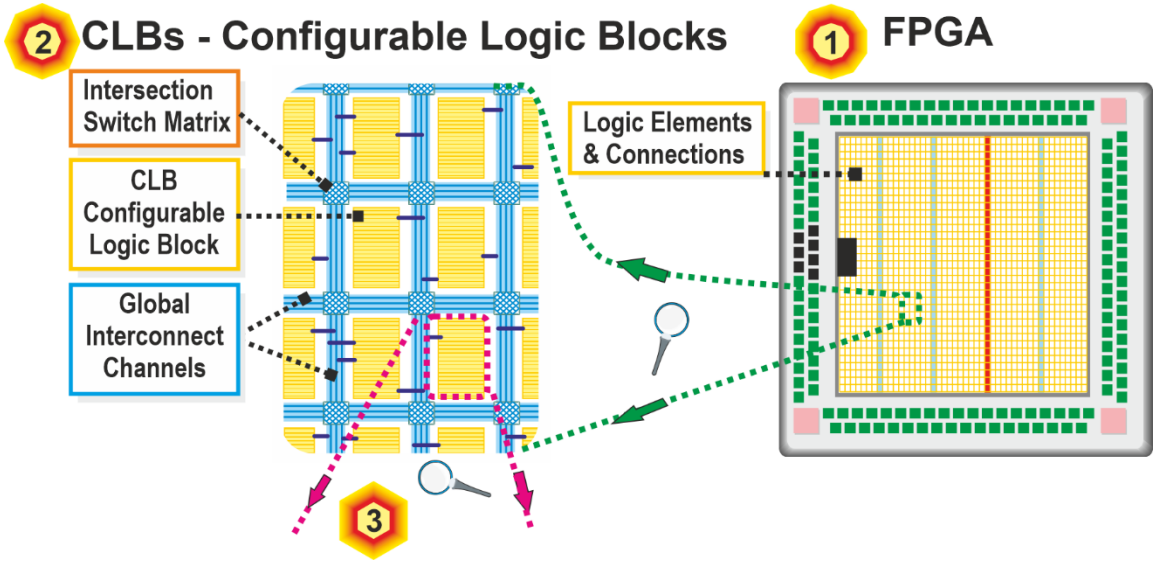


Obrázek 100 - M4K v konfiguraci ROM paměti 512 bytů

SRAM v FPGA mívají běžně dvojí výběrovou logiku, *2-port SRAM*, takže lze najednou číst data ze dvou různých adres, či na ně zapisovat, a každý přístup i řídit jinými hodinami. V nitru některých procesorů se používají SRAM i s početnější výběrovou logikou, trojí a více.

5.5.6 Logické elementy a propojky

Logické elementy, LE, představují základní stavební prvky FPGA a obsahuje je každý jejich typ. Skládají se nejméně z jedné LUT se synchronním klopným obvodem a konfigurační logikou. K přiblížení jejich struktury použijeme lupu, kterou si postupně zvětšíme logické elementy a propojky v FPGA.



Obrázek 101 - Struktura FPGA: Konfigurovatelné logické bloky CLBs

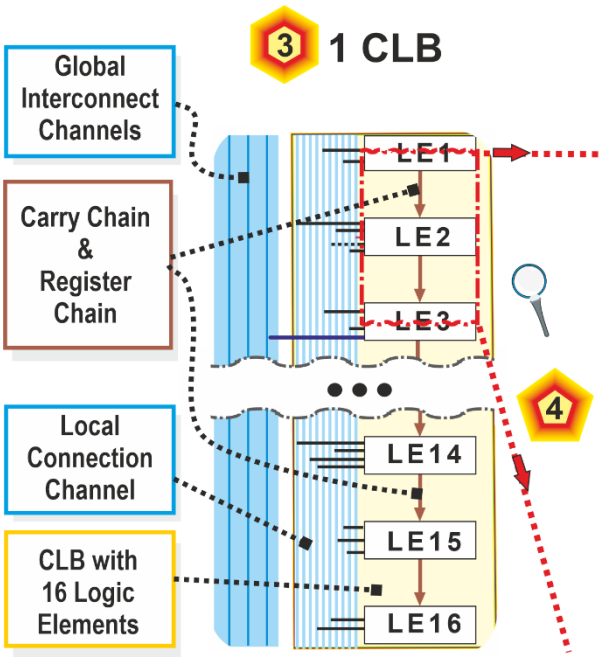
Uvnitř FPGA, viz [1], se logické elementy sdružují do konfigurovatelných logických bloků **CLB**, výřez [2]. Na strukturu CLB se podíváme až v následujícím výřezu [3].

Mezi CLB se nacházejí nejkritičtější součásti všech existujících FPGA obvodů, a to konfigurovatelné propojky. Sdružují se v kanálech, *Interconnect Channels*, v nichž existují v různých délkách, od krátkých až po dlouhé, někdy i s opakovači signálů. Syntézní nástroj vývojového prostředí si z nich vybírá spoje dle jejich potřeby a dostupnosti. Na jejich křížení se určuje jejich vzájemné propojení v *Intersection Switch Matrix*, konfigurovatelné matice přepínačů.

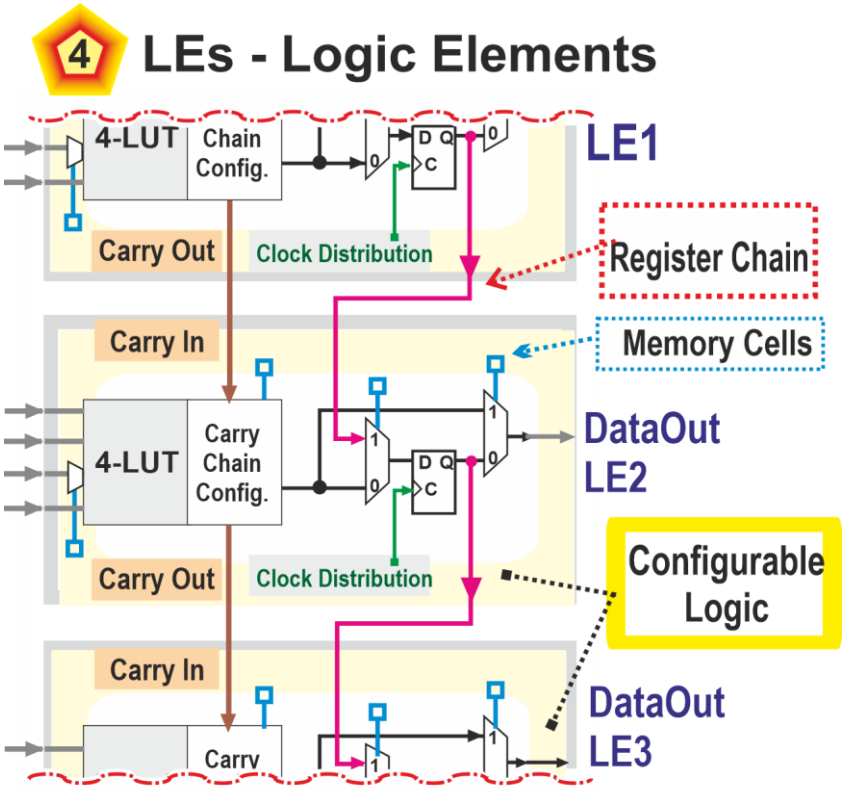
Výřez [3] ukazuje nitro jednoho konfigurovatelného logického bloku CBL. Obsahuje 16 logických elementů, LEs, na něž se podíváme až v následujícím výřezu [4].

Logické bloky, LEs, jednoho CLB lze propojovat pomocí vodičů v lokálním spojovacím kanálu. Vstupy či výstupy LE můžeme také napojovat na globální vodiče, pokud potřebujeme přijmout signál od jiného CLB či mu poslat výstupy.

Mezi LEs vedou i přímé spoje, ale jen k fyzicky následujícímu LE. Jimi se realizují rychlé přenosy, Carry, nebo spojení LEs do vícebitové logiky.



Obrázek 102 - Struktura FPGA: Konfigurovatelný logický blok CLB

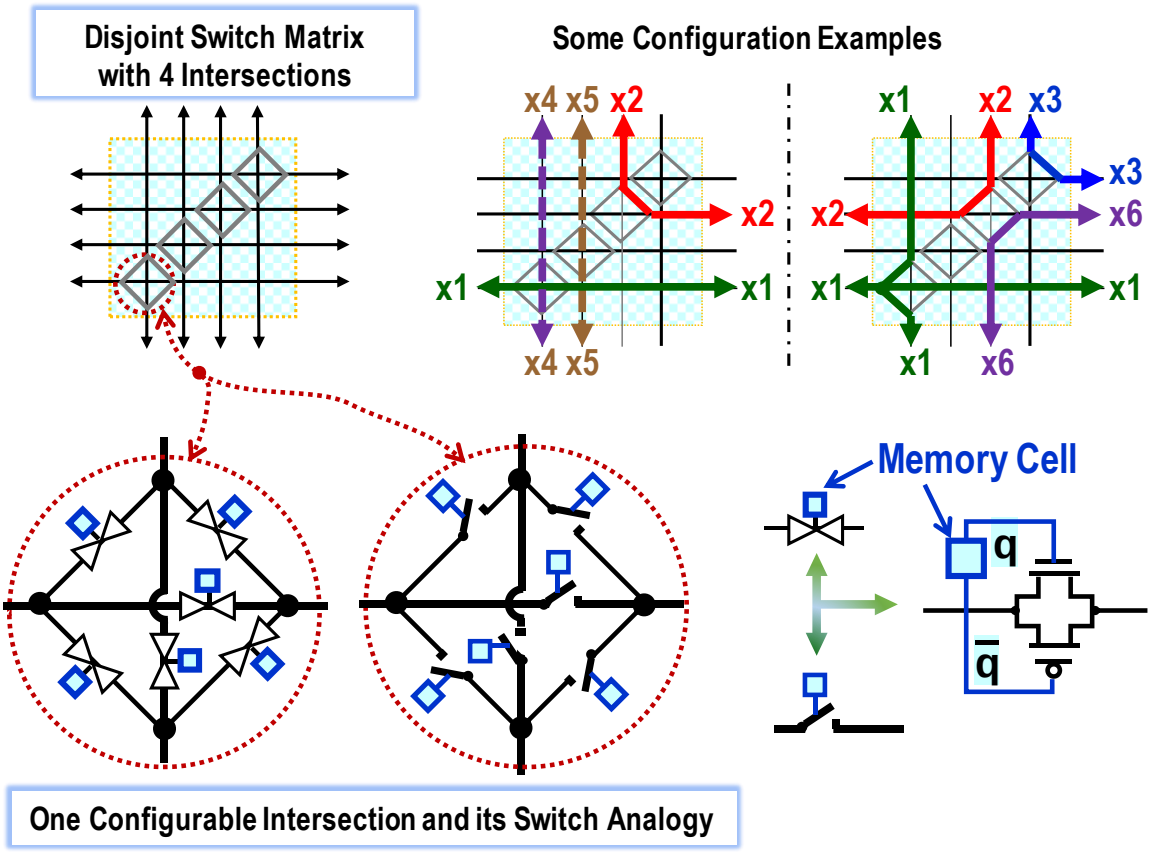


Obrázek 103 - Obrázek 100 - Struktura FPGA: LEs-logické elementy

Poslední výřez [4] ukazuje přímé propojení LE2 s předchozím LE1 a následným LE3. Jedny přímé propojky slouží jako již zmíněný *Carry Chain*, viz dále i kapitola 6.1. Druhým typem jsou spojky, *Register Chain*, jimiž se zřetězí klopné obvody uvnitř LEs tak, aby výstup jednoho vedl na vstup následujícího, což se hodí třeba u posuvných registrů. Propojení se řídí multiplexory 2:1, jejichž vstup adresy je zapojený na paměťovou buňku (*Memory Cell*) nastavenou při konfiguraci FPGA. Ta drží poté svou hodnoty až do nahrání nového zapojení.

Paměťovým buňkám konfigurace věnujeme celou kapitolu 5.6 začínající na str. 95.

Zmíněné matice přepínačů, *Intersection Switch Matrix*, zmíněné na výřezu [2], Obrázek 101, obsahují nastavitelná křížení. Obrázek dole uvádí jednoduchý typ matice *disjoint* (cz: rozpojovací?), která šesticí *transmission gates* propojuje vodiče jen v diagonále jejich křížení [30].



Obrázek 104 - Propojovací matice typu *disjoint*

Výřez 3 na předešlé stránce, Obrázek 102, ukazoval kvůli zjednodušení jen výsledné spoje logických elementů. Ve skutečnosti se realizují v propojovacím boxu, *Connection Box*, který se nachází u každého logického elementu a má rozličnou strukturu dle výrobce. Jeho prvky se opět řídí paměťovými buňkami nastaveným při naprogramování FPGA na naše zapojení.

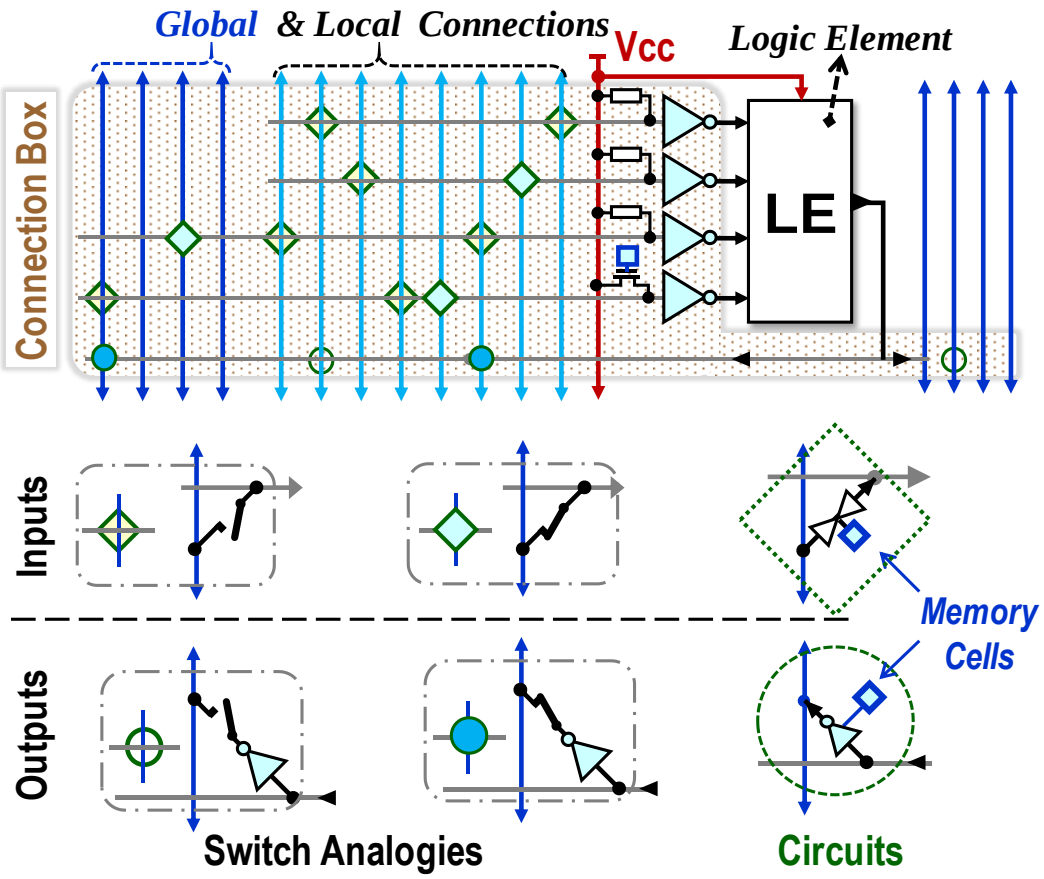
Výstupy se mohou třeba realizovat třístavovými invertory, o nichž víme, že jsou rychlé (str. 60 a 64). Vstupy se zase připojí přes *transmission gates*, které mají vysoké impedance ve stavu odpojeno, jimiž nezatěžují vodič. Jelikož v sepnutém stavu mají odpory o hodnotách kiloohmů, na nichž dochází k úbytkům napětí, a tak se hodí při příjmu signálu obnovit plné úrovně '0' a '1'. Používají se různé varianty.

Následující obrázek ukazuje jedno možné řešení s invertory a vstupními *pull-up* odpory (*český termín nenalezen*) připojenými na rozvod napájecího napětí Vcc. Jimi se současně zaručí úrovně logických '1' na vstupech i v případě, že všechny propojky zůstaly ve stavu odpojeno.

Velikost *pull-up* odporu lze zvolit i v řádu megaohmů[31]. Na vodič jsme tak poslali výstup přes invertor a nyní se přijme přes další invertor, takže se obnoví původní stav signálu.

[30] Další typy propojovací matice jsou třeba v https://www.researchgate.net/publication/221224917.

[31] Konkrétní hodnota pull-up odporu závisí na parametrech technologie CMOS. Realizuje se často NMOS transistorem technologie *depletion*, viz kapitola 4.2 na str. 54.

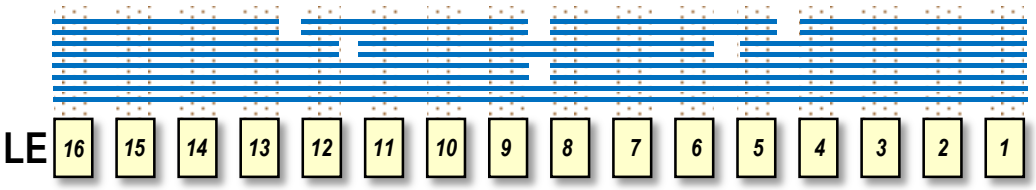


Obrázek 105 - Příklad jednoho možného řešení propojovacího pole

Dolní invertor u LE naznačuje i alternativní řešení, a to konfigurovatelné připojení vstupu hradla na Vcc. Paměťová buňka ovládá NMOS transistor, který zde plně stačí, neboť jím prochází proud jen jedním směrem, na rozdíl od obousměrných *transmission gates*.

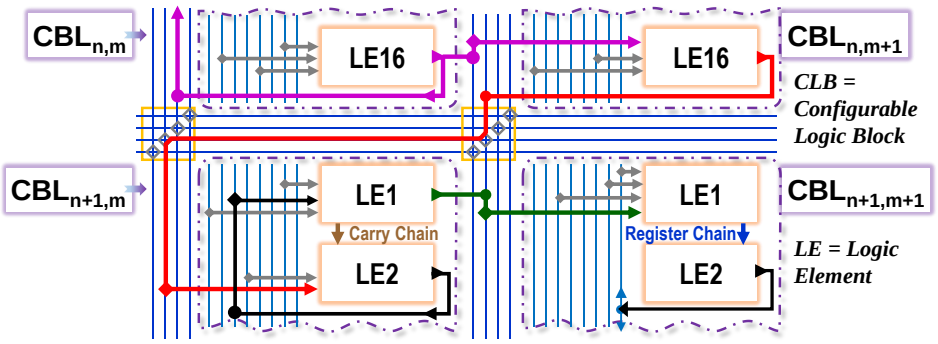
Polohy přípojných bodů většinou nesledují pravidelný vzor. Například v lokálním kanále každý ze šestnácti logických elementů jednoho konfigurovatelného logického bloku, CLB, je může mít jinak rozhozené. Jejich rozmístění volí výrobci na základě svých analýz výsledků propojovacích algoritmů tak, aby vytvořili kombinaci vedoucí na statisticky nejlepší výsledky.

I vodiče v lokálním spojovacím kanále uvnitř CLB se dělí na různé segmenty, aby jeden výstup nevyčerpal propojení všem logickým elementům. V některých FPGA bývají mezi nimi i propojovací matice, což opět zvyšuje variabilitu jejich možného využití.



Obrázek 106 - Příklad možné segmentace lokálních vodičů

Jakmile návrhové prostředí našlo optimální zapojení našeho obvodu, vezme jemu známé uspořádání našeho konkrétního typu FPGA a začne řešit rozmístění výsledku do logických elementů a vzájemné spoje mezi nimi, *placement and routing* (cz: rozmístění a propojení?). Úloha zabírá nejdelší čas rostoucí se složitostí obvodu. Hledá se nejen vhodné rozmístění logických elementů a spojky mezi jejich vstupy a výstupy, ale hlídají se i zátěže vodičů.



Obrázek 107 - Příklad propojení logických elementů v FPGA

Heuristické algoritmy překladače poskytnou přijatelný výsledek v polynomiálním čase, ale ne vždy. Naše zkušenosti ukazují, že se průměrně vyčerpá od 70 do 90 % dostupných logických elementů v FPGA, pak se již rozmístění nepodaří, a nejčastějším důvodem bývá právě vyčerpání propojek. Konkrétní procento závisí nejen na složitosti obvodu, ale také na jeho vhodném popisu. Neoptimální návrhy snižují využitelnost FPGA i pod 60 %.

5.5.7 Srovnání Cyclone II a Cyclone IV

V předchozím popisu jsme se zaměřili na menší Cyclone II v našich starších vývojových desek DE2. Nově se v předmětu LSP používají pokročilejší desky VEEK-MT2 se Cyclone IV, který obsahuje podobné prvky, pouze ve větším počtu. Uvedeme srovnání obou FPGA.

Třída obvodu	**Cyclone II**	**Cyclone IV**
Typ	**EP2C35F672**	**EP4CE115F29**
Technologie	90 nm	60 nm
Logické elementy	**33216** rozložených do 2076 CLBs	**114480** rozložených do 7155 CLBs
Paměťové bloky	483840 bitů (**105** M4K bloků)	3981312 bitů (**432** M9K bloků)
DSP násobičky	**35** (18x18), nebo 70 (9x9)	**266** (18x18), nebo 532 (9x9)
User I/O	**475**	**528**
Cena (rok 2022)[32]	~ $ 20	~ $ 65

Tabulka 5 - Srovnání FPGA Cyclone II s Cyclone IV

V obou typech jsou 4 digitální PLL (fázové závěsy) k násobení frekvencí a DSP hardwarové násobičky 18x18 bitů, které lze individuálně nakonfigurovat na dvě nezávislé násobičky 9x9 bitů, kterých tak může být až dvojnásobek.

Paměťové bity se v Cyclone II alokují po M4K blocích. Každý M4K obsahuje 4096 bitů + 512 paritních použitelných jen u konfigurace na výstup o šířce bytu. U Cyclone IV se přidělují po M9K blocích, každý M9K má 8192 bitů + 1024 paritních pro bytovou konfiguraci.

[32] Při nákupu vyššího množství se zpravidla poskytuje sleva, třeba až 20 % při odběru deseti tisíc kusů.

5.6 Konfigurační paměťové prvky v FPGA

Klopné obvody v logických elementech se budují z členů na bázi CMOS transistorů, stejně tak se realizují i paměťové bloky, aby se do nich svižněji uložila informace.

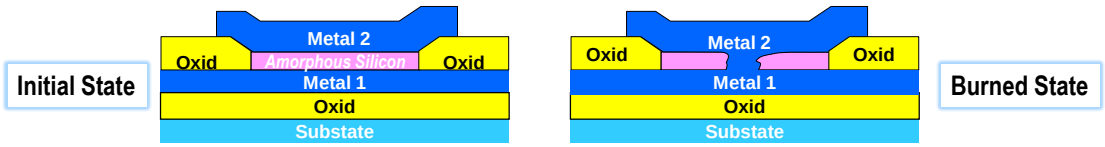
V téhle části se budeme zabývat paměťovými buňkami, jimiž se FPGA konfiguruje při svém programování na nové zapojení. Stručně přiblížíme vlastnosti jejich tří nejpoužívanějších typů. Podrobnější rozbor by si žádal více prostoru a necháme ho specializovaným publikacím.

Nejvíce se (podle USA údajů za rok 2022) prodávají FPGA, v nichž se konfigurovatelné paměťové buňky tvoří CMOS transistory, a tak se tak podobají SRAM. Synchronně se nahrávají během programování FPGA, ale poté mají trvale dostupnou svou hodnotu. Konfigurace FPGA je rychlá a bez omezení počtu opakování.

I naše FPGA řady Cyclone II a Cyclone IV jsou na bázi SRAM.

Obsah buněk se sice ztrácí po vypnutí napájení, ale běžně se k FPGA přidává externí obvod s permanentní pamětí. V té se uloží jeho počáteční konfigurace, které se po zapnutí napájení automaticky nahraje do FPGA, čímž se v něm obnoví zapojení.

V četnosti užití se druhém místě nacházejí FPGA typy s *antifuse* paměťovými prvky nazvanými podle opačného chování vůči pojistkám. Ve výchozím stavu nevedou, napěťovým pulzem se nevratně prorazí. Potřebují nutně externí programovací zařízení, nelze je konfigurovat zapájené na plošném spoji, a jde to pomalu. Udávají se desítky minut na obvod.



Obrázek 108 - Antifuse

Antifuse mají ze své podstaty vysokou odolnost proti radiaci a dlouhodobou stabilitu. Nelze však ve výrobě otestovat jejich funkčnost. Na ni se přijde až při konfiguraci. V katalozích se udává statistika, že úspěšnost je lepší než 95 %, tzv. *programming yield*. Jinými slovy při konfiguraci 100 kusů bude chybných méně než 5. A výrobci si kladou podmínku, že se nepřijímají reklamace. Zákazníci musí s tím počítat a koupit si víc kusů.

Jiným oblíbeným typem jsou opakovaně konfigurovatelné FPGA s ***flash*** paměťovými prvky, tedy stejnými jako v SSD discích. Nabízejí jejich vlastnosti, a to nejen udržení svého obsahu po vypnutí napájení, ale i nízkou klidovou spotřebu energie v zapnutém stavu. Programují se pomaleji než SRAM FPGA, ale mnohem rychleji než *antifuse*.

FPGA obvody užívající SRAM či Flash ke své konfiguraci jsou citlivější na radiaci, ale i tak se nasazují i v kosmických aplikacích. Pro ně se vyrábějí typy *radiation-hardened*, respektive *radiation tolerant*, vybavené i stíněním. V případě potřeby lze do nich dálkově nahrát novou konfiguraci obvodu.

Jsou ještě jiné levné možnosti?

Někteří výrobci nabízejí univerzální polotovary, třeba *gate-arrays*, které se někdy také označují názvem ULAs, *Uncommitted logic arrays*. Jiným jejich typem jsou i *standard cells*, které obsahují i sbírku užitečných malých obvodů, jako čítače, posuvné registry a podobně.

Lze z nich vytvořit zákaznické monolitické integrované obvody. Zakoupíme si od výrobce několik *wafers* (cz:wafery?) s předpřipravenými *dies* (cz:ploškami obvodů?) a necháme si je dotvořit napařením vrstev propojek podle našeho popisu obvodu odladěného na FPGA.

Celková cena bude sice nižší než v případě vývoje celého integrovaného obvodu, ale nikoli zanedbatelná. Vyplatí se jen u sérií začínajících někde kolem dvou tisíc kusů. Malosériové produkce se dnes realizují na FPGA.

6 Aritmetické kombinační obvody

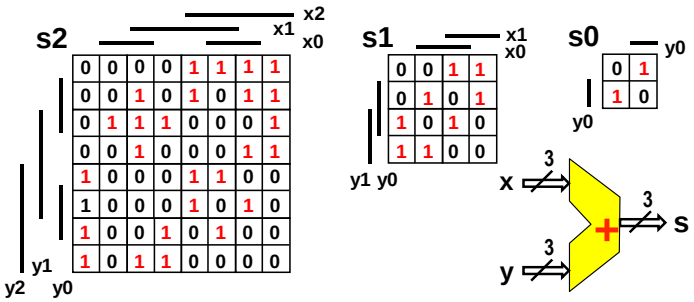
Komparátory, sčítačky a násobičky nebudeme sami zapojovat, jen specifikujeme žádanou aritmetickou operaci v návrhovém prostředí, které se již plně postará o realizaci. Hodí se znát aspoň trochu vlastnosti jejich zapojení. V kombinačním aritmetickém obvodu nelze něco sdílet, například volat funkci, či běhat v cyklu.

Každá operace se v obvodu převádí vložením dalšího výpočetního bloku, a tak se výpočty provádějí technikou *inline expansion* (cz: přímá expanze kódu?). Třeba for cyklus se v obvodu nahrazuje opakovaným vkládáním svého těla. Musíme každou operaci držet optimální, neboť výsledek bude platný až po průchodu změn skrz všechny. A k lepšímu výsledku pomohou i zdánlivé drobnosti. Například operace s čísly, která mají nějakého svého dělitele tvaru 2^N, se vykonají rychleji. Aritmetika se v obvodu zapojují až od vyšších bitů. Naproti tomu procesor vždy sčítá od nejnižšího bitu, a tak přičte konstantou 79 stejně tak rychle jako 80.

6.1 Sčítání a odčítání

Můžeme sice popsat celou vícebitovou sčítačku logickými funkcemi, avšak ty nedokážeme efektivně minimalizovat. Důvod naznačí Karnaughovy mapy sčítačky, v nichž se vyskytují malé skupinky jak logických '0', tak logických '1'. Potřebovali bychom hodně implikantů.

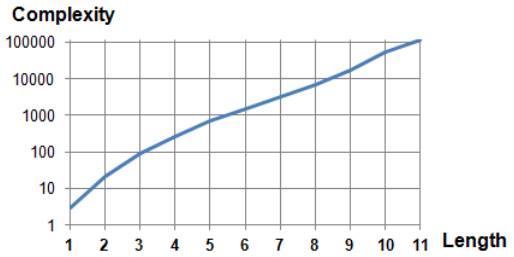
Fakt demonstruje obrázek vlevo na příkladu sčítačky dvou tříbitových binárních čísel x = |x2|x1|x0| a y=|y2|y1|y0|, kde x0 a y0 označují jejich nejnižší bity. Karnaughovy mapy udávají bity jejich výsledného součtu s=|s2|s1|s0|.



Složitost přímé implementace sčítačky roste exponenciálně, což ukazuje graf dole, zjištěný minimalizačním algoritmem Boom, zmíněným v kapitole 3.5, str. 54.

Vodorovná ose udává bitovou délku jednolité sčítačky a svislá pak její složitost v počtu členů ve všech logických funkcí.

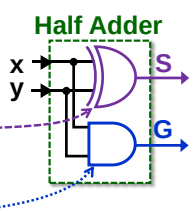
Experiment se zastavil u jedenáctibitové sčítačky, jelikož i doba čekání na výsledek rostla exponenciálně stejně jako složitost logické funkce.



Sčítačka se musí nutně dekomponovat na menší dílčí elementy, k čemuž se nabízí jednoduchá KM nejnižšího bitu s0, která se minimalizuje na s0 = x0 xor y0.

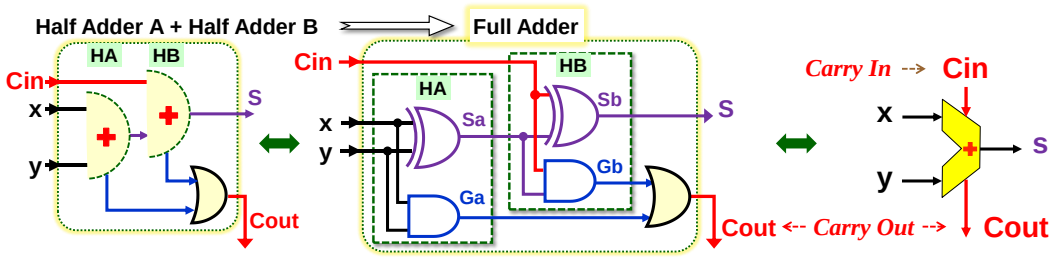
Obvod jednobitového součtu bez uvažování přenosu z nižšího řádu, se nazývá poloviční sčítačkou, *half adder*. Výstup jejího součtu bude S=x xor y. Přenos do vyššího řádu se generuje jen při x='1' a y='1', kdy se x+y dekadicky rovná 2, binárně "10" = | G | S |, tedy G=x AND y.

$(x+y)_{dec}$	x	y	$(x+y)_{bin}$ G	S
0	0	0	0	0
1	0	1	0	1
1	1	0	0	1
2	1	1	1	0



Obrázek 109 - Poloviční sčítačka

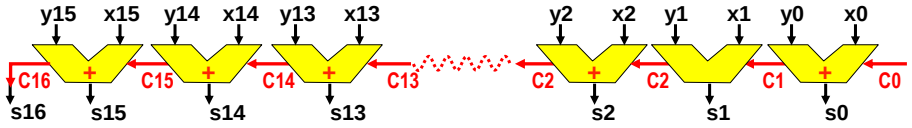
Poloviční sčítačka je stavebním prvkem dalších sčítaček, především úplné jednobitový sčítačky, *Full Adder*. Vznikne tak, že spojíme dvě poloviční sčítačky. První sečte vstupy x a y a druhá k jejímu výsledku přičte přenos z nižšího řádu Cin, *Carry In*.



Obrázek 110 - Úplná sčítačka, *Full Adder*

Výstupy Ga a Gb, generování přenosů obou polovičních sčítaček, spojíme OR operací, neboť nikdy nejsou oba v logických '1'. Gb se může objevit jen při Sa='1' a tehdy je Ga='0'.

Vytvořili jsme úplnou sčítačku. Spojí-li se jich několik za sebou, pak Cout u každé sčítačky vede na Cin vyššího bitu. Přenosy se nazývají dle stupně, do něhož se posílají.



Obrázek 111 - Sčítačka 16 bitů typu RCA - *Ripple Carry Adder*

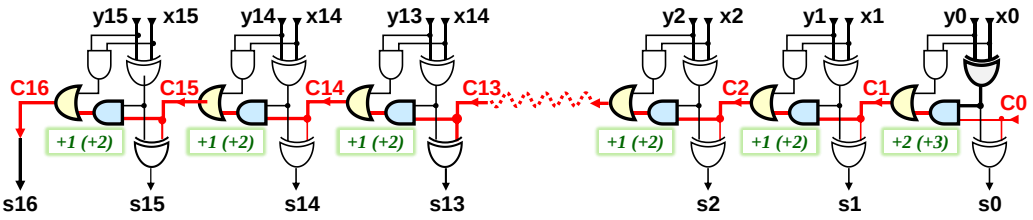
Vznikla sčítačka s přenosem, která se i v českých publikacích označuje zavedeným termínem RCA, od jejího anglického názvu *Ripple Carry Adder*. Její krajní vstup C0 se rovná '0', pracuje-li samostatně. Každá změna nějakého přenosu Ci se šíří řadou k vyšším bitům ve stylu vln[33]. Výsledek bude platný až po ustálení všech přenosů. Nejdéle potrvá například součet čísel x=2^{16}-1 a y=1, kdy přenos poběží od s0 až k s15, a výsledkem bude 0 a s16=C16='1'.

Poslední C16 bude i nejvyšším bitem součtu s16, neboť suma dvou 16bitových čísel dává až 17bitový výsledek. Sčítáme-li dvě čísla bez znaménka a žádáme-li jen 16bitový výsledek, pak C16='1' je příznakem přetečení. Náš součet se již nevešel do 16bitového limitu.

Jak rychlá bude naše sčítačka s přenosem? Schémata dnes představují lidem srozumitelný popis žádané funkce obvodu, nikoli přesnou interní strukturu jeho zapojení. Při fyzické realizaci obvodu se využívají zkratkovité konstrukce, viz třeba struktura XOR hradla nastíněná v kapitole 4.6 na str. 64, kde se také nebudovalo otrocky dle jeho logické rovnice.

[33] Analogické *ripple* šíření se v ekonomice nazývá dominový či lavinový efekt a ve fyzice zas řetězová reakce.

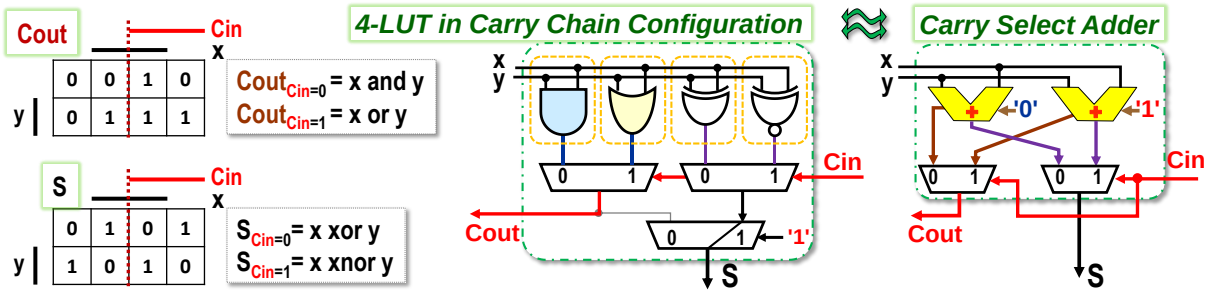
Rozkreslíme-li RCA z obrázku nahoře. V ní vyznačíme kritickou cestu za předpokladu, že se najednou změnily všechny její vstupy x, y a C0.



Obrázek 112 - Kritická cesta v RCA

Celková doba ustálení součtu závisí na obvodové realizaci. V kapitole 4.4.1 na str. 63 jsme si vytvořili jsme speciální hradlo AND-OR, přes nějž se přenos bude šířit se zpožděním jediného logického členu. Když ho využijeme, C1 se zpozdí o dva logické členy. Mezitím se ale nastaví ostatní vstupní poloviční sčítačky. Ve vyšších bitech se pak přidává jen průchod skrz AND-OR hradlo, takže 16 bitová RCA se ustálí nejdéle za dobu zpoždění 2+15*1=17 prvků.

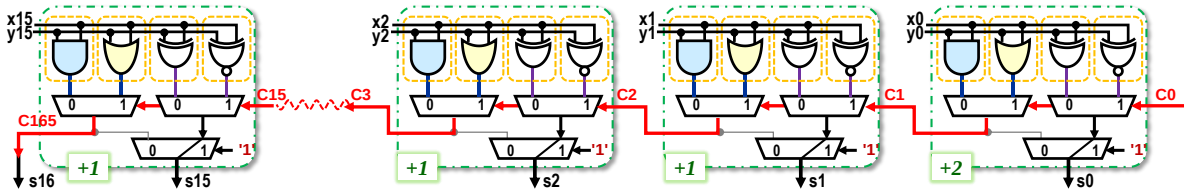
Podíváme se ještě, jak se úplná sčítačka implementuje v konfiguraci LUT u FPGA na šíření přenosu, v níž máme dvě 3-LUT, viz Obrázek 98 na str. 88. Rozklad multiplexory v LUT provedeme Shannonovu expanzí. Vytvoříme si kofaktory dle vstupu Cin, čímž Karnaughovy mapy úplné sčítačky dekomponujeme na dva dvouvstupové multiplexory přepínané Cin.



Obrázek 113 - Úplná sčítačka ve 4-LUT jako *Carry Select Adder*

Úplná sčítačka se zde duplikovala na dvě se stejnými vstupy, ale jedna z nich má svůj Cin pevně na '0', zatímco druhá na '1'. Vstup Cin jen přepíná, z jaké z nich se vybere výstup. Podobné zapojení je ve skutečnosti CSelA, *Carry Select Adder*, (český termín nenalezen).

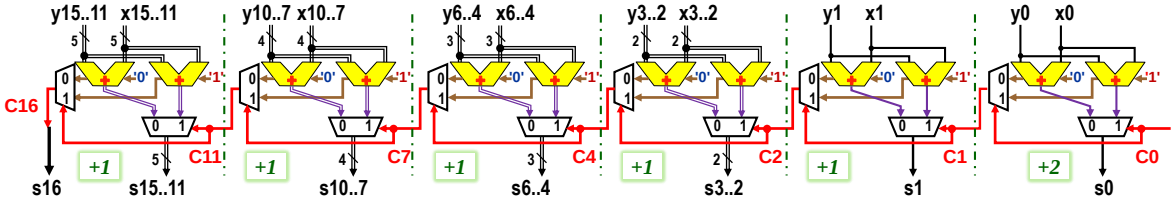
V FPGA budou úplné sčítačky propojené stylem RCA, ovšem jejich přenosy se přes multiplexory expresně šíří se dle obrázku dole. Každý vstup přenosu Ci přepíná naráz oba multiplexory obou 3-LUT, na něž se teď nakonfigurovala 4-LUT.



Obrázek 114 - FPGA implementace 16bitové sčítačky *Ripple Carry Adder*

Změní-li se najednou x, y a C0, pak se u sčítačky nejnižšího bitu s0 dostane přenos na výstup C1 přes tři řady multiplexorů. Ty se však přepnuly naráz a vybrané hodnoty vyšší řady procházejí už jen přes *transmission gates*, tedy pouze přes odpory. Celkové zpoždění u bitu 0 lze tedy rovněž pokládat za 2 členy. U dalších bitů se přidá leda doba zpoždění na společného přepnutí obou multiplexorů řady C, což znamená jen jeden další člen. Sčítačka 16 bitů bude mít v FPGA zpoždění 17 členů, bude tedy stejně optimální jako realizovaná přímo v CMOS.

I tak nebude sčítačka RCA příliš rychlá. Z úplných sčítaček se dá sice vybudovat i delší *Carry Select Adder*, využijeme-li multiplexory k přepínání RCA sčítaček postupně rostoucích bitových délek, neboť následující měly již delší čas na ustálení svých dílčích výstupů.

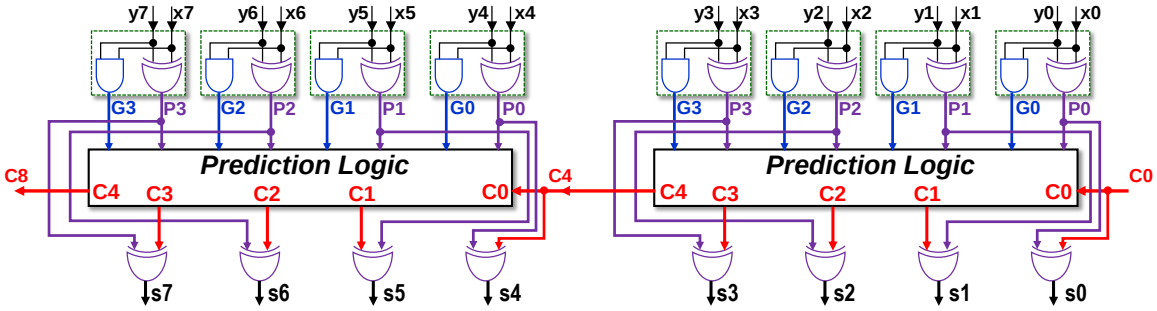


Obrázek 115 - 16bitový CSelA - *Carry Select Adder*

Zpoždění u 16 bitové sčítačky CSelA činí jen 7 členů, 2 u prvního stupně a po 1 u následujících bloků. 32bitová CSelA by si žádala jen přidání dalších tří bloků, a tak by její zpoždění činilo jenom 11 členů, zhruba třetinové vůči FPGA implementaci 32bitové RCA.

CSelA není efektivní ani počtem použitých CMOS transistorů, ani odběrem energie. Obsahuje dvě RCA sčítačky a k nim ještě mnoho sběrnicových multiplexorů.

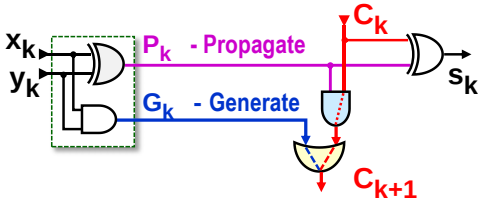
Výhodný způsob řešení nabízí často užívaná sčítačka s predikcí přenosu, běžně zkracovaná na CLA, *Carry Lookahead Adder*. Využívá též poloviční sčítačky, u nichž se tady výstup jejich polovičních součtů nazývá Propage. V obrázku jsou to P0 až P3. Z jejich výstupů se pak predikují přenosy C1 až C4 finálních součtových hradel XOR.



Obrázek 116 - Prvních osm bitů sčítačky CLA se 4bitovou predikcí

Použité názvy výstupů naznačuje obrázek součtu bitů s indexem k v RCA sčítačce.

- Přenos C_k z nižšího řádu projde, *propagates*, na výstup C_{k+1} jedině tehdy, když je P_k='1'.
- Zato G_k='1' vždy generuje výstup C_{i+1} = '1'.



Zapojení vede na vztah predikce, v němž ke zkrácení použijeme notaci **.** a **+** pro AND a OR.

$$C_{k+1} = G_k + P_k.C_k; \qquad \text{(Ca1)}$$

Slovy můžeme (Ca1) vyjádřit, že přenos se pošle do vyššího řádu jen tehdy, pokud ho úplná sčítačka buď sama generuje, nebo povolí průchod přenosu z nižšího řádu. Rozepíšeme si logické funkce pro přenosy C1 až C4 (U delších CLA se Cj pro j>4 napíší analogicky):

$$C1 = G0+P0.C0;\ C2=G1+P1.C1;\ C3=G2+P2.C2;\ C4=G3+P3.C3;\ \ldots \qquad \text{(Ca2)}$$

RCA počítá logické funkce iterativně s využitím výsledků nižších bitů, ale právě ty musíme dosadit do vztahů, chceme-li ji urychlit, a logické funkce rozvést až na mintermy:

C1 = G0+P0.C0;

C2 = G1+P1.(G0+P0.C0) =G1+P1. G0+P1.P0.C0;

C3 = G2+P2.(G1+P1.(G0+P0.C0))=G2+P2.G1+P2.P1. G0+P2.P1.P0.C0;

C4 = G3+P3.(G2+P2.(G1+P1.(G0+P0.C0)))

= G3+P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.C0; (Ca3)

Slovy popíšeme vztahy (Ca3) tak, že přenos vygenerovaný v nějakém nižším stupni se šíří přes vyšší řády, dokud všechny mají své výstupy Propage v '1'. Výsledné C_k pak dostaneme logickým OR všech vlivů.

Počet členů v jednotlivých rovnicích C_k, kde k>0 je predikovaný bit, roste s (k+2)(k+1)/2, tedy se součtem aritmetické řady. Používají se i CLA až s osmibitovou predikcí, ale nejčastěji se predikuje pouze přes 4 bity. CLA sčítačku se 4bitovou predikcí lze, na úrovni monolitických integrovaných obvodů, zapojit s počtem CMOS transistorů o jednotky procent menším než RCA sčítačku a se spotřebou energie jen o padesát procent vyšší oproti ní[34].

Rychlost CLA sčítačky opět závisí na jejím skutečném zapojení. Realizace predikce přesně podle vztahů (Ca3) není výhodná, protože výraz C4 má AND implikant vedoucí na 5vstupové AND hradlo. Z části o CMOS víme, že bude pomalejší. V každém bloku CLA by se změna jeho vstupu C0 šířila na C4 se zdržením až 3 logických členů, a ne jen dvou. Oproti RCA by se CLA urychlila pouze o čtvrtinu, jak dokazuje práce zmíněná v poznámce na předchozí straně.

Existuje řada triků jak CLA zapojit lépe. Můžeme třeba využít naše hradlo AND-OR stejně jako u RCA. První čtyřbitový blok CLA bude opět pracovat se zpožděním zhruba 3 členů, neboť predikuje až po výsledcích dodaných vstupními polovičními sčítačkami.

Následující 4bitové CLA využijí rozklad predikce. V době čekání na vlnu přenosu si předpřipraví mezivýsledky, třeba u kritické C4 půjde o členy nezávislé na jejich C0 vstupu:

C4g = G3+P3.G2 + P3.P2.G1 + P3.P2.P1.G0; **C4p** = P3.P2.P1.P0; (Ca4)

Až dorazí vlna přenosu k jejich C0, rychle pošlou výstup C4 přes AND-OR hradlo:

C4 = **C4g** + **C4p**.C0; (Ca5)

Vyšší čtveřice CLA pak přidávají též po jednom zpoždění svých AND-OR hradel, kromě poslední, u níž se zahrne i její výstupní XOR. Vylepšená 16bitová CLA může tak ustálit výsledek se zpožděním 3+1+1+2=7 členů, tedy srovnatelně s CSelA, a 2.4x rychleji než RCA.

Existují ještě svižnější sčítačky? Ano, nazývají se prefixové sčítačky, častěji ale označované za **prefixové paralelní sčítačky**, *Prefix Parallel adders*, **PPAs**, což již přesněji specifikuje jejich funkci. Slovo „prefix“ se u nich totiž vztahuje k matematické notaci použité autory.

PPA využívá vztahy (Ca3), ale počítá je slučováním párů *Generate* a *Propagate* na paralelní binární stromové struktuře, v jejíž uzlech se nacházejí dvojice jednoduchých logických funkcí, a to AND a AND-OR, jejichž pár se nazývá operátorem. Zdvojnásobí-li se délka PPA, do predikce se přidá jen jedna další vrstva do stromu. Zpoždění se tak zvýší pouze o jediný člen.

PPA predikuje přes celou délku sčítačky, třeba i přes 128 bitů, kdy najednou spočte přenosy od C1 až po C128. Má však jak značný odběr ze zdroje, tak složité propojení. Používá hlavně ve velkých procesorech na dlouhé sčítačky. U kratších tolik nevynikne.

[34] R. Uma, Vidya Vijayan, M. Mohanapriya, & Sharon Paul. (2018). Area, Delay and Power Comparison of Adder Topologies. https://doi.org/10.5281/zenodo.1410195

Za nejrychlejší PPA sčítačku se pokládá KSA, *Kogge-Stone Adder*, která má úplný predikční strom, ale je též nejnáročnější ze všech na příkon i velikost. Další PPA implementace, je jich dost, se snaží tohle napravit a redukovat strom, aniž by příliš klesla rychlost sčítání.

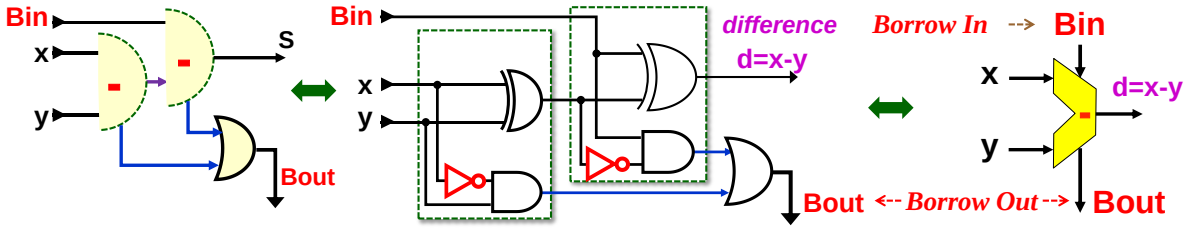
Princip PPA sice představuje ukázku perfektní implementace algoritmu na úrovni CMOS, ale necháme ho odborným publikacím. Její strukturu jen mírně přiblížíme na str. 104.

Proč logické elementy FPGA používají pomalé RCA sčítačky? Existuje k tomu víc důvodů:

- RCA mají nejnižší odběr energie ze všech možných sčítaček, dle údajů v různé literatuře.
- Lineární uspořádání RCA se velmi snadno propojuje.
- Šíření přenosu, *Carry Chain*, na něž se přepnou LUT v logických elementech FPGA, se s výhodou využije i jinde, třeba v komparátorech. Predikční logika by sloužila jen CLA.
- V zapojení se často pracuje hlavně s kratšími čísly, které RCA zvládá za přijatelný čas.
- **Obvod se nejvíce zrychlí** vhodným rozkladem jeho funkce na paralelní struktury. PPA sčítačky nepoužívají lepší hradla, jen jejich výhodnější propojení.
- Výkonnější typy FPGA aplikují urychlení jim přirozenější. Mají variabilní LUT s více vstupy, třeba až s osmi, např. v Intel FPGA Stratix IV. Lze je nakonfigurovat i na dva výstupy, a tak jeden logický element dokáže realizovat dvoubitovou sčítačku. Přenosy se pak šíří po skocích dvou bitů, tedy dvojnásobně rychleji. Další logické elementy mohou přidat predikční logiku vyšších bitů. Jejich sčítačky pak pracují srovnatelně s CLA.

6.1.1 Odčítání

Úplnou odčítačku, *full subtractor*, můžeme zapojit ze dvou polovičních odčítaček.



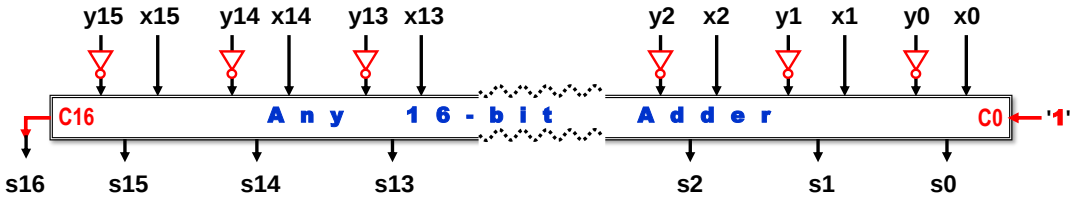
Obrázek 117 - Úplná odčítačka složená za dvou polovičních

Oproti dříve probrané úplné sčítačce, Obrázek 110 na str. 98, vidíme jen nevelké změny:

- Výsledek rozdílu se nyní nazývá diference, a tak má symbol d.
- Přenos odčítačky se nazývá *borrow (cz:vypůjčka?)*, neboť při operaci '0'-'1' se musí od vyššího řádu vypůjčit bit. Jde sice o přesný obvodový termín, ale v literatuře se často nerozlišuje mezi podtečením a přetečením. Zejména aritmetické jednotky procesorů používají označení *carry* v obou případech.
- V přenosech polovičních odčítaček přibyly jen invertory v signálech menšenců, tedy vstupu, od něhož odečítáme. Je-li ten rovný '0', pak menšitel '1' vyvolá podtečení.

Odčítačka se rovněž snadno rozloží do LUT v její *Carry Chain* konfiguraci, a tak pracuje stejně rychle jako sčítačka.

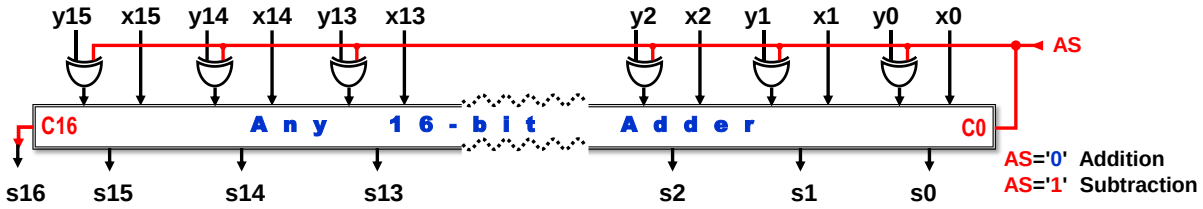
V procesorech se odčítání častěji nahrazuje přičítáním záporného čísla. Odečítaný vstup y se na něj převede negací všech jeho bitů (první doplněk) a nastaví se C0 na '1', tedy přičtení +1. Vytvoříme tím -y ve druhém doplňku, *two's complement*, což je dnes nejrozšířenější počítačový formát *signed*, viz Binární prerekvizita.



Obrázek 118 - Realizace x-y v 16bitové aritmetice

Připomeneme si část z prerekvizity. Sčítání čísel *signed* a *unsigned* se provádí, z hlediska fyzické realizace, úplně stejně a jen se jinak vyhodnocuje přetečení výsledku. U *unsigned* ho signalizuje *carry* nejvyššího bitu. U *signed* se mluví o *overflow*, které testuje validitu nejvyšších bitů (znamének) sčítanců a součtu. Nastaví se, když výsledek má nesmyslné znaménko, jako třeba záporný součet dvou kladných čísel, apod.

Mnohdy se v procesorech hodí přepínání mezi sčítáním a odčítáním, třeba u celočíselného dělení. Invertory se pak nahradí hradly XOR, o nichž víme z kapitoly 2.3.1 na str. 23, že se pro jeden svůj vstup chovají jako hradlo přepínatelné mezi chováním *buffer* a invertor.



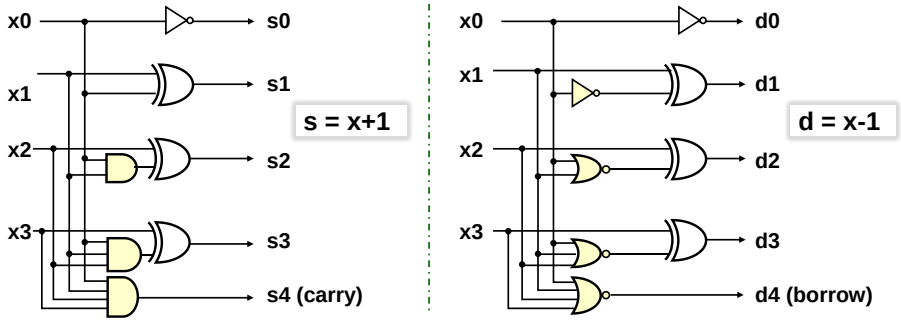
Obrázek 119 - Univerzální sčítačka a odčítačka

6.1.2 Sčítání a odčítání konstant

Zde si dovolíme napřed připomenout několik zřejmých faktů:

- Konstanta C, která je beze zbytku dělitelná nějakou mocninou 2^M, má svých M spodních bitů nulových, a tak se v operacích x+C, respektive x-C, bity 0 až M-1 vstupu x vedou na přímo výstup. Sčítá se, odčítá se, jen horní část x, čímž se operace urychlí.
- Návrhová prostředí minimalizují i sčítačky či odčítačky dle bitů připojené konstanty.

U přičítání či odčítání mocniny 2 se zapojení výrazně redukuje. Ukážeme si ho na 1, tedy 2^0.



Obrázek 120 - Přičtení a odečtení čísla 1 u 4bitového čísla

Slovy lze zapojení vyjádřit dvojicí pravidel:

- nejnižší bit se invertuje vždy, jak při přičtení 1, tak při odečtení 1;
- při přičítání jedničky se bit invertuje, pomocí XOR coby řízeného prvku *buffer*/NOT, pokud se všechny bity nižších řádů rovnají '1', což vyplývá z vlastnosti řady binárních čísel:

  0000 0001 0010 0011 0100 0101 0110 0111 1000 0111... atd.

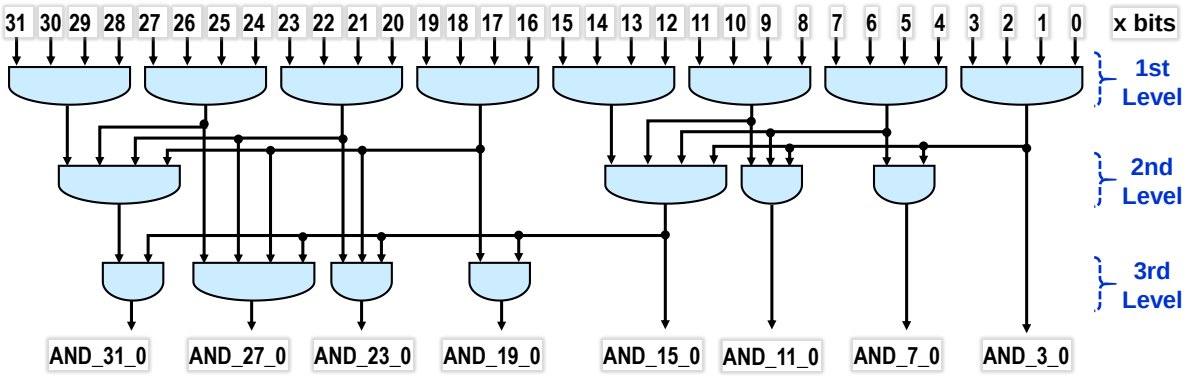
- Při odečtení 1 se naopak testuje, zda dolní bity jsou všechny v '0'.

  0000 1111 1110 1101 1100 1011 1010 1001 1000 0111... atd.

Pokud by číslo x mělo více bitů, pak začne narůstat délka AND/NOR hradel. Jejich nárůst omezíme paralelním výpočtem vztahů typu

$$AND\_m\_0 = x_m \text{ and } x_{m-1} \text{ and } \ldots \text{ and } x_1 \text{ and } x_0$$

Rozložíme je pomocí teorému o asociativitě (str. 16) na stromovou strukturu, v níž zavedeme limit na užití AND hradel s nejvýše 4 vstupy. Ke zvýšení přehlednosti nenakreslíme celý strom, ale jen jeho čtvrtinu. Ostatní AND_m_0 by se stanovily analogicky. U stromu odčítačky by se jen používala hradla OR místo AND a invertor by se dával až za jejich výsledek, jak bude naznačeno na další stránce.



Obrázek 121 - Výpočet AND_i_0 na paralelní stromové struktuře

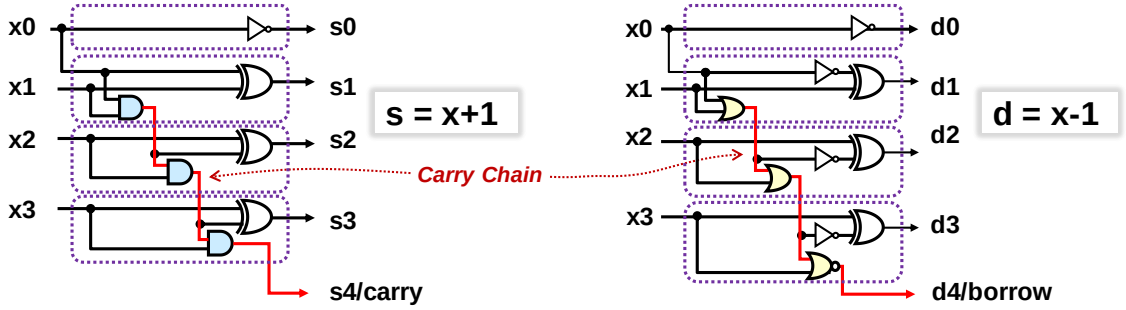
Vidíme, že 3 úrovně hradel paralelně vyhodnotí všechny členy AND_m_0 potřebné pro 32bitovou sčítačku +1. S trojicí úrovní by se vystačilo i u délky 64bitů, až 128bitová čísla by potřebovala přidat další vrstvu hradel AND.

Naše sčítačka přičte +1 za poloviční čas než nejrychlejší známá sčítačka KSA, jejíž paralelně prefixová podstata též využívá slučovací strom. V něm ale kombinuje predikce složitějšími výrazy, které se dají spojovat jen po dvojicích. 32bitová KSA tak potřebuje 5 vrstev slučovacího stromu a k nim ještě 1 na přípravnou a 2 na zakončení. Naší sčítačce či odčítačce 1 stačí pouhé 3 vrstvy hradel a jedna konečná s XOR hradly.

Programovací jazyk C získal své proslavené operace ++ a -- právě kvůli expresní činnosti sčítačky a odčítačky 1, které přinášely významné urychlení zejména v časech, kdy procesory běžely na megahertzových frekvencích. Přičtení a odečtení 1 se v nich realizovalo zvláštními obvody. Strojové kódy kvůli nim zahrnovaly bytové instrukce *increment* a *decrement*. U procesorů Intel šlo kódy asembleru INC a DEC, na něž se operace ++ a -- překládaly.

S rozvojem proudového zpracování instrukcí, *pipelining*, upadal význam obou instrukcí. Překladače programovacích jazyků je dnes zpravidla nahrazují přičtením +1 či -1, které novější výkonné sčítačky procesorů sice provedou za nepatrně delší dobu, ale nastaví i další stavové bity výsledku. Intel procesory, běží-li v 64bitovém módu, už nemají INC a DEC — jejich krátké kódy se přidělily jiným instrukcím, dnes důležitějším pro zpracování programu[35].

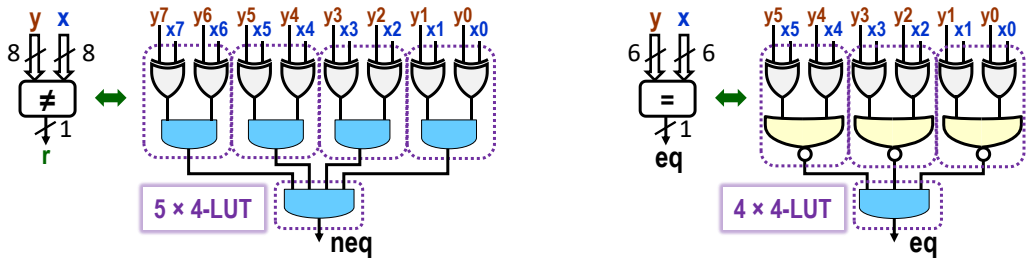
Na úrovni FPGA se použitím konstanty u sčítání a odčítání ušetří především propojky. Přenos se bude opět šířit přes *Carry Chain*. U +1 /- 1 operací vypadne ze sčítačky/odčítačky jediný člen u bitu x0, viz obrázek dole, což znamená jen mírné urychlení. OR hradla se řetězí, negace je až za nimi. Výkonnější FPGA s variabilními LUT, které dovolují dva bitové výstupy, pak i opět urychlí běh po dvojicích bitů.



Obrázek 122 - Realizace 4bitové sčítačky a odčítačky 1 v logických elementech FPGA

6.2 Komparátory

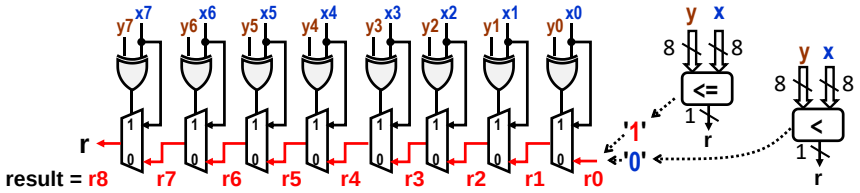
Test rovnost dvou čísel se zapojí efektivně paralelně pracujícími bitovými porovnání, např. hradel XOR, která vracejí '1' při různosti bitů. Pokud výsledky složíme hradlem AND, dostaneme komparátor nerovnosti, při spojení hradlem NOR pak rovnosti. Obrázek dole naznačuje rozložení operace na FPGA logických elementech, které obsahují čtyřvstupové 4-LUT.



Obrázek 123 - Komparátor nerovnosti a rovnosti na 4-LUT

[35] INC a DEC kódy se v x64 asembleru přidělily REX.R prefixům, jimiž se specifikuje, že následující instrukce použije buď přístup do rozšířené sady registrů či 64bitovou modifikaci starší instrukce z i386 podmnožiny.

Obecné porovnání lze realizovat kaskádou Mux 2:1, v níž každý stupeň přijímá informaci, zda všechny nižší bity splnily podmínku, a nahoru posílá svůj výsledek.



Obrázek 124 - Princip komparátoru y<=x a y<x

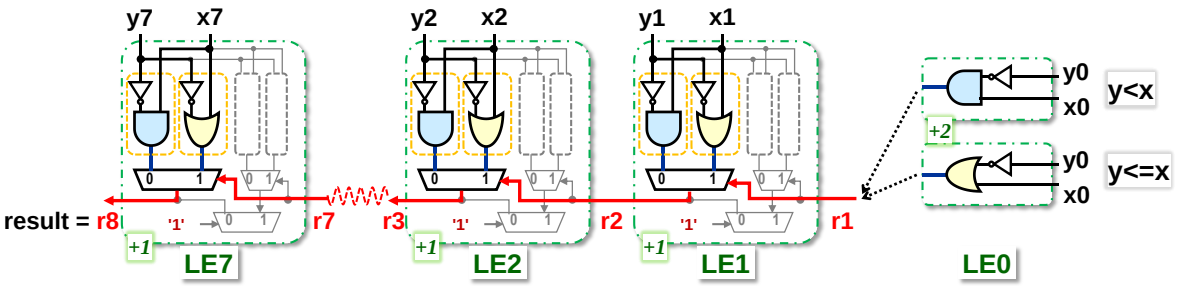
Porovnání x<=y a x<y pracují stejně, ale liší se posledním bitem. Pokud se v bitu s indexem m, xm nerovná ym, tedy ym xor xm='1', nehrají nižší bity význam. Stačí tedy jen výsledek porovnání do vyššího řádu jako dílčí výsledek r_{m+1}. Pro ni platí, že za xm='1' a ym='0' je porovnání pravdivé, a naopak nepravdivé při xm='0' a ym='1'.

Jsou-li bity xm a ym rovné, pak se na výstup r_{m+1} propouští podmínka r_m z nižšího řádu.

Až u porovnání nejnižšího bitu se rozhodne o typu x<=y nebo x<y, a to výsledkem posílaným při shodě nejnižších bitů, u x<=y je to '1', neboť podmínka se splnila, u x<y pak '0'.

Logické elementy FPGA využijí svou *Carry Chain* konfigurací. Přepíšeme-li uvedené podmínky na logické funkce, získáme rozklad na logické elementy LE0 až LE7.

Nejnižší LE0 má běžnou konfiguraci, ale jeho výstup posílá do přímého propojení k LE1. Vyšší logické elementy pracují v *Carry Chain* konfiguraci. Na rozdíl od sčítaček se při komparaci neposílá ven bit součtu, a tak se využívá jen jedna 3-LUT. Mezi nimi se dílčí výsledky ri šíří stejně tak rychle jako ve sčítačce. Poslední r8 je výsledkem komparace.



Obrázek 125 - Komparátor rozložený do logických elementů s 4-LUT

6.2.1 Porovnání s konstantou

Snadno odvodíme dvě bazální pravidla:

- Porovnání typu x rovná se K, kde K je celočíselná konstanta, respektive x nerovná se K, vyžaduje sestavení celého mintermu, neboť rovnost nastane pro jedinou hodnotu čísla x.
- Jiná porovnání s K vyjdou snáze, pokud splňující obě následující podmínky:
  a) K je beze zbytku dělitelné 2^M; M>0. Má pak M nulových bitů na pozicích 0 až M-1.
  b) Podmínka se dá zapsat buď ve tvaru x<K, nebo K<=x.

  Dolních M bitů čísla x pak neovlivní výsledek, neboť ty mají váhy 2^p; p=0..M-1., které jsou nižší než 2^M prvního nenulového bitu konstanty K. Překladač našeho návrhu vloží jen obvod porovnání horních bitů x, což nejen zrychlí zapojení, ale zmenší obvod.

Může užít zejména u čítačů, u nichž se často testuje dosažení žádané hodnoty. Pokud není z důvodu funkce nezbytně nutné porovnávat na rovnost, pak **nerovnost vede na jednodušší** obvod. FPGA má sice hodně logických elementů, ale i tak se hodí tvořit výhodnější podmínky.

6.3 Konstanty užité k násobení, dělení a modulo

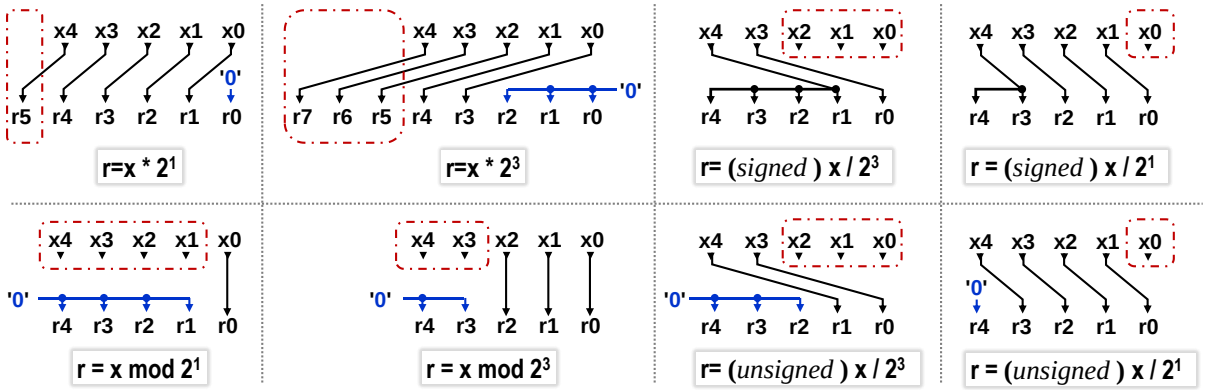
Nechť je K>0 celočíselnou konstantou a x značí celé binární číslo *signed* nebo *unsigned*. Budeme studovat obvodovou realizaci operací, které mají v jazyce C zápisy:

x * K; x / K;

x % K; *zbytek po dělení, ve VHDL zapisovaný jako* x mod K (A1)

6.3.1 Mocnina dvou: K=2^M; M>0

Mocniny dvou patří k nejoblíbenějším hodnotám číslicové techniky. V obvodech se s nimi provedou výrazy (A1) nejrychleji ze všech logických operací, jelikož se realizují pouhým propojením vodičů, jak ukazuje obrázek dole.



Obrázek 126 - Operace s mocninou dvou s 5bitovým číslem x

- Násobení K=2^M je posunem doleva[36] o M bitů. Pokud se žádá výsledek stejné bitové délky, jakou má vstup x, pak horní bity zmizí. Označily se orámováním.
- Dělení K=2^M odpovídá posunu doprava, při němž vypadávají dolní bity 0 až M-1, takže **celočíselná aritmetika ořezává vše pod binární řádovou tečkou**, k čemuž přihlédneme v dalších kapitolách.
- Liší se dělení čísel bez znaménka, typ *unsigned*, a s ním, typ *signed*. V případě *unsigned* x se horní bity r zaplňují logickými '0', zatímco u *signed* se do nich kopíruje nejvyšší bit čísla x, v obrázku x4, neboť určuje znaménko, které se musí zachovat.
- Operace modulo, zbytek po dělení, se realizuje pouhým výběrem bitů s indexy 0 až M-1. Zbývající bity výsledku r se vyplní '0'.

Poznámka: I v programovacích jazycích se výše uvedené operace s 2^M často překládají na posuny. Zbytek po dělení se zas realizuje maskou, která bitovým & vybere M dolních bitů.

[36] Směry posunů se v obvodech udávají vždy podle vah přiřazených bitům, nikoli podle jejich rozložení na schématu. Posun doleva odpovídá přemístění bitu do pozice, v níž bude mít vyšší váhu, doprava je opačný. V jazyce je posun doleva << a doprava >>. Procesory oba realizují rychle na multiplexorech.

6.3.2 Násobení součtem mocnin dvou

Máme-li celočíselnou konstantu $K=2^{M1}+2^{M2}$; $M1>=0$, $M2>=0$, pak se v obvodu všechna násobení typu x*K realizují součty dvou hodnot, a to vstupu x posunutého doleva o M1 bitů a o M2 bitů. Hardwarové násobičky sice provedou výpočet skoro stejně rychle, avšak všechny se uvnitř FPGA nacházejí na pevných pozicích. Sčítačka se dá z logických elementů vybudovat kdekoli. Překladač tak získá větší volnost v rozmístění prvků obvodu.

Zkuste tak zadávat konstanty, je-li to možné. Vzpomeňte si na pravidlo třeba u volby dimenzí matic uložených v SRAM paměti.

Příklad: V obvodu potřebujeme hodnoty uložené v paměti v matici 22x30. Pokud ji uložíme po řádcích, pak výpočet adresy elementu z jeho indexů potřebuje násobení 30, délkou řádku, viz obrázek dole. Uložením po sloupcích si nepomůžeme, násobili bychom 22.

Bude-li se paměťová adresa prvku matice počítat v obvodu na více jeho místech, na každém z nich se pak musí zapojit další hardwarová násobička.

Máme-li dost volné paměti, pak s výhodou uložíme naše data s nadbytečností. Přidáme sloupce, třeba vyplněné 0, abychom jejich počet vhodně zarovnali na výhodnější konstantu, nejlépe na mocninu 2. Naši matici rozšíříme o dva nulové sloupce na 22x32. Násobíme pak 32, což se zapojí pouhým přepojení vodičů.

	0	1		29
0	x0,0	x0,1	…	x0,29
1	x1,0	x1,1	…	x1,29
	…	…	…	…
21	x21,0	x21,1	…	x21,29

memory address = 30 * row_index + column_index

address	0	1		29	30	31		30*21-1
element	x0,0	x0,1	…	x0,29	x1,0	x1,1	…	x21,29

	0	1		29	30	31
0	x0,0	x0,1	…	x0,29	0	0
1	x1,0	x1,1	…	x1,29	0	0
	…	…	…	…	0	0
21	x21,0	x21,1	…	x21,29	0	0

The simpler evaluation of memory addresses
address = 32 * row_index + column_index

address	0	1		29	30	31	32	33		61	62	63	..	32*21-3	32*21-2	32*21-1
element	x0,0	x0,1	…	x0,29	0	0	x1,0	x1,1	…	x1,29	0	0	..	x21,29	0	0

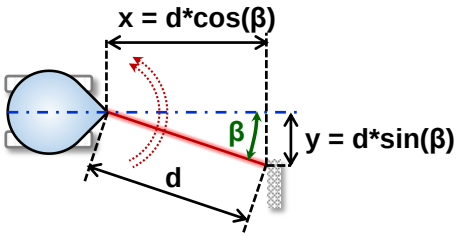
Obrázek 127 - Matice přizpůsobená obvodu

Lze též upravit počet sloupců na součet mocnin dvou, ten se provede pouhým součtem dvou posunutých čísel, tedy opět bez hardwarové násobičky.

6.3.3 Násobení malých hodnot reálným číslem, třeba goniometrickou funkcí

V FPGA se občas nevyhneme reálným číslům. Může uvést příklad přepočtu informace z laserového dálkoměru, jehož paprsek se rozmítá rotujícím hranolem, jímž se vytvoří jeho kmitání z pravé strany na levou. Tok dat je extrémní, a tak naměřené údaje se nutně musí předzpracovat hardwarovým akcelerátorem, aby nezahltily procesor.

Na prvním stupni akcelerátoru se přepočtou změřené vzdálenosti d na polohu překážky x a y vůči ose drona jako předstupeň dalšího zpracování, v němž se i filtrují ruchy, ale tady se již pracuje i s uloženými daty předchozích běhů paprsku, a tak se omezíme na úvodní kombinační blok konverze na souřadnice x a y.



Při přepočtu řešíme dvě otázky:

1. Inkrementální snímač posílá okamžité natočení rotujícího hranolu, které převedeme integer aritmetickými operacemi na úhel β od osy drona, pro nějž vypočteme **sinus a cosinus**. Goniometrické funkce, nebo jiné složité, se v obvodech nepočítají, ale tabelují se v pamětech ROM. FPGA běžně obsahují 2-portové paměti, které dovolují číst ze dvou různých adres naráz; vyzkoušíme si je i v úloze našeho předmětu LSP. Hodnotu úhlu jen převedeme adresy ROM, z níž načteme sin(β) a cos(β).
2. Goniometrické funkce dávají reálná čísla v intervalu <-1;1>. Jak je uložíme? Odpověď z ní, že využijeme aritmetiku v pevné řádové čárce, *fix-point*.

Aritmetika v pevné řádové čárce vyjadřuje všechna čísla jako zlomky se stejným jmenovatelem 2^N, kde celé číslo N>0, aby se jimi dobře násobilo a dělilo. Volba N závisí na námi požadované přesnosti. Ta bude sice nižší než u reálných čísel, ale musíme vzít v úvahu, že úhel β a d vzdálenost změřená dálkoměrem jsou zatížené chybami, které lze očekávat kolem 1 %.

V pevné řádové čárce musíme hlídat rozsah, aby nedošlo k podtečení výsledku, jinak pracujeme shodně jako s čísly integer, to dle pravidel pro zlomky o shodném jmenovateli. Násobení integer konstantou je snadné, ale součin dvou čísel v pevné řádové čárce se následně koriguje posunem doprava o N bitů, aby si zachoval stejný jmenovatel. Při něm se však uřezávají dolní bity, a tak přidáme zaokrouhlení přičtením poloviny 2^{N-1}, tedy poloviny 2^N.

$$\frac{Pa}{2^N} \pm \frac{Pb}{2^N} = \frac{\boldsymbol{Pa \pm Pb}}{2^N};\ K * \frac{Pa}{2^N} = \frac{\boldsymbol{K * Pa}}{2^N};\ \frac{Pa}{2^N} * \frac{Pb}{2^N} = \frac{Pa * Pb}{2^{2N}} = \frac{(\boldsymbol{Pa * Pb + 2^{N-1}})/\boldsymbol{2^N}}{2^N}$$

Zvolíme-li N=10, tedy 2^N = 1024. Uložíme si tabulku hodnot funkce sinus ve vhodném kroku. Stačí nám jen rozsah 0 až 90 stupňů, z něhož odvodíme ostatní hodnoty. Každá se však vynásobí 1024 a uloží se jako 11bitové číslo integer, aby se nám vešla i 1 převedená na 1024.

Má-li úhel β hodnotu odpovídající třeba 10 stupňů, pak jeho sinus bude 0.173648..., ale v ROM bude vynásobený 1024; po zaokrouhlení tedy **178**. Hodnotu cosinu najdeme ve stejné tabulce na úhlu 80 stupňů, kde je 0.984807 vynásobený 1024 na **1008**.

Pokud dálkoměr hlásí 900 mm, pak spočteme: xfix=900***1008**=907200. Výsledek konvertujeme zpět na integer pomocí posunu doprava o N bitů, před nímž přičteme 2^9 kvůli zaokrouhlení. Spočítáme x=(907200**+2⁹**)/2^{10} = 886. Přesné x=886.327. Naše x má chybu jen 0.03 %.

Analogicky získáme i yfix=900***178**=160200, z toho y=(160200+2^9)/2^{10} = 156 po převodu se zaokrouhlením. Jeho hodnota vykazuje chybu 0.2 %. (Přesné y=156.283). Do dalších stupňů akcelerátoru tak posíláme souřadnice zatížené menšími chybami, než mají výchozí data.

6.3.4 Dělení malou konstantou

I číslo 1/K lze vyjádřit jako fix-point, tedy zlomkem P/2^Q. Ukážeme si jeden z možných způsobů, jak lze ručním výpočtem najít přesnější celočíselné konstanty P a Q. Využijeme bazál-

ního pravidla [37], že každé celé K>0 lze rozložit na $K=2^Q * D$; Q>=0 a D je liché číslo a platí, že ke každému celému lichému číslu D existují celá čísla P>0 a N>0 taková, že $D*P=(2^N-1)$. Někdy se však dříve najde rozklad $D*P=(2^N+1)$, který existuje jen pro některá D, třeba $17 = 2^4+1$.

Až najdeme rozklad, pak ho v obvodu dělení aproximuje jednou ze dvou možností:

$$\frac{x}{K} \approx E_{LT} = x * \frac{P}{2^N} \qquad \text{výraz } E_{LT} < x/K\text{; má } \textbf{–chybu} = \quad -\frac{1}{K * 2^N} \qquad \text{(A2)}$$

$$\frac{x}{K} \approx E_{GT} = x * \frac{P+1}{2^N} \qquad \text{výraz } E_{GT} > x/K\text{; má } \textbf{+chybu} \approx \quad \frac{K-1}{K} * \frac{1}{2^N} \qquad \text{(A3)}$$

Při vhodném N se však chyby ocitnou mimo rozlišovací schopnost našeho zobrazení čísel.

Příklad 1: Převeďte dělení x/10 na násobení.

Číslo $10 = 2*5$. Pětku napíšeme jako $5*3=(2^4-1)$, což upravíme identitou $(x^m+1)*(x^m-1)=x^{2m}-1$.

$$\frac{1}{5} = \frac{3}{2^4-1} * \frac{2^4+1}{2^4+1} = \frac{3*(2^4+1)}{2^8-1} = \frac{51}{2^8-1} \approx \frac{51}{2^8}$$

Můžeme ještě víc zpřesňovat, dle naší momentální potřeby, přidáním dalších kroků:

$$\frac{1}{5} = \frac{3*(2^4+1)}{2^8-1} * \frac{2^8+1}{2^8+1} = \frac{3*(2^4+1)*(2^8+1)}{2^{16}-1} = \frac{13107}{2^{16}-1} \approx \frac{13107}{2^{16}}$$

Chceme však 1/10, bude tedy dělit ještě 2, tedy 2^{17}. Víc nemůžeme, protože by nám příliš narostla bitová délka výsledků součinů.

Aproximace (A2) vykazuje menší chybu, ale zápornou, celočíselné operace uřezávají bity pod binární tečkou, což ovlivní hlavně hodnoty dělitelné K beze zbytku.

Výhodnější aproximace **(A3)** dává lepší výsledky, jak u zaokrouhlení:

$$x/10=(x*\mathbf{13108+2^{16}})/2^{17} \qquad \text{(A5)}$$

tak u celočíselného dělení, kde její kladná chyba nastaví správně část nad binární tečkou:

$$x/10 \approx (x*\mathbf{13108}) / 2^{17} \qquad \text{(A6)}$$

Obě aproximace (A5) i (A6) vydělí deseti přesně čísla x, která mají i 14bitové délky. Poprvé pochybí až u 15 bitů, které se již blíží bitové délce zvoleného základu.

Příklad 2: Převeďte dělení x/11 na násobení.

Nejbližší tvar nalezneme v $11*3=33=2^5+1$.

$$\frac{1}{11} = \frac{3}{2^5+1} * \frac{2^5-1}{2^5-1} = \frac{3*(2^5-1)}{2^{10}-1} = \frac{93}{2^{10}-1} \approx \frac{93}{2^{10}}$$

$$\frac{1}{11} = \frac{3*(2^5-1)}{2^{10}-1} * \frac{2^{10}+1}{2^{10}+1} = \frac{95325}{2^{20}-1} \approx \frac{95325}{2^{20}}$$

Aproximuje užitím (A3) buď jako $x/11 \approx (\mathbf{95326+2^{19}})/2^{20}$, tedy se zaokrouhlováním podílu, nebo bez něho $x/11 \approx \mathbf{95326}/2^{20}$. Konstanta má však 17bitů, ve 32 bitové aritmetice ji lze aplikovat až na čísla 14bitové délky, a to jak ve verzi se zaokrouhlením, tak bez něho.

[37] Vztah lze dokázat přes teorii kongruencí i prostou úvahou. Binární reprezentace čísla 2^N-1 jsou samé 1. Liché číslo D má zas svůj nejnižší bit d0=1. Budeme tedy k D přičítat jeho hodnoty posunuté doleva tak, aby se bitem d0 postupně vyplnily všechny 0, jak původní, tak 0 vzniklé součty. Součet bitových vah posunů dává číslo P.

6.3.5 Přesnější integer násobení a dělení reálným číslem

Obě operace jsou ekvivalentní. Dělení převedeme na násobení převrácenou hodnotou, ale se zvětšováním jmenovatele zlomku pevné řádové čárky, nám **roste** bitová délka mezivýsledků.

V klasickém programování zvolíme formát *double*, ale v obvodu máme omezené efektivní délky integer čísel dostupnými hardwarovými násobičkami. Násobení konstantními zlomky vyjde efektivněji, pokud čitatel aproximujeme Hornerovým schématem výpočtu polynomů, které je sice pomalejší, ale přesnější. Celočíselné dělení postupně ořezává po menších částech.

Příklad dělení 10: Operaci převedeme na násobení reálným číslem 0.1, jehož konverze na binární obraz dává nepřesné číslo, v němž se do nekonečna opakují skupiny bitů "1100". Chceme-li přesnosti 6 dekadických číslic, otestuje mocniny dvou kolem 2^{24}, až najdeme hezký hexadecimální tvar. $0.1 \approx 838861*2^{-23} = 0xCCCCD*2^{-23}$. Zaokrouhlilo se nahoru i za cenu poslední číslice D. Potřebujeme kladnou chybou výsledku kompenzovat ořezávání dolních bitů. Rozepíšeme si číslo v libovolném radixu mocniny 2, třeba $16=2^4$, tedy po hex-číslicích.

$$x * 0.1 \approx \frac{x*12*2^{20} + x*12*2^{16} + x*12*2^{12} + x*12*2^{8} + x*12*2^{4} + x*13}{2^{23}}$$

Zlomek zkrátíme vydělením 2^{20}. Ke sdílení x*12 hodnoty si poslední člen upravíme na 12*x+x.

$$= \frac{x*12 + x*12*2^{-4} + x*12*2^{-8} + x*12*2^{-12} + x*12*2^{-16} + (\boldsymbol{x*12} + \boldsymbol{x}) * 2^{-20}}{2^{3}}$$

Hornerovo schématu výpočtu polynomů zapíšeme odzadu, abychom si členy uspořádali ve směru jejich výpočtu, neboť sčítání musí vždy začít od nejnižšího, tedy od nejmenšího členu.

$$x*0.1 \approx (((((\ \mathbf{x*12 + x}\ )/2^4 + x*12\ )/2^4 + x*12\ )/2^4 + x*12\ )/2^4 + x*12\ )/2^7 \qquad (F1)$$

Navrženou aproximaci násobení zlomkem vyzkoušíme v programu, ale nenapíšeme ji jedním výrazem, který by překladač mohl rozložit i jinak, případně i roznásobit. Chceme přesné pořadí operací. Využijeme opakování x*12, což se v obvodu provede jako suma dvou posunutých x na x*8+x*4, takže naše dělení deseti se v obvodu sestaví ze samých sčítaček.

```
int div10(int x) { int xk=12*x ;   int r = (xk + x) >> 4;  r = (xk + r) >> 4;  r = (xk + r) >> 4;
                   r = (xk + r) >> 4; return (xk + r) >> 7;  }
```

Algoritmus dělí i 22bitové x. Jeho verzi se zaokrouhlení výsledku vytvoříme, když před každým posunem doprava přičteme polovinu čísla, jímž se při něm dělí, co bude 8, v závěru 64.

```
int div10(int x) { int xk=12*x ;   int r = (xk+x+8)>> 4; r = (xk+r+8) >> 4;r = (xk+r+8) >> 4;
                   r = (xk+r+8) >> 4; return (xk + r+64) >> 7;  }
```

Máme přesné výsledky až do 21bitového vstupu x. Zkusíme i delší rozklad x*0.1 ve vyšším radixu $256=x^8$. Jeho 8bitové konstanty pracují na 32bitové aritmetice přesně až 23bitových x.

```
int div10ex(int x) { int xk=x*204; int r=(xk+x) >> 8; r=(xk+r) >> 8; return (xk+r)>>11; }.
```

Celočíselně lze vynásobit jakýmkoli reálným číslem, jen málokdy se sdílením dílčích násobení jako v případě binárního obrazu 0.1. Například sinus 10 stupňů, z kapitoly 6.3.3, aproximujeme v radixu 2^{10} výrazem: $(((x*252+512)/2^{10}+x*269+512)/2^{10}+711*x+2048)/2^{12}$ se zaokrouhlením mezivýsledků. Jeho 10bitové konstanty spočítají součiny i s 21bitovými čísly x ve 32bitové integer aritmetice, a to se zaručenou chybou < 0.1%, ale s průměrnou jen $2.6*10^{-7}$ % oproti hodnotě získané výpočtem v *double* přesnosti: round(x*sin(M_PI*10/180)).

Dělení se zaokrouhlováním lze rychleji spočítat rozkladem na paralelní výpočty[38], ovšem s posuny doprava až o několik řádů bitových délek radixů. Ty se špatně kompenzují, chceme-li aproximovat celočíselné dělení, které budeme potřebovat hned v kapitole 6.4.1 na str. 114.

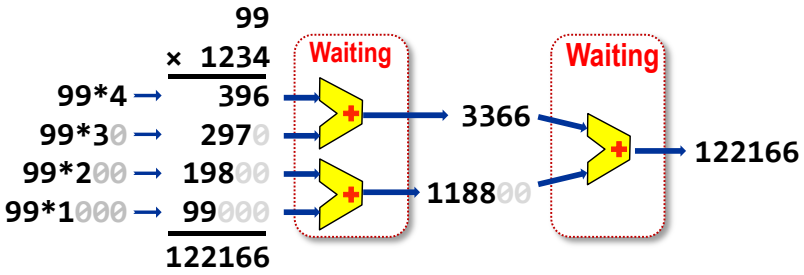
Zde uvedené aproximace předpokládají **celočíselnou aritmetiku** s uřezáváním dolních míst. Kdyby se počítaly v pohyblivé řádové čárce, která je zachovává, vyšly by větší chyby.

6.3.6 Hardwarové násobičky

FPGA zpravidla obsahují hardwarové násobičky, které sice budeme jen využívat, ale i tak se podíváme na jejich princip. Jednak již známe veškeré jejich komponenty, jen je propojíme, a jednak se opět jedná o hezkou demonstraci techniky rozkládání algoritmu na paralelní běh v obvodech.

Pro zjednodušení si objasníme algoritmus hardwarové násobičky, která aplikuje princip podobný ručním výpočtům, kterou popíšeme v nám bližší dekadické soustavě.

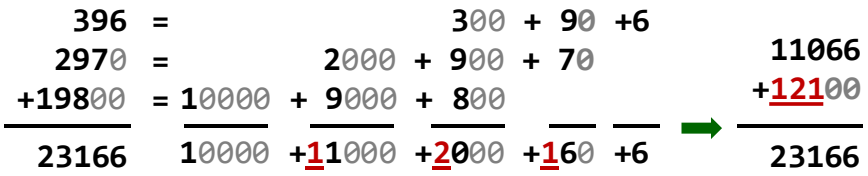
Násobíme-li třeba celá čísla 99 a 1234, pak získáme čtyřmi výsledky dílčích násobení, posunuté vždy o řád. Můžeme je paralelně sečíst po dvojicích a mezivýsledky pak sečíst, což bude výsledným součinem.



Podobné zapojení není však výhodné. Mezivýsledky levé řady sčítaček nás vůbec nezajímají. Chceme znát až výsledný součet, náš součin. Navíc čekáme na ustálení dvou řad sčítaček, v nichž se šíří přenosy, neboť se přičítají přenosy z nižšího řádu. A to zdržuje.

Co kdybychom je nesčítali ihned, ale až v závěru? Můžeme je v mezistupních vyvést ven coby další výstup. Ukažme si princip napřed na horní trojici výsledků dílčích násobení, a to na 396+2970+19800. Každé číslo rozložíme na jeho řády, tedy na součty desetitisíce, tisíců, stovek, desítek a jednotek, které zpracujeme nezávisle. Šedé nuly jen vyznačují řád číslice, ty budeme jen kopírovat.

Například stovky sečteme jako tři číslice 3+9+8=20 a k výsledku doplníme dvě nuly. Nečekáme zde na žádné přenosy. Sčítáme všechny řady paralelně a nezávisle. Z jejich součtů pak sestavíme dva sčítance, do prvního zapíšeme číslici řádu a druhou tu, která přes něj přetekla. V obrázku dole je zvýrazňuje podtržení.

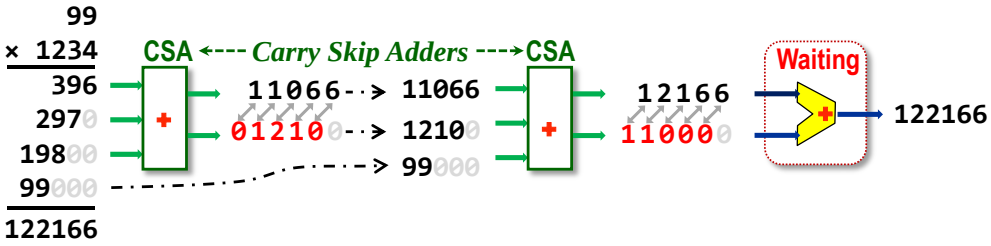


Obrázek 128 - Princip CSA sčítačky

[38] Jiné paralelní metody vhodné v FPGA probírá článek Ugurdag F.,Dinechin F., Gener Y., Gören S., Didier L.: Hardware division by small integer constants. IEEE Transactions on Computers, 2017,

Původní tři sčítance jsme paralelním výpočtem zredukovali na dva, čímž jsme snížili počet sčítaných členů, jejichž součet dává pořád správný výsledek.

Sčítačka, která umí redukci, se nazývá CSA, *Carry-skip Adder*. Řidčeji se v některých publikacích označuje i jako *Carry-bypass Adder*. Ustálené české termíny se žel nepodařilo najít. Postupným skládáním sečteme i původní čtveřici.



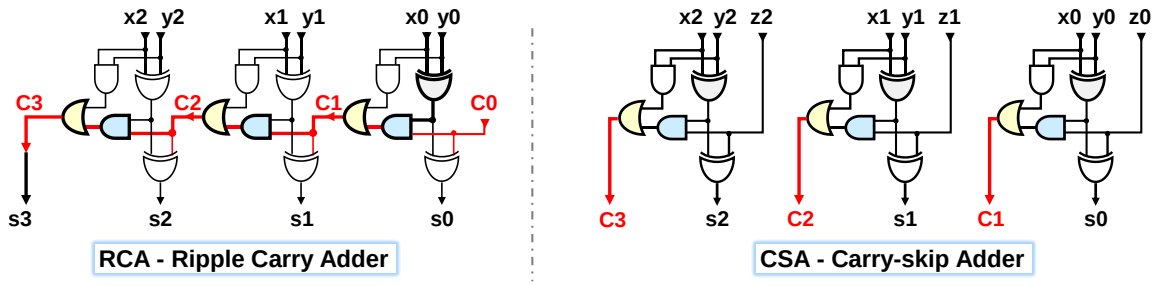
Obrázek 129 - Princip hardwarové násobičky s Wallacovým stromem

Pomocí CSA jsme zapojili hardwarovou násobičku nazvanou Wallacovým stromem, dle jejího autora. Její sčítání trojic zdůvodňuje, proč FPGA hardwarové násobičky mají činitele bitových délek dělitelných třemi, třeba 18×18 či 9×9. Dílčí výsledky násobení se jim efektivně slučují. Wallacův strom sčítání zkracuje dobu násobení. Provede ho za jen o málo delší dobu než součet dvou čísel, protože v mezistupních se nečeká na přenosy, ale u větších bitových délek bývá již objemný. Další typy násobičky od něj odvozené se ho hlavně snaží zredukovat, aniž by došlo k většímu zpomalení.

Struktura Wallacova stromu z předchozího obrázku ale nefunguje na čísla se znaménkem. Záporné výsledky dílčích násobení by v binární soustavě potřebovaly totiž znaménková rozšíření na celou délku finálního výsledku, aby zůstaly pořád záporné. Wallacův strom lze však minimální zásahy modifikovat na Baugh–Wooleyův algoritmus. Stačí v něm jen negovat vybrané bity ve výsledcích dílčích násobení. Chyby při sčítání záporných čísel se pak navzájem anulují. Matematické vysvětlení leží mimo rozsah naší učebnice[39].

Jak však CSA sčítačku tří čísel zapojíme? Nemusíme, už ji máme. Dříve probraná úplná sčítačka přece sčítala tři bity, vstupy x a y a přenos C z nižšího řádu. Její přenos posílaný do vyššího řádu pouze vyvedeme ven jako další její výstup.

Obrázek ukazuje srovnání 3bitových sčítaček typu RCA a CSA. U CSA se jen zavedlo jiné označení vstupu přenosu, nyní bude další číslem, a zrušilo se propojení přes přenosy.



Obrázek 130 Srovnání sčítaček RCA a CSA

Sčítačka CSA využije na úrovni CMOS i naše AND-OR hradlo, takže pracuje se zpožděním jen dvou členů. Mezistupně Wallacova stromu tak příliš nezdržují a hardwarové násobení je pouze nepatrně pomalejší než sčítání.

[39] Nástin Baugh–Wooley algoritmu najdete třeba na: https://www.dsprelated.com/showarticle/555.php.

6.3.7 Problematické obecné dělení dvou čísel

Obecné dělení se zatím neumí efektivně paralelizovat. Návrhová prostředí ho dovedou sice zapojit algoritmem postupného odečítáním jako při ručním výpočtu, ale dostaneme pomalý obvod se spoustou logických elementů. V nouzi ho může použít, pokud nenajdeme jinou cestu, ale lepší je se mu vyhnout.

Existují i zapojení rychlých děliček, ale všechny známé mají složité realizace, které se musí realizovat na úrovni CMOS, aby zůstaly efektivní. Výkonné procesory často aplikují nějakou variantu *High-Radix Division* algoritmu, který tvoří výsledek po skupinách bitů, odtud i jejich název, čímž se počet kroků zkrátí[40]. Vyberou skupiny nejvyšších bitů okamžitého zbytku po dělení, dělence a dělitele, spojí je v adresu do ROM paměti[41], v níž načtou pravděpodobný dílčí podílu a nový zbytek po dělení. Odhad upřesňují iteracemi během sčítání dílčích výsledků násobení ve stromu paralelních násobiček.

Metodou dobře aplikovatelnou v FPGA je také postupné rozšiřování zlomku z dělence a dělitele členy $1+2^M$, dokud se chyba nesníží pod rozlišovací schopnost aritmetiky[42]. Dělení se tím převede na násobení, ale za cenu velké spotřeby FPGA prvků.

Obecné dělení se příliš nehodí k implementaci v logických elementech FPGA. Je-li nutné, hodí se zvolit si typy FPGA s jeho hardwarovou podporu.

6.4 Příklad: Konverze algoritmu na zapojení obvodu

6.4.1 Příklad 1: Převod binárního čísla na BCD

Převod čísla provádí v jazyce C třeba funkce prinf(). I v obvodech lze zapojit konverzi, například pokud chceme číslo zobrazit třeba na 7-segmentovém displeji. Pak každou dekadickou číslici výsledku 0 až 9 potřebujeme mít samostatně ve čtveřici bitů. Podobný způsob se nazývá formát BCD, *Binary Coded Decimals*, viz Binární prerekvizita. Jeho tvar se podobá hexadecimálnímu zápisu čísel s jediným rozdílem, že ve čtveřicích bitů jsou jen binární kódy čísel 0 a 9. Neobjeví se v nich 10 až 15, hexadecimálně zapisované jako A až F.

Postup si vyzkoušíme napřed programem, třeba v jazyce C. Víme, že se algoritmy se v kombinačním obvodu realizují technikou *inline expansion*. Nehodí se tedy cykly závislé na vstupní hodnotě, zde vstup x, u nichž není pevně determinovaný počet opakování.

Náš prvotní experiment může třeba vyjít z běžného algoritmu s postupným dělením. Máme v něm těsně vedle sebe jak dělení, tak jeho zbytek. Překladač C jazyka pak snáze detekuje, že budou potřeba oba výsledky instrukce strojové DIV, jak podíl, tak zbytek.

```
int byte2BCD_v1( byte x )
{   int bcd = 0, d, m;
    for (int ix = 0; ix <= 1; ix++)      { d = x / 10; m = x % 10; // our hint to C compiler
                                           bcd |= (m << (4 * ix)); x /= d; }
    return bcd |= (d << 8);   // max. upper digit can be 2
}
```

[40] Srozumitelný popis uvádí třeba výukový materiál https://www.utdallas.edu/~ivor/ce6305/m13.pdf

[41] Právě 5 špatných hodnot v ROM paměti způsobilo slavnou chybu dělení v procesoru Pentium (rok 1994).

[42] Popis uvádí například: G. Paim, P. Marques, E. Costa, S. Almeida and S. Bampi, "Improved goldschmidt algorithm for fast and energy-efficient fixed-point divider," 2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2017, pp. 482-485, doi: 10.1109/ICECS.2017.8292070.

Pokud budeme obvod zapojovat přesně podle C kódu, pak jsme ho nenavrhli, ale naprogramovali:-(Dělení 10 převedeme raději na násobení zlomkem. Vstupem byte, a tak využijeme postup z kapitoly 0 na str. 109, kde se aproximovalo celočíselné dělení 10 bez zaokrouhlení. Zbytek získáme odečtením, tedy: d = (13108 * ix) >> 17; m = x - d * 10;

Zkusíme upravenou verzi, kterou kompilátor nejspíš konvertuje *inline expansion*, aby zmizel for-cyklus, který je krátký a s malým počtem opakování smyčky. Akorát by zdržoval.

Opravený kód

```
int byte2BCD_v2(byte x)
{   int bcd = 0, d, m;
    for (int ix = 0; ix <= 1; ix++)
    {   d = (13108 * x) >> 17; m = x - d*10;
        bcd |= (m << (4 * ix)); x = d;
    }
    return bcd |= (d << 8);
}
```

Inline expansion

```
int byte2BCD_v2(byte x)
{   int bcd = 0, d, m;
    d = (13108 * x) >> 17; m = x - d * 10; // ix=0
    bcd |= m; x = d;
    d = (13108 * x) >> 17; m = x - d * 10; // ix=1
    bcd |= m<<4; x = d;
    return bcd |= (d << 8);
}
```

Násobení deseti se v FPGA nahradí sečtením dvou posunutých hodnot, $x*2^3+x*2$, a hardwarové násobičky rychle provedou operaci d*13108.

Jde sice už o lepší řešení, ale stále blízké programování. V obvodu se každý převod na BCD vkládá jako samostatné zapojení, a tak se na něj spotřebují dvě hardwarové násobičky. I v něm můžeme sice napodobit volání funkcí a sdílet jejich bloky, využijeme-li synchronní obvody, které řídí konečný automat, *Finite State machine*, FSM. Potřebujeme-li však jen několik BCD převodů, akorát bychom jím zbytečně zvyšovali složitost našeho návrhu.

A co se obejít bez násobiček? I to lze. Z Binární prerekvizity víme, že celé číslo lze konvertovat na binární jeho opakovaným dělení 2, jímž emulujeme posuny doprava. Zbytky po dělení, mizející nejnižší bity, jsou binárními číslicemi, akorát jdou v řadě od bitu 0 k vyšším.

Příklad převodu celočíselným dělením 2 v dekadickém zápisu

13	13÷2=6; mod 1	6÷2=3; mod 0	3÷2=1; mod 1	1÷2=0; mod 1
1101	1101→0110 | 1	0110→0011 | 0	0011→0001 | 1	0001→0000 | 1

Převod užitím unsigned *posunů doprava*

Konverzi lze reverzovat. Můžeme posuny doleva nasouvat i bity převáděného binárního čísla, a to od jeho nejvyššího postupně k nižším. Musíme však jinak násobit dvěma.

Mějme dvě BCD číslice uložené v 8bitovém čísle x, jehož horní (bity x7 až x4) jsou 0000. Posouváme BCD číslice doleva spolu s převáděným binárním číslem x, čímž nasuneme jeho další horní bit. Ten si označíme φ; φ=0 nebo 1. U BCD kódů <= 4 pracuje posun správně:

BCD	**0**	**0**	**x**	**0**	**1**	**x**	**0**	**2**	**x**	**0**	**3**	**x**	**0**	**4**	**x**
	0000	0000	φ--	0000	0001	φ--	0000	0010	φ--	0000	0011	φ--	0000	0100	φ--
←	0000	000φ	--	0000	001φ	--	0000	010φ	--	0000	011φ	--	0000	100φ	--
BCD	**0**	**0+φ**		**0**	**2+φ**		**0**	**4+φ**		**0**	**6+φ**		**0**	**8+φ**	

BCD číslice >= 5 se po vynásobení 2 posunem ale zvětší na hodnoty >=10, čímž se stanou nedovolenými v jeho kódování, viz následující tabulka:

BCD	0	5	x	0	6	x	0	7	x	0	8	x	0	9	x
	0000	0101	φ--	0000	0110	φ--	0000	0111	φ--	0000	1000	φ--	0000	1001	φ--
←	0000	101φ	--	0000	110φ	--	0000	111φ	--	0001	000φ	--	0001	001φ	--
BCD	0	10+φ		0	12+φ		0	14+φ		1	0+φ		1	2+φ	

Chceme-li správný výsledek, musíme přeskočit šest hodnot 10 až 15 chybějících v BCD, což provedeme tím, že každou číslici, tedy čtveřici bitů, před posunem samostatně korigujeme. Vykonáme úpravu tak, že ke každé hodnotě větší než 4, přičteme +3. Po vynásobení posunem doleva bude změna +6, tedy požadované přeskočení hodnot mimo BCD formát. Algoritmus se v angličtině nazývá *double dabble*.

	0	5	0	6	0	7	0	8	0	9
BCD před korekcí	0000	0101	0000	0110	0000	0111	0000	1000	0000	1001
Dočasné hodnoty po +3	0000	1000	0000	1001	0000	1010	0000	1011	0000	1100
BCD po posunu doleva	0001	000φ	0001	001φ	0001	010φ	0001	011φ	0001	100φ
	1	0+φ	1	2+φ	1	4+φ	1	6+φ	1	8+φ

Zkusíme si algoritmus v jazyce C. Tři horní bity vstupu x, pro něž se nebude ještě neprovádět korekce, využijeme k inicializaci proměnné bcd. Korigujeme však pouze dolní dvě BCD číslice, protože převod bytové hodnoty na BCD má třetí číslici nejvýše "0010"=2.

```
int byte2BCD(byte x)
{       int bcd = (x & 0xE0)>>5; // Variable bcd is initilized by upper 3 bits
        for (int ix = 4; ix >= 0; ix--)
        {     if ((bcd & 0xF) >= 5) bcd += 3;  // We correct the least significant BCD digit
              if ((bcd & 0xF0) >= 0x50) bcd += 0x30;  // Correcting the second BCD digit
              bcd = (bcd << 1) | ((x >> ix) & 1);
              // In a circuit, the complex statement above leads to simple connections.
        }
        return bcd;
}
```

Naše finální verze C programu ověřuje algoritmus obvodu. Hodí se jako program leda výpočetním elementům bez hardwarového dělení. Zdržovala by procesory s proudovým zpracováním instrukcí, *pipelining*. Vždy není pro ně. Simuluje zapojení, a ne nějaký „céčkový“ kód! Obsahuje totiž četná větvení instrukcemi if, jejichž podmínky závisí na vstupních datech.

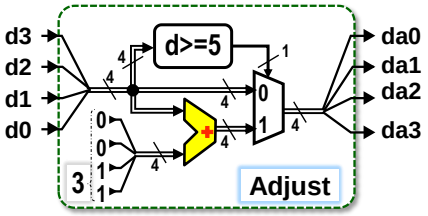
Jednotka procesoru, která řídí běh *pipeline*, načítá strojové instrukce dopředu. A zde neodhadne, co bude následovat po provedení příkazu if, který se vykoná až někdy ve vzdálené nanosekundové budoucnosti. Náhodně by zvolila jedno větvení. Pokud by ho špatně predikovala, musela by zrušit 20 i více již načtených a předzpracovaných strojových instrukcí a začít znovu od skutečně provedeného větvení. Náš poslední kód by ji akorát citelně zpomaloval.

Obvodům ale větvení nevadí. Zpracují se všechny jeho případy paralelně a podmínkou se z nich jen vybírá hotový výsledek.

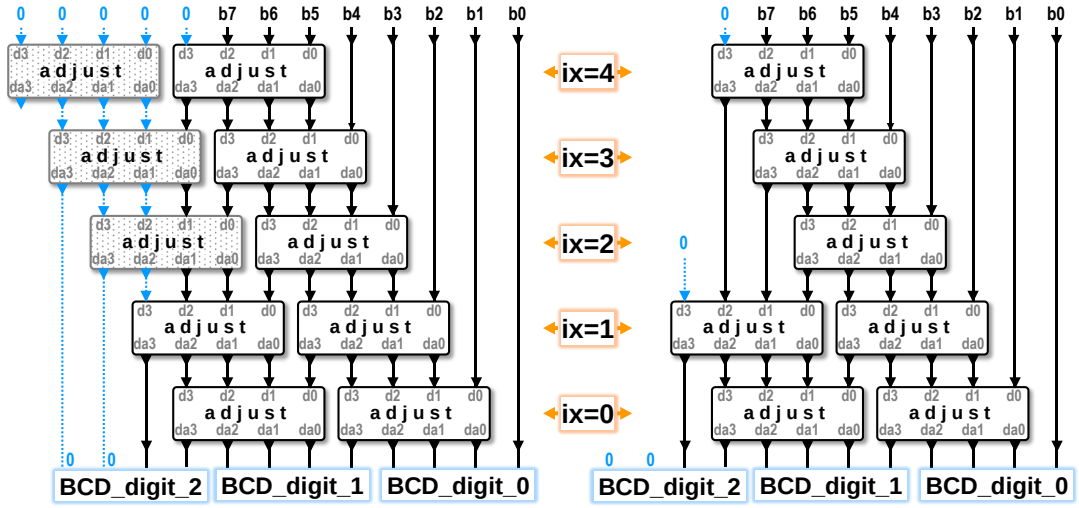
V příkladu jsme demonstrovali rozdíl mezi návrhem obvodů a jeho programováním, každý implementační nástroj chce jemu přirozené postupy.

Obvodovými postupy zapojíme nyní převodník ze vstupu typu byte na tři číslice BCD pomocí nám již známých prvků.

Ke korekci +3 si sestavíme obvod Adjust. Vložíme do něho sčítačku konstanty 3 ("0011") ke vstupnímu d, jímž je 4bitový kód BCD číslice. Podmínku if realizujeme multiplexorem MUX 2:1, jehož adresní vstup je ovládaný komparátorem d>=5. Při jejím splnění se na výstup da obvodu posílá d+3, jinak d.[43]



S užitím Adjust zapojíme převod byte2BCD() podle posledního C kódu. Tělo jeho for-cyklu vkládáme opakovaně technikou *inline expansion*, a pokaždé dosadíme hodnoty 4 až 0 za index ix. Posuny doleva[44] převádíme na pouhá propojení k další řadě obvodů Adjust.



Obrázek 131 - Převodu byte na BCD

Když si prohlédneme výpočetní schéma vlevo, vidíme, že tři Adjust nikdy nepřičtou 3. Buď mají všechny své vstupy nulové, anebo se na ně přicházejí nejvýše 2 bity BCD číslice, které by mohly různé od 0, takže se jen propouští své vstupy na výstupy.

Návrhové prostředí má jako vstup strukturu vlevo. Samo vynechá nadbytečná Adjust a zapojí finální schéma uvedené vpravo.

Napíšeme si převod na BCD v další naší učebnici, v níž se vysvětluje VHDL styl. Realizujeme tam konverzi na BCD čísel libovolné délky jak sérií bloků Adjust, tak konečným automatem (*FSM*), která napodobí opakované užití těla for-cyklu tím, že bude pracovat v taktu. Převod mu potrvá mnohem déle, což nevadí, posíláme-li výstup na zobrazovací segment. Lidské oko nepostřehne, že se hodnota objevila o několik mikrosekund později:-)

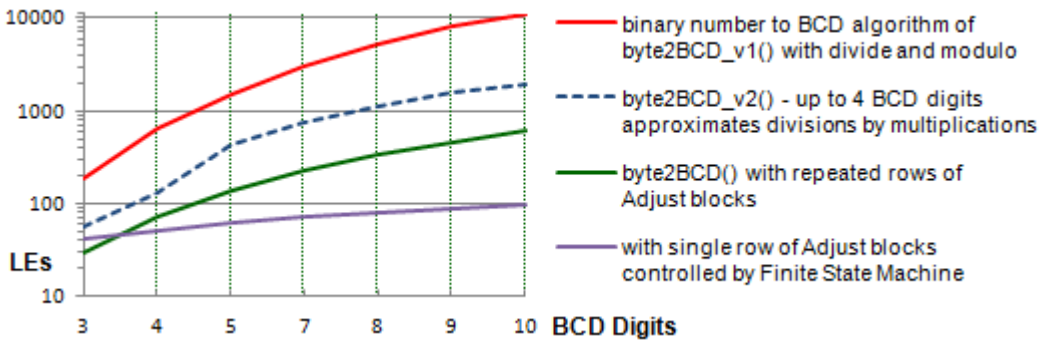
Uvedeme ještě složitost obvodů v FPGA podle ukázaných algoritmů, které se rozšířily na převody i delších vstupních čísel než byte a více BCD číslic.

---

[43] Sice předbíháme, ale pro zajímavost uvedeme, že v HDL jazycích se Adjust obsahující jednoduchý MUX 2:1 popíše jediným příkazem. Ve Verilogu: assign y = x>=5 ? x+3 : x; Ve VHDL: y<=x+3 when x>=5 else x;

Musíme však rozumět tomu, co doopravdy vytváříme. Vždyť HDL, *Hardware Description Language*, znamená popis obvodu. Třeba napřed znát jeho zapojení, teprve pak ho specifikovat příkazy:-)

[44] Opět připomínáme, že směry posunů se vždy určují podle vah bitů, kterou po nich získají, ne z jejich orientace na nakresleném schématu. Výstupní bity Adjust se po posunech dostanou vždy na vstupy další linie, na níž mají vyšší pozice (váhy). Kvůli tomu mluvíme o posunech doleva.



Obrázek 132 - Složitost FPGA obvodů vytvořených ukázanými algoritmy

Z grafu vidíme, že prvotní programové řešení s dělením a operací zbytku je zcela nepoužitelné. Hodí se jen velkým procesorům, na nichž zase pracuje lépe než další naše kódy.

Aproximace dělení jedním násobením lze jen využít v případě vstupů do 14 bitů, poté musíme zvolit komplikovanější metodu podle kapitoly 6.3.5, což vyvolá skokový nárůst složitosti. Není sice velký, neboť zbytek po dělení se dál počítá násobením a odečtením. Nicméně i tak graf naznačuje, že její algoritmus se nehodí ke konverzi na obvod.

Námi zapojená verze se sérií Adjust má skvělé parametry u kratších délek, ale její složitost roste se zvětšováním vstupního čísla. Delší čísla se úsporněji konvertují verzí obvodu s konečným automatem.

Do hodnocení kvality návrhu musíme však zahrnout i **náš čas** věnovaný na jeho vymýšlení, neboť jde o též optimalizovanou veličinu. Do nejběžnější délky pěti až sedmi zobrazených číslic realizujeme převod nejsnáze mnoha řadami Adjust. Vždyť jsou navzájem přímo propojené, a tak je FPGA snadno rozmístí. Složitost obvodu bude vyšší jen o málo vyšší oproti FSM verzi.

Není-li nutná úspora spotřebovaných prvků z jiných důvodů, pak nemá ani význam, abychom tvořili konečný automat. FPGA obsahuje desítky tisíc logických elementů.

6.4.2 Úkol 2: Zapojte rychlou sčítačku na FPGA

Zde jen rozpačitě pokrčíme rameny. K vyzkoušení funkce můžeme klidně zapojit ledacos. FPGA obvody zvládnou i numerické řešení diferenciálních rovnic v reálném čase a jiné triky, ale sčítačky patří do kategorie obvodů, jejichž zapojení se musí optimalizovat na úrovni CMOS transistorů, aby se zrychlilo. FPGA ho rozvrhuje na pouhé logické funkce.

Můžeme si klidně otestovat i CLA, *Carry Lookahead Adder*. Rovnice jeho predikcí známe z kapitoly 6.1, ale výsledek nebude rychlý. Na Cyclone IV, s nimiž pracujeme v našem předmětu LSP, se na CLA spotřebuje dvakrát tolik logických elementů a bude třikrát pomalejší než RCA, kterou automaticky vytvoří návrhové prostředí. Vyzkoušeno:-)

I když se dovedně aplikují různé triky rozložení predikcí v CLA, abychom využili *Carry Chain* konfiguraci logických elementů, předběhneme RCA sčítačky FPGA až na extrémních délkách, jako zpracování 200bitových a delších čísel[45].

Zapojení nejrychlejší PPA, kterou je KSA, *Kogge-Stone Adder*, by poskytlo lepší výsledky[46]. Jeho 16bitová verze by byla pouze o 20 % pomalejší než výchozí RCA, ale zato by spotřebo-

[45] Hui Li, Zhidong Liang, Hanwen Li, and Yazhou Ye. 2021. A High-Performance Wide FPGA Adder Based on Carry Chains. In Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering (EITCE 2020). https://dl.acm.org/doi/10.1145/3443467.3443868

vala čtyřikrát tolik logických elementů. Ani u ní totiž FPGA nemůže nasadit CMOS triky, třeba ve stylu našeho AND-OR hradla, jímž se výrazně akceleruje i KSA stromová struktura.

Pokud však roste délka sčítanců, KSA se už začíná vyrovnávat výchozím RCA. Její 128bitová FPGA varianta je dokonce o pět procent rychlejší, ovšem na ni se spotřebuje desetinásobek logických elementů než na RCA.

Potřebujeme-li v obvodu realizovat velmi rychlou aritmetiku, pak si zvolíme takový FPGA typ, který podporuje aritmetické operace ve svých přídavných hardwarových blocích. V nich se vytvořily na úrovni CMOS transistorů s využitím celé šíře jejich možností. A mnohé FPGA obvody zahrnují i celé procesory, viz třeba úvodní Obrázek 1 na str. 8. Složité aritmetické výpočty se pak udělají na nich.

Logické elementy vynikají nejvíce v realizaci paralelně spolupracujících logických operací, ne v akceleraci jedné z nich, která navíc vyžaduje optimalizaci na úrovni CMOS, zatímco FPGA zvládne pouhou úroveň logických funkcí.

[46] VHDL popis KSA lze najít třeba na https://github.com/sehraf/genericKSA.
Verze ve Verilogu je třeba na: https://github.com/jeremytregunna/ksa

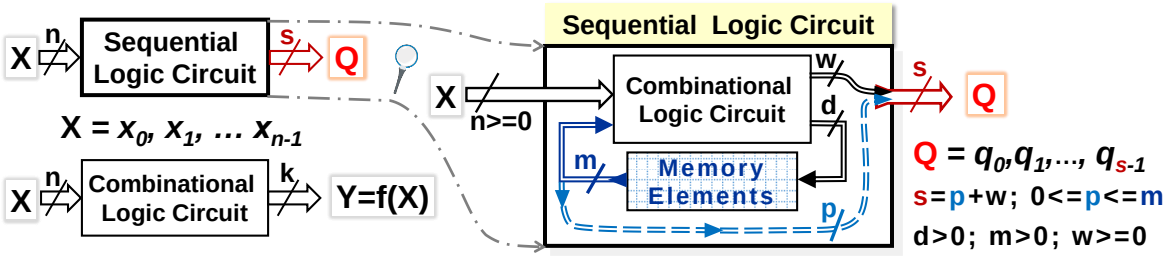
7 Sekvenční obvody

Definice kombinačních obvodů, kterou jsme uvedli v kapitole 3 na str. 27, zdůraznila, že jejich výstupy závisí výhradně na vstupu. Tentýž vstup dává pořád ten samý výsledek.

Sekvenční obvody, pro něž tohle neplatí, přiblížíme jednoduchým příkladem, ve kterém se budeme zajímat o výsledek vrhu dvojicí šestistěnných hracích kostek.

- Kombinační obvod nám vyřeší úlohu součtu právě vržených ok, a to pomocí sčítačky.
- Sekvenční obvod vznikne, žádáme-li třeba klouzavý průměr, *moving average*, současného vrhu a předešlého, který musíme držet v paměti. Výstup bude již záviset nejen na okamžitém vstupu, ale jejich sekvenci mající historii dvou vrhů.
- Klouzavý průměr můžeme i rozšířit na 16 hodů, což iterativně vyřešíme frontou (paměť FIFO) a registrem součtu. Od něho jen odečteme čelo fronty, přičteme k němu současný vrh, který navíc zařadíme na konec FIFO. Výstup již závisí na sekvenci 16 vstupů.
- Bude-li nás zajímat jen součet všech vrhů, pak potřebujeme mnohem menší paměť, ale sekvence se tentokrát prodluží k okamžiku počáteční inicializace po zapnutí napájení.
- Autonomní sekvenční obvod si hodnoty vrhů může i pseudonáhodně generovat, což se lehce zařídí třeba posuvnými registry s lineární zpětnou vazbou, LFSR, které uvedeme na konci této učebnice. Budeme-li jimi tvořit výsledky oba vrhů, pak obvod nemusí mít ani vstup X. Promění se na pouhý generátor.

Příklad můžeme ještě ilustrovat nástinem zapojení obvodu. Vstup X bude složený z hodnot obou vrhů, má délku n=6 bitů, neboť počet ok se vejde do dvou 3bitových čísel.



Obrázek 133 - Příklad sekvenčního obvodu

Oproti kombinační variantě, která jen sčítala, potřebuje sekvenční obvod navíc paměťové elementy. Má sice také kombinační logickou část, ale její vstupy tvoří kompozice dvou složek, z nichž zvnějšku vidíme jenom jednu, a to naše vstupy X. Druhá se načítá z vnitřních paměťových elementů, jejichž obsah je odrazem historie vstupů. Z té vybíráme vhodné veličiny, které připojíme je ke vstupům X, což způsobí, že **výstup Q závisí na sekvenci vstupů.**

Logické funkce v kombinační části vytvoří vnitřní výstupy, z nichž některé se využijí k aktualizaci paměťových elementů, mezi něž patří i registru součtu. Do vnějšího výstupu Q můžeme vyvést i nějaké hodnoty paměťových elementů, pokud nás zajímají, či vytvořit kombinačním obvodem odvozené veličiny, třeba klouzavý průměr nebo i informace, že čelo fronty, paměti FIFO, má nenulovou hodnotu, tudíž se celá zaplnila a výsledek je relevantní, a mnoho dalšího.

Schéma na obrázku nahoře je poměrně univerzální, bude vyhovovat většině sekvenčních obvodů. Zdůrazňuje fakt, že výstup Q závisí rovněž na vnitřních veličinách.

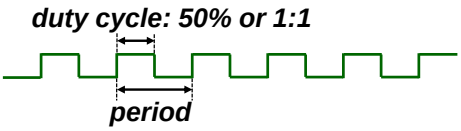
7.1 Terminologie sekvenčních obvodů

Nyní shrneme pojmy, které se používají v sekvenčních obvodech. Některé z nich jsou již hodně ustálené, ale jiné ne.

Hodinový signál

Kterýkoliv signál si lze zvolit za hodinový, *clock*. Zpravidla se vybírá nějaký pravidelný, ale obecně můžeme jakýkoli. Volba je naše. Je-li periodický, pak má následující parametry:

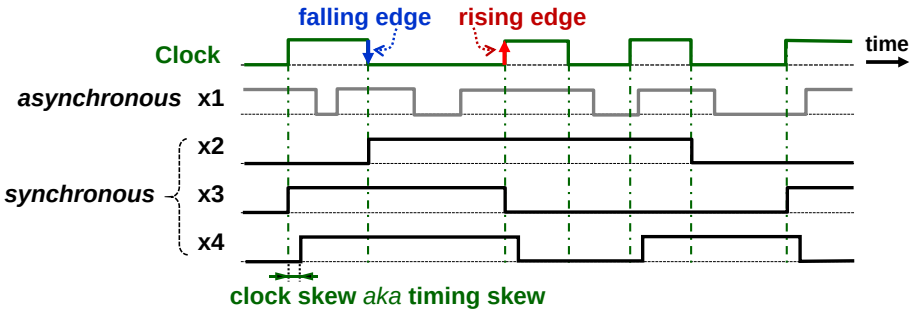
- **perioda** je dobou mezi opakováními.
- **střída**, *duty-cycle*, se udává v procentech periody, po jakou její část setrvá signál ve stavu '1'.
  Vyskytuje i ve tvaru poměru vyšší a nižší úrovně, kdy 1:1 odpovídá 50% střídě či činiteli plnění, ale tohle se vypátralo jen v českých publikacích.



Obrázek 134 - Střída hodin, *duty cycle*

Synchronní a asynchronní vůči hodinám

Vůči hodinám může být jiný signál buď synchronní či asynchronní.

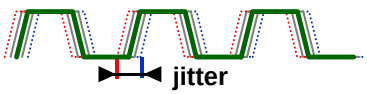


Obrázek 135 - Asynchronní a synchronní signály a clock skew

V obrázku nahoře jsme úmyslně použili neperiodické hodiny. I takové si můžeme vybrat. Mohou přicházet třeba z výstupu z jiného synchronního obvodu:

- x1 signál je asynchronní vůči Clock, neboť se mění nezávisle na průběhu zvolených hodin.
- x2 signál je synchronní se spádovou hranou, *falling edge*, hodin Clock. Pokud se x2 změní, pak se tak stane v okamžicích poblíž přechodu Clock z '1' do '0'.
- x3 signál je synchronní s náběžnou hranou, *rising edge*, kdy Clock '0'→'1'.
- x4 signál je také synchronní s náběžnou hranou hodin, avšak se zpožděním nazývaným ***clock skew***, alternativně jako ***timing skew***. (Český termín není známý). Zpožďuje-li se, pak má kladnou hodnotu, ale může hodiny také předbíhat vlivem různých zpožděními na cestách jejich distribuce. Pak je záporný.

Periodický signál může vykazovat i sníženou kvalitu efektem zvaným i *jitter* (česky nejistota?). Objevuje se u něho nahodilý posun fáze kolem přesné periody.



Jev se vyskytuje především při komunikaci s externími zařízeními a jeho náhodný charakter ho odlišuje od *clock skew*, které naopak mívá relativně stálou hodnotu posunu.

Pojmenování sekvenčních obvodů

Česká terminologie zná jen pojem „klopný obvod“, který se svou zkratkou KO shoduje i s kombinačním obvodem. Žel jeho lidový název „klopák“ (též výraz pro mikrofon) nezískal poctu spisovné formy. Spojení „klopný obvod“ se tak upřesňuje přídavnými jmény prodlužu-

jícími jeho název, jako „bistabilní“, „úrovňový asynchronní“, „hranou řízený synchronní“, apod. Nelze se divit, že v českých textech užívá „klopný obvod“, aniž se upřesňuje jeho typ.

Angličtina má dva pojmy, a to latch a flip-flop. První z nich označuje západku na dveřích a pochází z dob reléové techniky, kdy se vyráběl i jeden typ paměťového relé na podobném mechanickém principu, u nás zvaný „západkové relé“. Druhé slovo flip-flop znamená přemet nazad, prudký obrat o 180 stupňů, ale též boty žabky podle zvuku, který vydávají při chůzi.

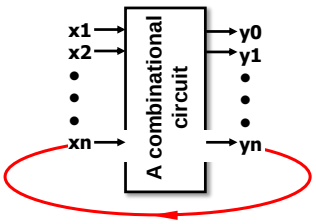
Uvedeme anglickou terminologii podle jejího užití ve vývojových nástrojích obvodů:

- latch v nich vymezuje jen asynchronní klopné obvody, jako RS latch a D-latch. Jejich přesné české ekvivalenty jsou „RS a D úrovňové klopné obvody“.
- *transparent* latch - je sice přesný technický termín, ale místo něj se častěji píše D-latch.
- *edge-triggered* latch - čili hranou řízený latch zahrnuje celou kategorii klopných obvodů, které změní svůj výstup jen při příchodu nějaké hrany hodinového signálu.
- *flip-flop* - určuje v návrhových prostředích nejčastější typ *edge-triggered* latch zapojení, viz dále. To se označuje zavedenou zkratkou DFF, data flip-flop.

Další text obohatíme o nová „ryze“ česká slova latch a flip-flop, s nimiž lze elegantněji psát „flip-flop se překlopil“ místo „klopný obvod se překlopil“. Pojem „klopný obvod“ degradujeme na jejich obecné synonymum. Nainstalovali jsme tím další update našeho jazyka[47] :-)

7.2 Obvod typu RS Latch

Vytvoříme-li *latch* z logických elementů FPGA, vždy jde o **závažnou chybu** v našem návrhu. Překladač ji ohlásí varováním, buď „*combinational loops*“ či vypíše „*inferring latch(es)..*“.

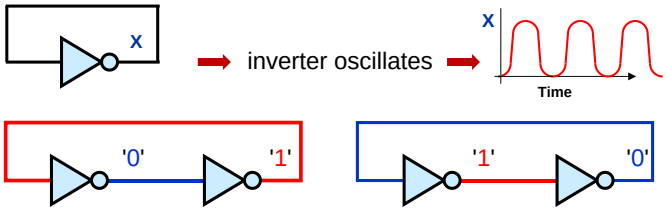


RS latch vznikne snadno, stačí jen nechtěně propojit nějaký výstup kombinačního obvodu s jeho vstupu. Vznikne smyčka, *loop*, v jiných oborech nazývaná zpětnou vazbou, která má paměťový charakter.

Poznámka: Zavedení výstupů obvodu na jeho vstupy je chybou výhradně v ryze kombinačních částech. U synchronních obvodů se běžně používá, ale ty se samy o sobě chovají jako paměťové prvky, a tak jim smyčkou nevnutíme uložení hodnoty výstupu.

Aplikujeme-li souběžné, *concurrent*, příkazy, pak se smyčky v nich lépe uhlídají a vytvoří se jen hrubou chybou. Popisujeme-li obvod stylem *behavioral*, jde relativně častý omyl počínajících návrhářů. Musíme tedy vědět, co jsme tím spáchali, abychom se tomu příště vyhnuli.

Pokud se smyčka uzavře přes lichý počet invertorů, pak se divoce rozkmitá, protože nemá jediný stabilní stav. Vede-li se však přes sudý počet inverzí, pak se hodí se k uchování informace, neboť má dva stabilní stavy.



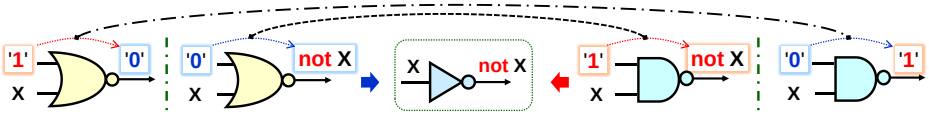
Obrázek 136 - Smyčka invertorů

Stabilní smyčku jsme nakreslili jako dva invertory zapojené za sebou.

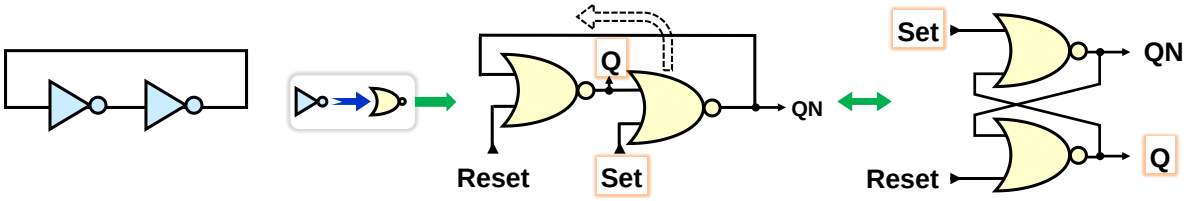
[47] Latch a flip-flop zatím nejsou na webu https://cestina20.cz/, který se v roce 2018 dočkal i knižního vydání.

Šla by teoreticky vytvořit i z hradla *buffer*, ale to je interně složené ze dvou invertorů. Důvody jsme uvedli v kapitole 4.3 na str. 60.

Přidáme k smyčce i možnost změny jejího stavu tím, že invertory nahradíme hradly NOR nebo NAND, u nichž využijeme jejich agresivní a neutrální[48] logické vstupní hodnoty (str. 17).



Napřed si ukáže variantu RS latch, v níž se invertory nahradily NOR hradly:

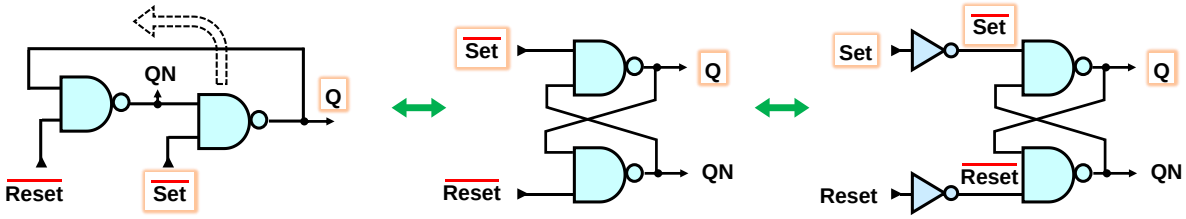


Obrázek 137 - RS latch z NOR hradel

Schéma se častěji kreslí v uspořádání vpravo, v němž se druhé hradlo graficky posunulo nad první. Hlavní výstup obvodu se nazval Q a k němu protilehlý QN, jehož sufix sice naznačuje negaci Q, ale uvidíme dále, nebude jí vždy. Volné vstupy jsme nazvali Set a Reset[49]:

- **vstup Set** - Je-li rovný agresivní '1' pro NOR hradlo, pak mu nastaví jeho výstup QN do '0'. Horní hradlo bude mít '1' na obou svých vstupech a jeho Q přejde do '0'.
  Poznámka: Častou chybou ve zkouškových písemkách bývá právě mylné umístění Set na vstup NOR hradla s Q výstupem.
- **vstup Reset** uvede analogicky QN do '1', což povede k nastavení Q do '0'.

Zapojíme si i častější analogii RS latch s NAND hradly. Ty však mají logické '0' jako své agresivní vstupní hodnoty. Vstupy se u nich často označují jako negované, nebo před ně rovnou doplní invertory. Ty se později i s výhodou využijí v dalších odvozených obvodech.



Obrázek 138 - RS latch z NAND hradel

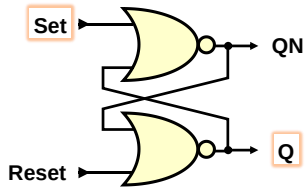
Obě verze vykazují blízké chování, ne ve všech případech, což už ihned ukážou jejich logické rovnice, kterou sestavíme si je postupem podrobně vysvětleným v kapitole 2.4 na str. 25, a vyjádříme výstup Q vyjádříme jako logickou funkci, kde je Q jak na levé tak pravé straně:

$$Q=fn(Set, Reset, Q),$$

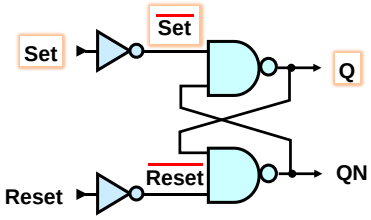
[48] Připomínáme, že česká terminologie kdysi zavedla agresivní a neutrální jako překlad *annulment* a *identity*. V učebnici raději vkládáme nové pojmy jako anglická slova kurzívou a nevymýšlíme si další podobné odlišnosti.

[49] *Jazyková poznámka:* Vstupy RS *latch* sice běžně označují Set a Reset i v anglické literatuře. Jde o tradiční názvy. Slovo „set" je však v angličtině širokým polysémantickým pojmem. Slovník Merrian-Webster u něj uvádí 16 různých významů jako sloveso, 11 jako přídavné jméno a 47 jako podstatné jméno. K nastavení do '1' se u jiných obvodů (ke zcela totožné funkci jako *Set*) upřednostňuje pojmenování **Preset** a operace *Reset*, tedy vymazání do '0', se označuje jako **Clear**, neboť jde o jednoznačnější termíny. Můžeme v nich vidět i obvodová synonyma k pojmům Set a Reset.

Právě skutečnost, že výstup Q závisí sám na sobě, vytváří již dříve smyčku zmíněnou na začátku této kapitoly, neboť v obvodu se čtení hodnoty vždy realizuje propojovacím vodičem od výstupu na vstup. Jinak to ani nejde.



Q =not (Reset or QN)
=not (Reset or not (Set or Q))¨
=not Reset and (Set or Q)



Q =not (not Set and QN)
=not (not Set and not (not Reset and Q))
=Set or (not Reset and Q)

Obrázek 139 - Logické rovnice RS-latch

Sestavíme si ještě jejich pravdivostní tabulky pro oba jejich výstupy Q a QN, a to pouhým dosazování hodnot za vstupy, abychom lépe viděli rozdíly.

NOR RS latch

inputs		*outputs*	
Set	**Reset**	**Q**	**QN**
0	**0**	***memory***	
0	**1**	**0**	**1**
1	**0**	**1**	**0**
1	**1**	**0**	**0**

NAND RS latch

inputs		*outputs*	
Set	**Reset**	**Q**	**QN**
0	**0**	***memory***	
0	**1**	**0**	**1**
1	**0**	**1**	**0**
1	**1**	**1**	**1**

Obrázek 140 - Pravdivostní tabulky RS-latch

Obě varianty se shodují ve třech podstatných řádcích:

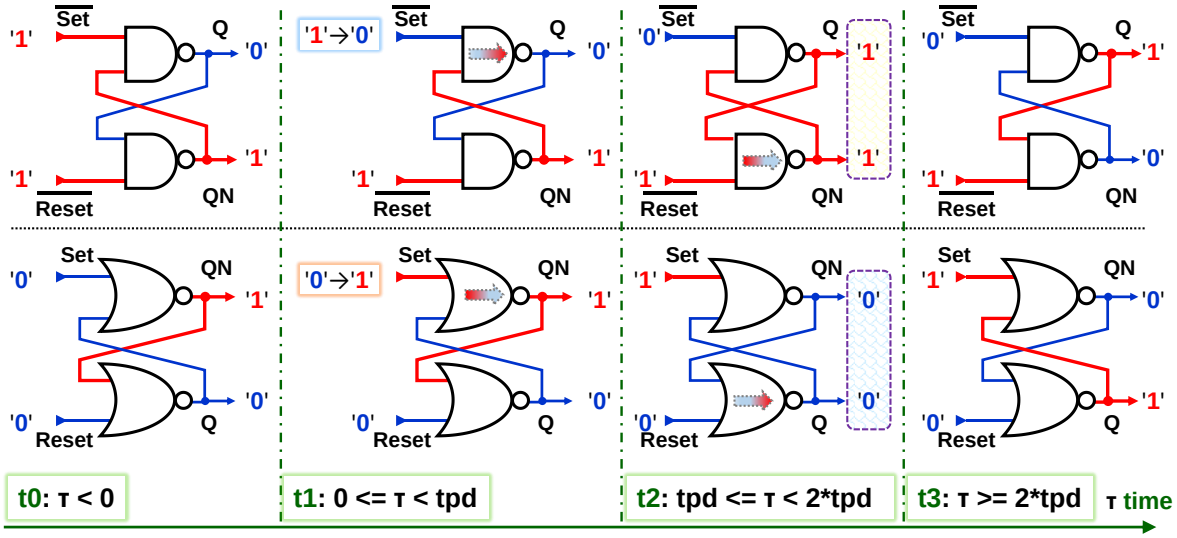
- v paměťovém stavu označeném jako *memory* si výstupy Q a QN drží své poslední hodnoty, a to buď Q='1' a QN='0', nebo Q='0' a QN='1'.
- ve stavu nulování se výstup Q nastaví na '0' a protilehlý QN na '1'.
- při nastavení bude naopak Q='1', zatímco QN='0'.

Pokud jsou Set a Reset vstupy oba v logických '1', pak vidíme odlišné chování:

- NOR verze RS latch má jak Q tak QN v logických '0'. Můžeme tedy prohlásit, že její vstup Reset má vyšší prioritu než Set, což ukazují i logické rovnice.
- Naproti tomu NAND varianta RS latch upřednostňuje Set a za stejné situace se v ní nastaví oba Q i QN na logické '1'.

Ale stav Set='1' a Reset='1' je přece zakázaný !

- Kdepak, hradla nemají žádné zapovězené vstupy!
  Výstup hradla NOR má jít do logické '0' při některém svém vstupu v '1' a NAND hradlo zase do '1', bude-li nějaký jeho vstup v '0'. Něco takového hradlům rozhodně nezakazujeme! Vždyť jde o primární požadavek na jejich činnost.
- **Zakázaný stav vznikne až přidáním omezující podmínky QN = not Q.** Výhradně vůči ní bude zakázaný, ale to ještě pouze ve svém ustáleném stavu.
  Tak zvaný „zakázaný stav“ je především **pracovním stavem** RS latch, ten by se bez něho ani neklopil, což demonstrujeme rozborem jeho dynamickým chováním, viz následující obrázek.



Obrázek 141 - Klopení RS latch

Nechť τ označuje relativní čas, kdy došlo ke změně vstupu Set. Pak v okamžicích:

t0: τ<0 předpokládáme, že oba obvody RS latch mají teď takové své vstupy, v nichž si pamatují. Jejich výstupy si drží své předchozí hodnoty, v obrázku Q='0' a QN='1'.

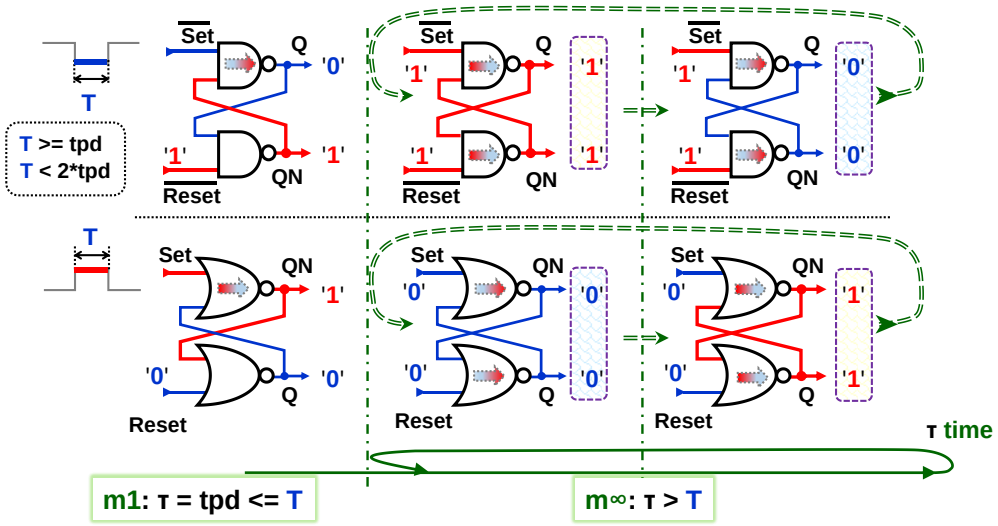
t1: V čase τ=0 se vstupy Set obou obvodů RS latch změnily na hodnoty vyvolávající nastavení jejich výstupu Q do '1'. Doba, která uplynula od změny vstupu, je však zatím kratší než doba zpoždění hradla, **tpd**, *propagation delay*, tedy τ<tpd. Výstup horního hradla se dosud nezměnil.

t2: Od změny vstupu Set již uběhla doba τ>=tpd, takže horní hradlo se překlopilo. Nyní se čeká na dolní. **Oba RS mají nyní dočasně své výstupy Q = QN**.

t3: Teprve za dobu τ>=2*tpd se změnou ovlivní také výstup druhého hradla a oba RS latch doběhnou k novým svým ustáleným stavům svých výstupů.

7.2.1 Metastabilita

Nechť ve stejné výchozí situaci jako nahoře přijde na vstupy Set impulz s polaritou vhodnou k nastavení Q='1'. Délka pulzu bude však delší než tpd, ale kratší než 2*tpd. V obou verzích RS latch se stihnou překlopit jen jejich horní hradla. Obvod se dostal do svého běžného pracovního stavu, kdy má krátkodobě Q=QN. Situaci ukazuje začátek části označené m∞

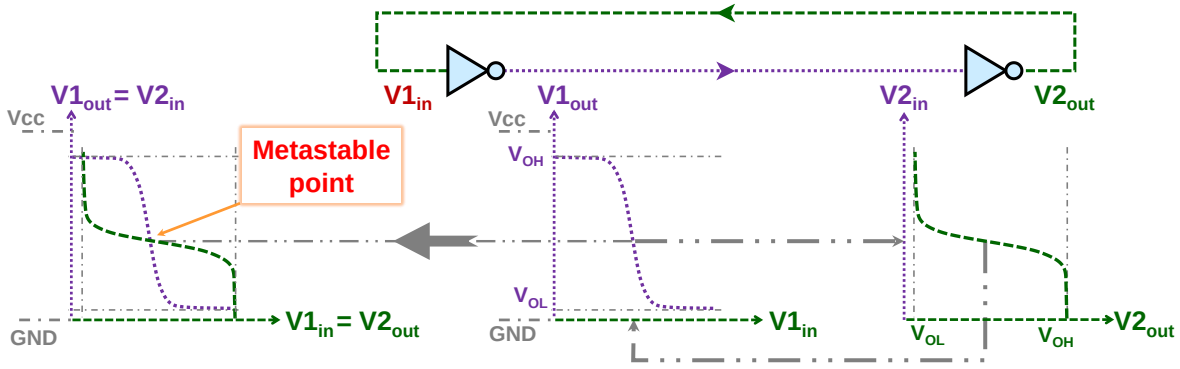


Obrázek 142 - Metastabilita RS latch

Hradla však již nic nediriguje, kam mají pokračovat dál, zda k výstupům Q='0' a QN='1', či naopak ke Q='1' a QN='0', neboť už skončil pulz a vstupy Set a Reset se vrátily k hodnotám, s nimiž si RS latch pamatuje poslední svůj stav. Všechna hradla mají tak hodnoty vstupů na jejich překlopení na opačné hodnoty, ale v těch naopak budou mít jeden ze svých vstupů na agresivní hodnotě nutící je k jejich návratu do předchozích stavů. A z nich zase zpět.

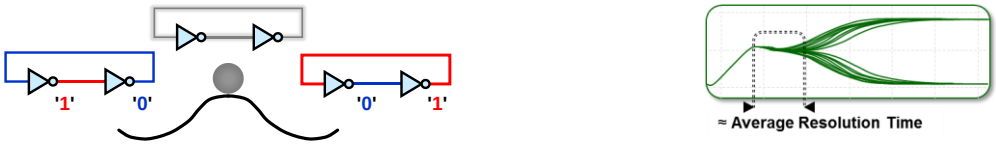
Teoreticky se budou překlápět věčně, prakticky ne, za nějaký čas se ustálí buď na '0' a '1', či '1' a '0' díky obvodovým drobným nesymetriím. Nevíme však v jakém. A jejich oscilace zatěžují napájecí přívod a generují rušení na zemnicích spojích, viz vodní model na straně 66.

Hradla ale nekmitají až do '0' a '1', ale oscilují poblíž rovnovážného metastabilního stavu. Když si ve smyčce nakreslíme napěťové průběhy invertorů, viz 4.9 str. 73, pak uvidíme po jejich spojení do jednoho grafu (levá strana obrázku), že křivky se protínají v bodu, v němž se vybalancovala jejich napětí vstupů a výstupů. Smyčka je v rovnováze.



Obrázek 143 - Metastabilní bod

Uvízla však v metastabilním bodu, který se často znázorňuje ve formě kuličky na vrcholku kulatého kopce. Pokud ji tam dobře vyvážíme, pak z vrcholu nespadne, aspoň po nějakou dobu. Stačí však nepatrný impulz a skutálí se, vlevo, nebo vpravo. Nevíme předem kam a kdy.



Doba setrvání výstupů v metastabilním stavu se označuje jako *resolution time,* doba rozhodnutí. Závisí na CMOS technologii. U dnes používaných lze její průměrnou hodnotu čekat někde v řádu pikosekund. Někdy se dá změřit osciloskopem, pokud vyvoláme opakované výskyty. Průběhy ukážou, že výstup chvíli osciluje kolem rovnovážného bodu. Dá se odhadnout i z nárůstu doby zpoždění[50], které se prodlužuje právě o dobu rozhodnutí.

Výrobci zpravidla uvádějí v katalozích jen MTBF, *Mean Time Between Failures,* statistický parametr, který se z ní počítá. Jeho vysvětlení je mimo rozsah naší učebnice. Popisuje ho [51].

Námi zapojený RS latch bude v FPGA složený nikoli z hradel, ale z logických elementů stejně tak jako veškerá kombinační logika, která se využije k řízení Set a Reset vstupů. A v té se mohou různým zpožděním na vnitřních cestách generovat hazardy, tedy velmi krátké rušivé pulzy, *glitches*, viz kapitola 4.10.1 na str. 75. A RS latch na ně zareaguje náhodnými změnami jak svých výstupů, tak svého zpoždění při metastabilitě, v níž se jeho výstupní napětí drží

[50] B. Medved Rogina, P. Škoda, K. Skala, I. Michieli, M. Vlah and S. Marijan, "Metastability testing at FPGA circuit design using propagation time characterization," 2010 East-West Design & Test Symposium (EWDTS), St. Petersburg, Russia, 2010, pp. 80-85, doi: 10.1109/EWDTS.2010.5742050..

[51] Jakub Šťastný: Techniky synchronizace asynchronních signálů, ASIC Centrum.cz, 2023.

poblíž středu napájení, tedy na úrovni, kterou snadno ovlivní indukce šumu generovaného jinými zdroji na sekvence úrovně '0' i '1'.

Chceme-li spolehlivý obvod, musíme v něm snížit riziko metastability, což vede na zásadu zdůrazněnou snad v každé odborné publikaci věnované návrhům na FPGA.

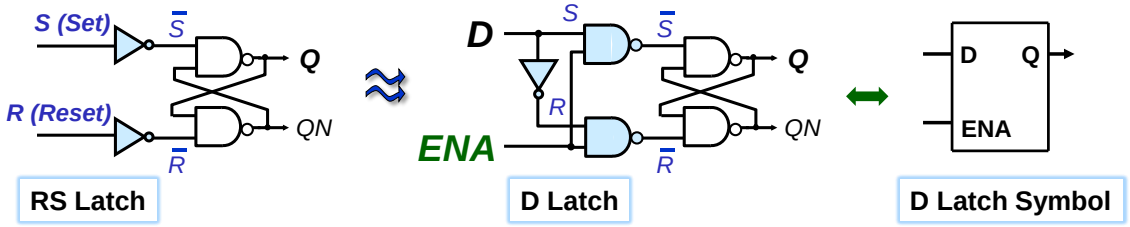
Nesmíme logickými elementy vytvořit obvody typu LATCH !

7.2.2 D-latch z hradel

Obvod D-latch má dlouhý český termín D úrovňový klopný obvod a odstraňuje některé problémy RS latch. Podíváme se na jeho strukturu, neboť v případě našeho chybného návrhu se zapojí z logických elementů ještě častěji než RS latch. Hodí se vědět, co se nám vytvořilo, a kvůli čemu musíme ihned změnit náš kód, aby z obvodu zmizel problematický prvek.

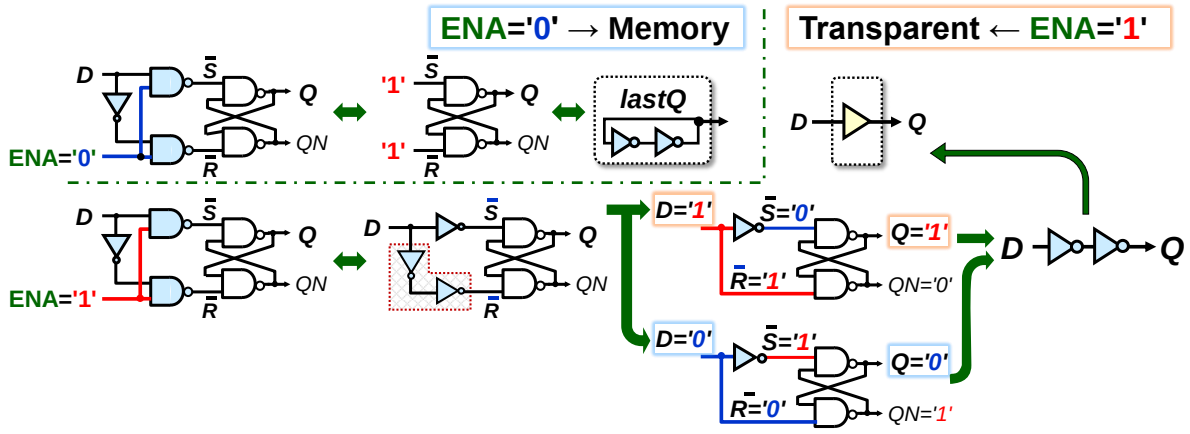
Vyjdeme z RS latch. Vstup Reset ještě odvodíme invertorem od Set, které přejmenujeme na D (Data). Oba vstupní invertory zaměníme za NAND hradla, která budou sdílet další vstup ENA (*Enable*), jímž budeme povolovat klopení.

Pozn.: Někdy ENA se zkracuje na En či E nebo se použije T. Nehodí se jen jeho označení C či CLK. Jde o zavedené symboly hodin synchronních obvodů, jimiž D latch ještě není.



Obrázek 144 - D latch

Nakreslíme si ještě jeho funkční analogie, které si odvodíme připojením logických '0' a '1' na vstup ENA.



Obrázek 145 - Chování D-latch v závistosti na ENA

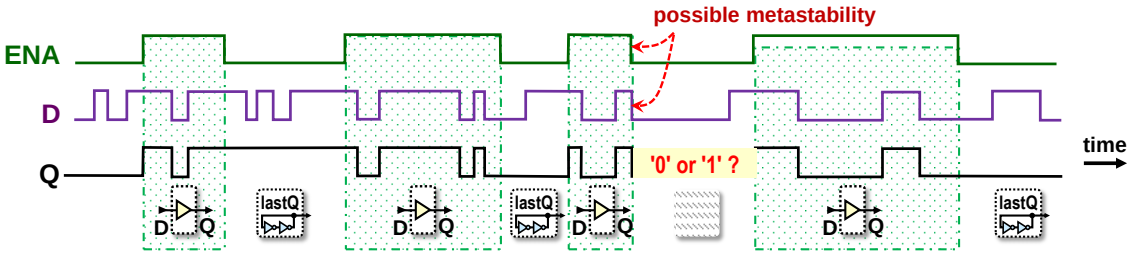
Obvod D-latch se hodnotou vstupu ENA přepíná mezi dvojicí svých funkcí:

- Při ENA='1' se vstupní NAND hradla chovají vůči D jako invertory. Dva za sebou vynecháme dle teorému o dvojí negaci (viz str. 18), a za D postupně dosadíme '0' a '1'. Vidíme, že nyní se kopíruje hodnota D na výstup Q zhruba se zpožděním dvou invertorů. Ty mů-

žeme vyjádřit jako hradlo *buffer*. D-latch se teď chová jako **transparentní** obvod[52], což je i názvem skupiny *transparent latches*, k níž patří.

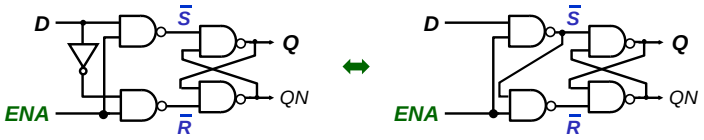
- Při ENA='0' se vnitřní smyčka invertorů odpojí od vstupu D, který ztratí svůj vliv na výstup. RS latch si **pamatuje** svou poslední hodnotu, kterou předává na svůj Q výstup.

Přepínání mezi dvěma režimy využijeme ke konstrukci průběhů signálů. Pro lepší názornost zatím zanedbáme *propagation delay* mezi vstupem D a výstupem Q.



Obrázek 146 - Chování D latch

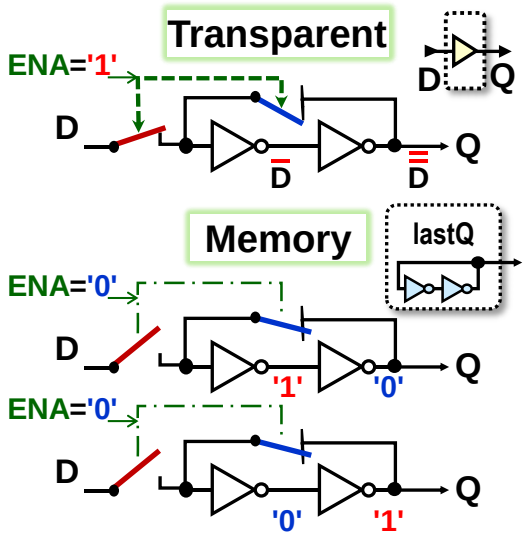
Vstupní část D latch jde zjednodušit skupinovou minimalizací, viz kapitola 5.2.1 na str. 81, a ušetřit invertor. Necháme na čtenáři, aby dokázal funkční shodu obou variant.



Obrázek 147 - Dvě funkčně shodné verze D latch

7.3 D latch na CMOS úrovni

Verze D-latch realizovaná na úrovni CMOS tvoří hlavní stavební prvek nejčastějších synchronních obvodů flip-flop a její vlastnosti se promítnou i do nich. V návrzích je musíme uvažovat. V CMOS verzi se přepíná mezi uzavřenou a otevřenou smyčkou, tedy mezi pamětí a transparencí, užitím *transmission gates* (str. 63), které pracují v protifázi.



Při ENA='1' je smyčka rozpojená a hodnota ze vstupu D prochází transparentně na výstup Q přes dva invertory zapojené v kaskádě, tedy přes ekvivalent hradla typu *buffer*.

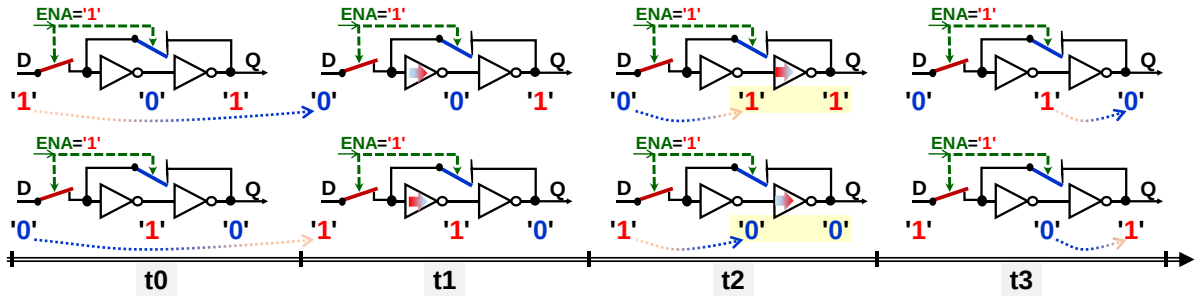
Když přijde závěrná hrana ENA, tedy její změna z '1'→'0', smyčka se odpojí od vstupu D a uzavře se. Zůstane teď ve svém posledním stavu.

Výstup Q bude trvale buď '0' nebo '1' po celou dobu, kdy ENA='0'. Jakmile ENA přejde opět do '1', děj se opakuje.

Obrázek 148 - D-latch - mód transparentní a paměťový

[52] Někde se D-latch alternativně nazývá též *gated latch*, neboť jeho klopení blokují vstupní hrdla.

Důležité je chování otevřené smyčky, tedy při ENA='1'. Nechť tpd je zpoždění jednoho invertoru, pak změna vstupu D v čase t0 se bude šířit postupně. V časech t0 až t3 nastanou děje:

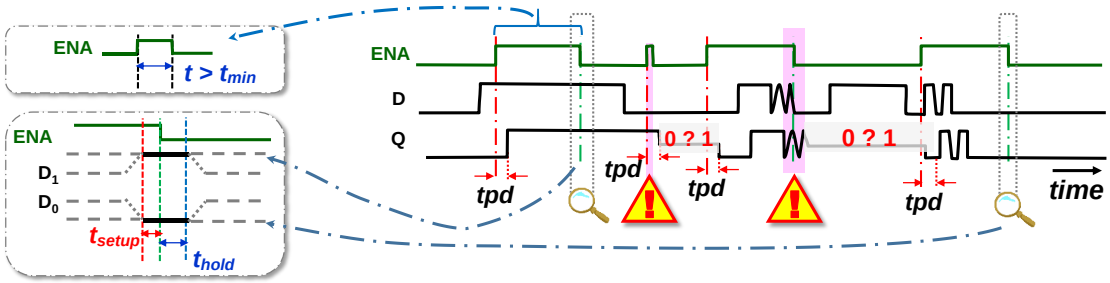


- **t0** - těsně před změnou D má otevřená smyčka ustálený stav;
- 0<**t1**<tpd - vstup D sice změnil svou hodnotu na opačnou, ale nová se teprve šíří skrze levý invertor. Na jeho výstup dorazí až za tpd.
- tpd<=**t2**<=2*tpd - levý invertor se již překlopil. Smyčka je v dočasném pracovní mezistavu, při němž oba její invertory mají shodné hodnoty výstupů, oba jsou buď v '0' nebo '1'. Nyní se smyčka nesmí rozhodně uzavřít přechodem ENA do '0', aby nedošlo k její metastabilitě.
- **t3**>=2*tpd - smyčka již získala ustálený stav, a tak smí už přijít závěrná hrana ENA, '1'→'0', neboť ENA='0' ji již korektně přepne na paměťovou konfiguraci.

CMOS D-latch má opět rizikovou oblast kolem závěrné hrany ENA, jen dva různé pracovní mezistavy, ale oba hrozící metastabilitou. Lze stanovit dvě nutné podmínky, v nichž konkrétní časy závisejí na použité technologii a výrobci je udávají v dokumentaci.

1. Puls ENA musí trvat nejméně dobu t_min, aby smyčka měla čas se ustálit.
2. V okolí závěrné hrany ENA se vstup D nesmí měnit v relativním intervalu vymezenému časy **setup** (předstih) a **hold** (podržení). Nebývají stejné. Setup má u některých typů někdy i zápornou hodnotou, neboť chvíli trvá, než se aktivuje uzavření smyčky, pak se vstup D smí měnit i po přechodu ENA z '1' do '0' po nějakou pikosekundovou dobu.

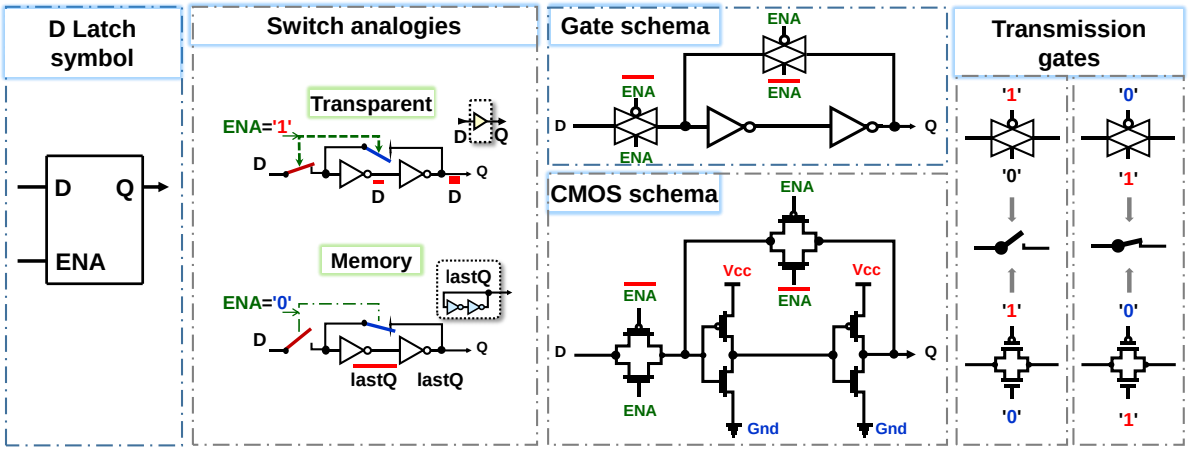
Podmínky ukazuje následující obrázek.



Obrázek 149 Podmínky časování a důsledek jejich porušení

Riziko metastability vyplývá ze samotného principu smyčky a nelze ho odstranit, jen mu předejít dodržením dob t_setup, t_hold a t_min dle katalogů výrobců.

Přepínače budou na úrovni obvodu *transmission gates*, což vede na následující přehled zapojení CMOS D-latch:

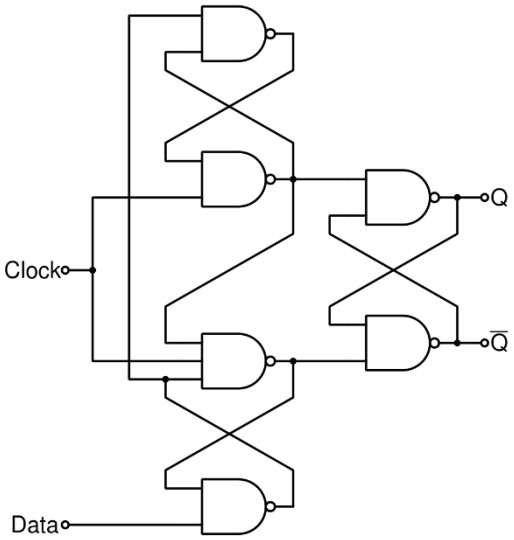


Obrázek 150 - Přehled zapojení D-Latch

7.4 Klopný obvod DFF - Data Flip-Flop

DFF, *Data Flip-Flop*, ovzorkuje svůj vstup D s hranou hodinového signálu. Nemá již transparentní režim D-latch, který při ENA='1' přenášel vše ze svého vstupu na výstup.

Existuje řada struktur DFF. Pro zajímavost uvádíme jednu na obrázku vpravo, převzatého z Wikimedia. Nazývá se *Earle latch* podle svého autora.



Obrázek 151 - DFF struktury Earle Latch

Vnitřně se skládá za tří RS latch, přičemž vždy jeden ze dvou vstupních RS se drží ve stavu svých výstupů v logické '1', jemuž se nepřesně říká zakázaný.

Obvod se dlouho používal v TTL logice, v níž se vyráběl pod kódovým označením integrovaný obvod 74, a to řadou výrobců, i českou firmou Tesla jako typ MH7474.

Nabízí četné výhody, ale potřebuje ke své realizaci 6 NAND hradel. Dnes se kvůli tomu dává přednost úspornějším zapojením.

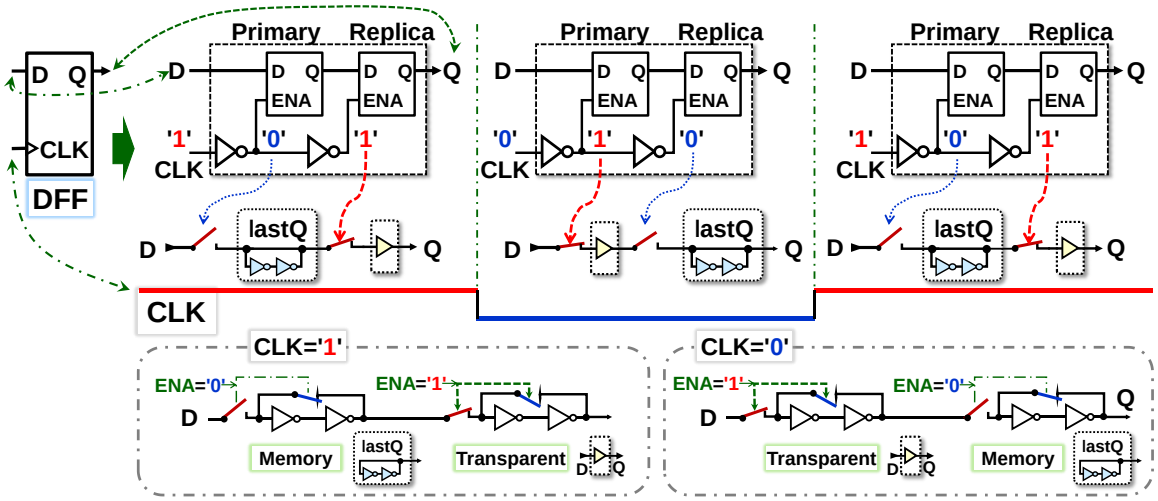
Drtivá většina používaných DFF vzorkuje vstup buď vždy jen na náběžnou hranu hodin, nebo pokaždé pouze na sestupnou.

Na náběžnou hranu hodin reaguje **i nejvíce používaný DFF**, který vznikne kaskádním spojením dvou D-latch pracujících v protifázi. Jeho struktura se půl století nazývala zavedeným termínem Master-Slave, který dnes se již pokládá za společensky nepřijatelný. Technikům se tak vytratil jednoznačný pojem. V době psaní učebnice se náhradní pojmenování dosud neustálilo[53]. V učebnici D-latch označíme **Primary** a **Replica**.

Vstupy ENA obou D-latch se ovládají dvojicí invertorů hodin CLK. První z nich jednak odděluje nitro obvodu a jednak u distribuce hodinového signálu sníží její zatížení, *fan-out*.

[53] Přehled navržených náhradních názvů lze nalézt na https://en.wikipedia.org/wiki/Master/slave_(technology)

Předpokládejme, že hodiny CLK jsou na začátku v '1'. Primary D-latch má svůj ENA invertovaný vůči nim a jeho smyčka setrvává v režimu paměti, v němž drží svou poslední hodnotu. Replica D-latch se řídí vstupem ENA v polaritě shodné s CLK. Její smyčka je rozpojená a transparentně kopíruje výstup Primary D-latch na výstup Q obvodu DFF, viz následující obrázek.



Obrázek 152 - Princip DFF

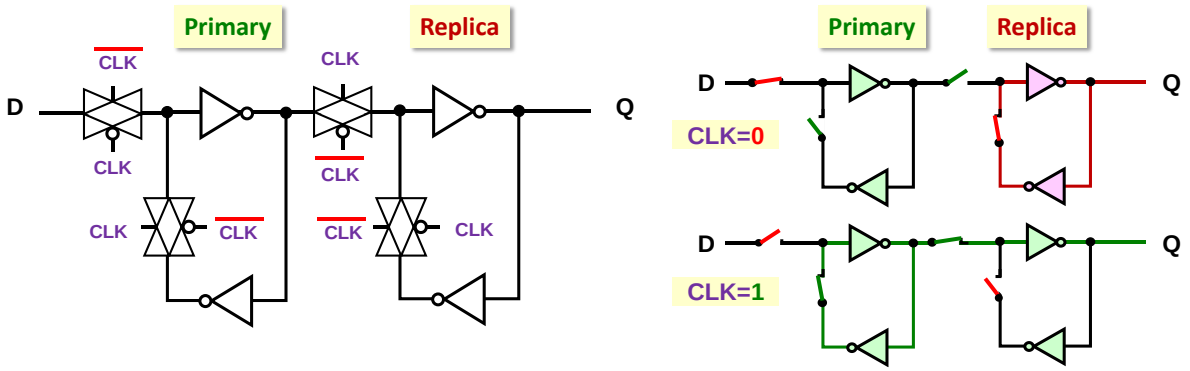
Přejde-li CLK do '0', Primary D-latch se připojí ke vstupu D obvodu DFF, ale ve stejném okamžiku se Replica D-latch oddělila od něho a přešla do módu paměti. Pořád posílá svůj poslední stav na výstup Q obvodu DFF, tedy hodnotu naposledy kopírovanou z Primary D-latch.

Při náběžné hraně, tedy při přechodu CLK z '0'→'1', se Primary D-latch odpojí od vstupu D obvodu DFF a podrží si ve své smyčce jeho poslední hodnotu. Tu však Replica D-Latch nyní kopíruje na výstup, jelikož si rozpojila svou smyčku, takže je v transparentním módu. Její výstup se projevuje jako ovzorkování hodnoty vstupu D obvodu DFF na náběžnou hranu CLK.

Pozor, právě při náběžné hraně CLK může v Primary D-latch dojít k metastabilitě, pokud se její smyčka invertorů nestihla ustálit, protože se nedodržely časy setup a hold na vstup D a ten se měnil v okolí náběžné hrany CLK, kdy Primary D-latch dostala závěrnou hranu svého ENA.

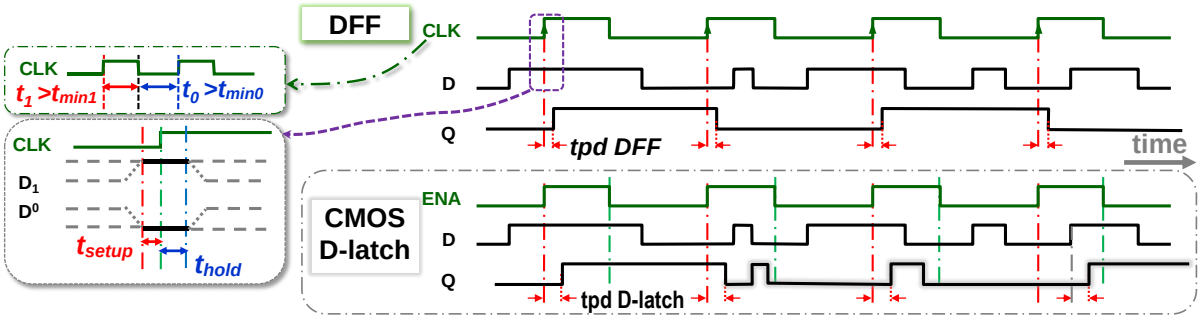
Poznámka: Sestupná hrana CLK nepotřebuje časová omezení, i když Replica D-Latch si při ní uzavírá svou smyčku. Ta je avšak rozhodně ustálená, neboť před tím kopírovala jen stabilní výstup Primary D-latch, který byl tehdy v paměťovém režimu.

Vazba mezi stupni DFF, kterou použilo principiální schéma nahoře, by zpožďovala výstup Q o čtyři invertory. Kvůli tomu se oba D-latch spojují přes své středy. Primary latch posílá negovaný výstup, ale Replica latch ho znovu invertuje, takže výsledek získá správnou polaritu.



Obrázek 153 - Skutečné propojení Primary a Replica

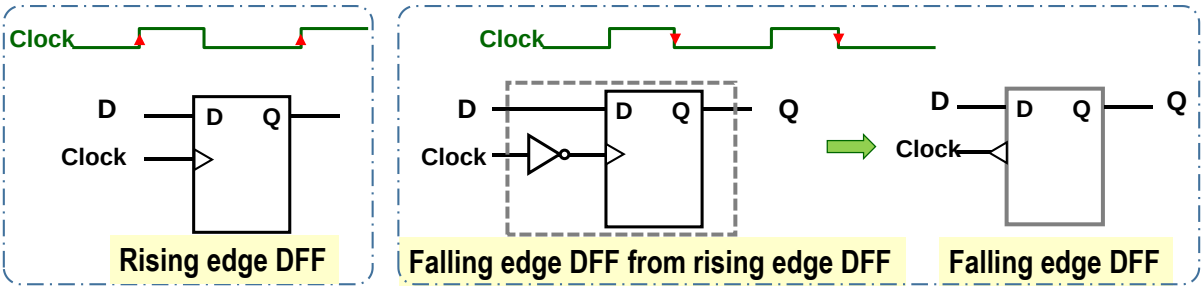
Činnost DFF nejlépe nastíní průběh signálů v následujícím obrázku. Spolu s ním uvedeme, jak by stejné vstupy působily na D latch. Jeho odlišné výstupy se zvýraznily. Navíc je pomalejší. V DFF se při náběžné hraně hodin přenese hodnota uložená v Primary D-latch přes jediný invertor v transparentní Replica D latch na výstup Q, zatímco samostatný D-latch se za stejné situace přepíná svým ENA do transparentního módu, a nová hodnota se z jeho vstupu D šíří přes dva invertory.



Obrázek 154 - Srovnání chování D-Latch a DFF

Ve schématech se vstup hodin DFF často označuje trojúhelníčkem, který směřuje do značky při náběžné hraně, *rising edge*. Naopak se orientuje hrotem ven, pokud DFF reaguje na sestupnou hranu, *falling edge*. Normalizované označení se však často nedodržují, ale bývá zavedeným zvykem přidělit hodinovému vstupu zkratku CLK či CLOCK, či aspoň C. Název jeho vstupu se někdy i vynechává. Značka trojúhelníku ho jasně specifikuje.

Inverzí polarity hodin lze z obvodu citlivého na náběžnou hranu udělat sensitivní na sestupnou, či obráceně. Stačí přidat invertor před hodinový vstup, jak ukazuje obrázek dole.



Obrázek 155 - Značka DFF citlivého na náběžnou/sestupnou hranu

Existují i zapojení DET DFF (Dual Edge Triggered Data Flip-Flop) klopného obvodu citlivého na oba typy hran hodin, jak na náběžnou, tak na sestupnou hranu, ale mají vyšší složitost. Drtivá většina FPGA nabízí uživateli jen DFF citlivý na jeden typ hrany s vnitřním uspořádáním Primary-Replica, který na úrovni CMOS vychází nejjednodušeji.

> **Výstup DFF nemá hazardy.** Vždyť se ho tvoří smyčky invertorů, v nichž nevznikají. Můžeme ho využít jako **hodiny dalších obvodů** za předpokladu, že jsme dodrželi časové podmínky na průběhy jeho vstupních signálů, čímž jsme vyloučili metastabilitu jeho Primary smyčky.

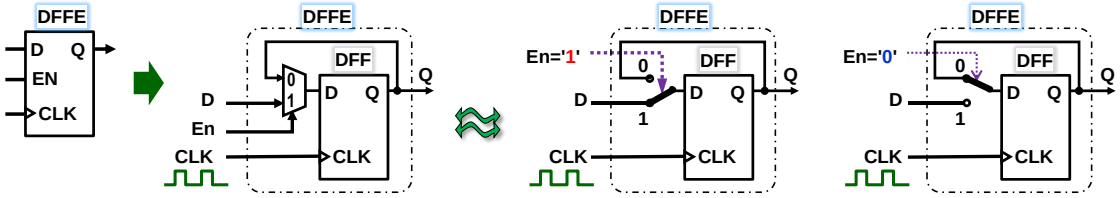
Musíme si ale dávat pozor na překročení **hodinové domény**. Tak se označuje skupina obvodů, které jsou synchronizované jedněmi hodinami. Přenos signálu z jedné hodinové domény do druhé hrozí metastabilitou, a tak si žádá pečlivé návrhy. Příklad řešení uvedeme hned v kapitola 7.4.2 na str. 136.

7.4.1 Doplnění DFF o Enable a asynchronní nulování

Do rozvodu hodin se nesmí vkládat kombinační logika, jak jsme uvedli již v kapitole 4.10.1 na str. 75. Výjimku mají jen hradla typu invertor a *buffer*. A DFF mění svůj stav pouze na náběžné hraně hodin. Jak zajistíme, aby se neklopil, když nechceme? Chová se jako paměť a u té potřebujeme její inicializaci po zapnutí napájení. Jde o dvě nutná vylepšení:

1. Musíme potlačit klopení DFF, aniž mu zablokujeme hodiny.
2. Hodí se asynchronní vstup, který podrží smyčky DFF ve známém stavu během celé doby, kdy krystalové oscilátory teprve nabíhají po zapnutí napájení, a tak dosud negenerují hodinové signály, což někdy může trvat i desítky milisekund.

První vylepšení splníme snadno. Pokud chceme, aby DFF nezměnil svůj výstup Q, i když trvale dostává hodiny, tak jeho výstup Q přivedeme na vstup D. DFF si nahraje jeho hodnotu, kterou již má v sobě. Invertory smyček si dál ponechají své původní stavy, tudíž si neřeknou o dynamický odběr energie při překlopení, viz kapitola 4.8 začínající na str. 66.

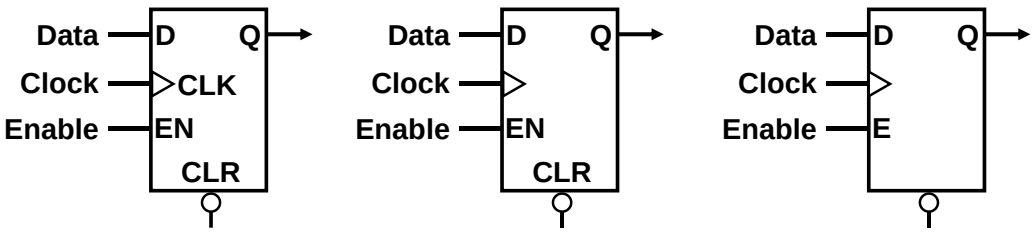


Obrázek 156 - Klopný obvod DFFE

Nový obvod se označuje DFFE, DFF s Enable, a při EN='1' funguje shodně s DFF. Při EN='0' nahrává stávající hodnotu svého výstupu Q. Za cenu přidání multiplexoru se zvýšila univerzálnost, a tak DFFE bývají častým prvkem logických elementů FPGA všech výrobců.

> Signál EN multiplexoru vybírá, která hodnota se připojí na D vstup vnitřního DFF, a tak se rovněž nesmí měnit kolem aktivní hrany hodinového signálu v intervalu vymezeném časy setup a hold z katalogu výrobce.

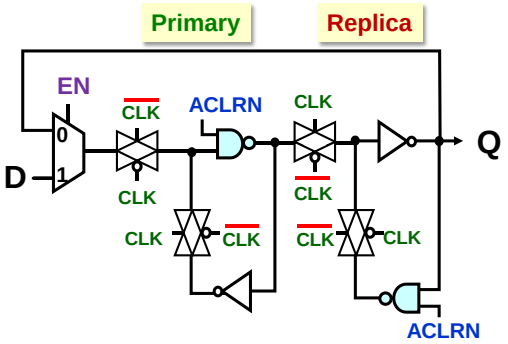
K inicializaci po zapnutí napájení se přidává vstup s asynchronním chováním, který ovlivní výstup Q okamžitě a nezávisle na hodinách CLK. Označuje se CLR, *Clear*. Občas má sufix N, tedy **CLRN**, jímž se zdůrazní, že je aktivní v '0'. Nejpřesněji by se k němu hodila zkratka ACLRN, tedy *Asynchronous Clear Negative*, jíž budeme dávat přednost. Ve schématech se však častěji používá kratší **CLR** a mnohdy se jen vyznačí polohou na dolní části značky DFFE, respektive i u DFF.



Obrázek 157 - Některé varianty schematické značky DFFE

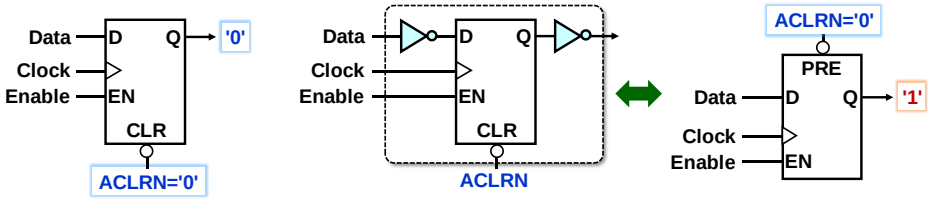
Nevyužitá ACLRN a EN vstupy se připojují na '1', což ve většině případů provede automaticky vývojové prostředí. V logických elementech jsou k jejich deaktivaci i zabudované prvky.

Do DFF se přidá asynchronní nulování tím, že se v každé jeho smyčce nahradí jeden vhodný invertor hradlem NAND. Ty se pak při ACLRN='1' chovají vůči svému druhému vstupu jako invertory a nezmění funkci DFF. Za ACLRN='0' budou výstupy mít v logických '1', což v obvodu nastaví takové vnitřní hodnoty, aby se Q='0' bez ohledu na stavy vstupů CLK a EN.



Obrázek 158 - Asynchronního nulování

Většina FPGA má ve svých logických elementech jen klopné obvody, které zahrnují asynchronní nulování. Jejich nastavení na '1' se provede negací výstupu a vstupu. Výsledný obvod se pak chová, jako kdyby se inicializoval do '1', ale bude nepatrně složitější.

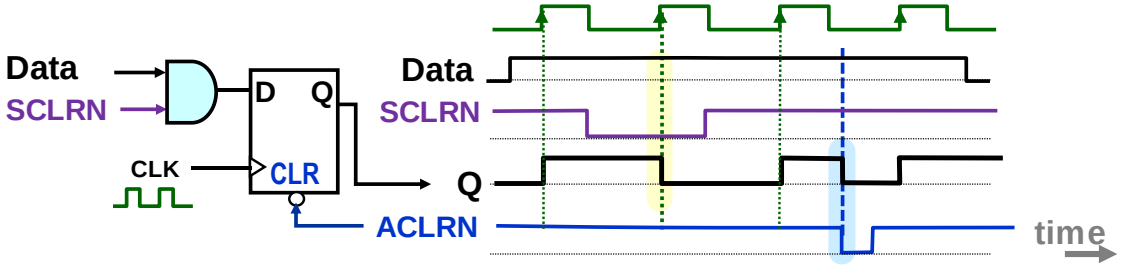


Obrázek 159 - Změna inicializace DFFE obvodu

Pokud se podíváme do katalogu výrobce a zjistíme si, co vše nabízí DFF/E logického elementu, pak můžeme upřednostňovat jeho asynchronní nastavení na hodnotu pro něj přirozenější.

Do FPGA s SRAM se po zapnutí napájení navíc zpravidla nahrává konfigurace z přídavné paměti typu Flash a u mnohých typů se uvádí, že všechny klopné obvody budou ve výchozím nulovém stavu. Moderní FPGA často dovolují u některých zadat i výchozí nastavení na '1'.

Pokud žádáme i nouzový reset, i tak v tomhle případě všechny DFF/E nepotřebují nulování, třeba děliče frekvence. Po nějaké době se samy dostanou do správného stavu odkudkoli. Doporučuje se asynchronní inicializace nahrazovat synchronními, které mají širší možnosti.



Obrázek 160 - Synchronní a asynchronní inicializace

Synchronní inicializace se provede nahráním nové hodnoty, zatímco asynchronní okamžitě, nezávisle na hodinách. Časové podmínky na průběh ACLRN si odvodíme sami z chování smyčky invertorů:

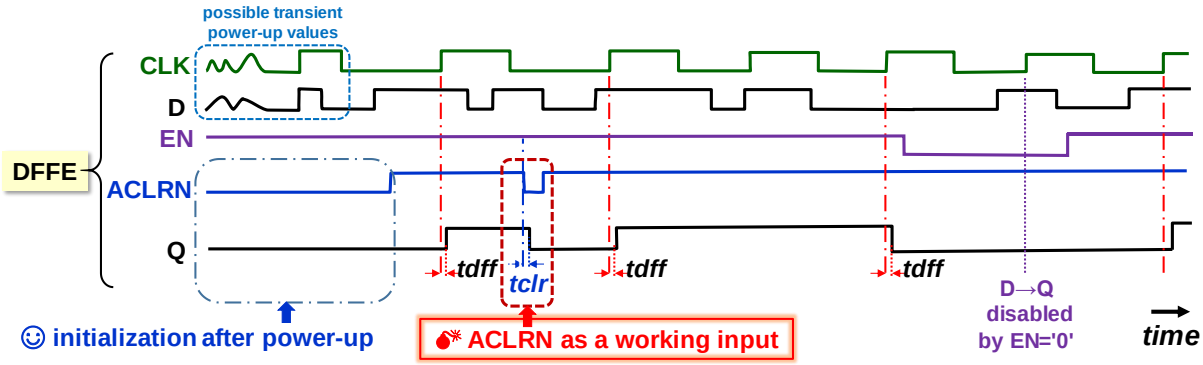
- Asynchronní ACLRN musí v logické '0' zůstat déle než zpoždění dvou invertorů, jinak hrozí, že smyčka, která je právě v paměťovém módu, se neustálí a zůstane v pracovním mezistavu, v němž mají její členy shodné výstupy. Může pak dojít k její metastabilitě. **Podmínka vylučuje generování ACLRN logickými funkcemi**, neboť mohou mít hazardy. V FPGA jsou jimi všechny s výjimkou operací NOT a *buffer*.

- ACLRN nesmí přejít z '0' do '1' v okolí náběžné hrady hodin CLK, pokud by se poté posílala nenulová hodnota vstupu D do Primary D-latch. Kdyby se její invertory nestihly ustálit, opět by hrozila metastabilita.

Příklad funkce obvodu DFFE, který má ACLRN, demonstruje obrázek dole. První stav ACLRN='0 držel obvod ve výchozím stavu během doby náběhu napájení a jde o správné užití.

Následující pulz ACLRN='0 se označil za časovanou bombu, jíž bude v případě, že se jeho pulz generoval logickou funkcí z jiných vnitřních signálů!

Problémem zde budou také **rychlé smyčky**. Vzniknou v tom případě, když logická funkce generující ACLRN='0' závisí i na hodnotách výstupů obvodů, které jím nulují. Jakmile ACLRN ovlivní některé výstupy a ty změní hodnoty, pak samo sebe zruší, tj. ACLRN= '1', což ukončí nulování. A nemuselo se nutno stihnout u všech prvků, na něž se ACLRN rozvádí☹



Obrázek 161 - Klopný obvod DFFE s asynchronním nulováním

Na webu najdete četná zapojení, v nichž se asynchronní vstupy používají jako pracovní a nulovací signály se v nich generují logickými funkcemi. Podobná schémata se vztahují k pomalejší bipolární TTL logice, která měla zpoždění od cca 7 ns výše, a tak i nižší citlivost na rušivé pulzy a šlo asynchronními vstupy využívat jako běžné pracovní.

Zpoždění dnešních hradel se měří v desítkách pikosekund. Konstrukce s asynchronní pracovními vstupy lze sice tvořit i s nimi, mají totiž rychlejší odezvy, ale dělá se to jedině v extra pečlivě odladěných návrzích realizovaných přímo na úrovni CMOS, v nichž se vyloučí hazardy, zdroje krátkých nedovolených pulzů. Obecně se to však nedoporučuje.

Asynchronní nulování nebo nastavení
se v FPGA používá výhradně k inicializaci celého obvodu,
což potřebujeme třeba po zapnutí napájení nebo jako nouzový restart.

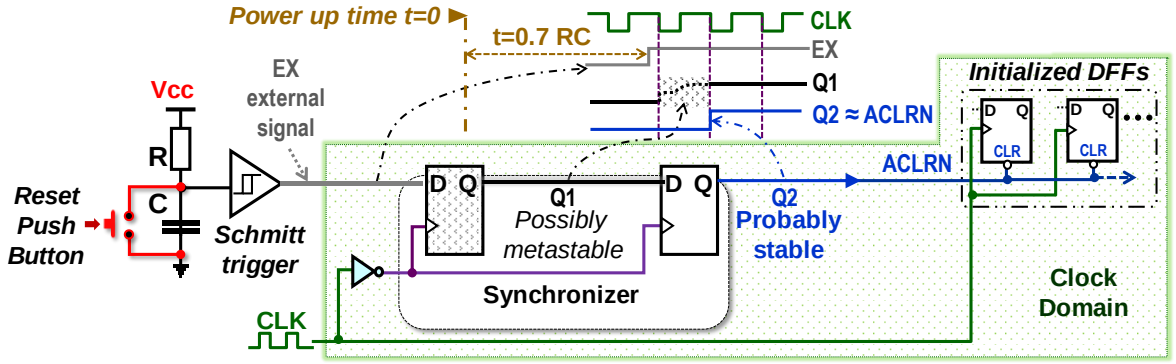
Připojíme reálnou historku. Jeden náš tvrdohlavý student prohlásil na počátku dnešního tisíciletí, že asynchronní vstupy využije jako pracovní i v FPGA a vyřeší s nimi zadanou zápočtovou úlohu. Prý mu vždy spolehlivě fungovaly. Nechali jsme ho, není nad vlastní zkušenost.

Nosil s sebou i plošný spoj s funkčním obvodem realizovaným pomalou TTL logikou a dva měsíce experimentoval s vytvořením její analogie v rychlém FPGA, než sám sebe přesvědčil, že tudy cesta opravdu nevede. A díky tomu nestihl poslední termín odevzdání své zápočtové práce. Za stvoření poučné příhody se však dočkal odměny v podobě individuálního prodloužení bez postihu:-)

7.4.2 Synchronizéry a tvorba ACLRN

DFF potřebuje, aby se jeho vstupy D a EN neměnily v okolí náběžné hrany, což můžeme zajistit u signálů, které si sami generujeme v obvodu, ale vnější neovlivníme. Jejich změny nejsou synchronizované s hodinami, a přesto vstupují do skupiny obvodů, které závisí na jedinných hodinách. Taková se nazývá *clock domain (cz: hodinová doména)*. Překračování její hranice si vyžaduje synchronizér, též zvaný resynchronizační obvod.

Asynchronní ACLRN bude vždy externím vstupem, buď z nějakého tlačítka či RC článku, jehož přechod do ‚0' a ‚1' se může objevit kdykoli. Chceme-li mít jistotu, že nevyvolá metastabilitu svým ukončením v nevhodném okamžiku, upravíme ho synchronizérem. Vytvoříme ho nejméně ze dvou DFF v kaskádě..



Obrázek 162 - Synchronizér na vstupu hodinové domény

Ke generování ACLRN využijeme známý RC článek. Jeho kondenzátor C se nabije na polovinu napájecího napětí za čas daný řešením diferenciální rovnice, což vede t=0.7 RC, vzorec známý z kapitoly 4.8.3 na str. 69. Lze ho využít i k havarijní inicializaci jako tlačítko reset, které vybije kondenzátor a ten se znovu nabíjí.

Pomalý náběh nárůstu napětí na kondenzátoru vytvarujeme Schmittovým klopným obvodem, viz dále, který nám zaručí čistou výstupní '1'.

Výstup EX překračuje hranici hodinové domény, a tak ho vedeme přes kaskádu DFF. První DFF může ještě mít někdy metastabilitu, pokud EX skončí v nevhodný okamžik. Předpokládejme, že se z ní zotaví do stavu '1'. Pokud by tak neučinil, nahraje si '1' v dalším taktu hodin.

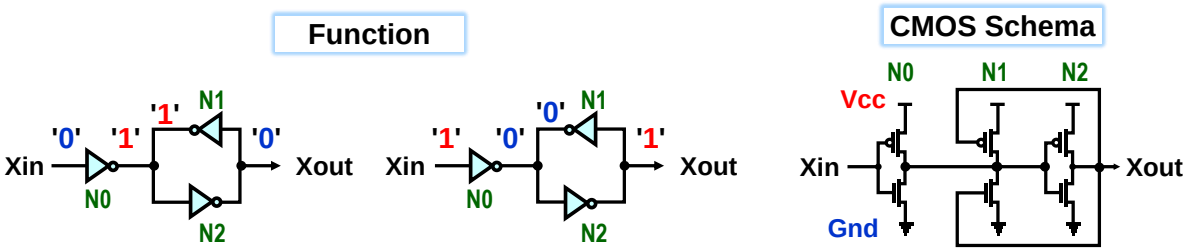
Druhý DFF bude s vysokou pravděpodobností mít již čistý výstup. Ten využijeme jako signál ACLRN.

Oba obvody DFF pracovaly na závěrnou hranu hodin, a tak ACLRN přejde do '1' mimo náběžnou hranu. Rozvedeme ho na všechny DFF, které na ni reagují, a potřebují inicializaci.

Ve výkladu jsme zmínili i Schmittův klopný obvod, *Schmitt trigger*, který patří k běžně vyráběným obvodům. Chová se jako napěťový komparátor s hysterezí, neboť potřebuje vyšší vstupní napětí ke svému přechodu do logické '1' než k návratu z ní do logické '0'.

Zapojuje se mnoha různými způsoby včetně operačních zesilovačů[54]. Můžeme si ukázat jedno jeho CMOS konstrukci, která využívá nám známou paměťovou smyčku invertorů.

[54] Viz třeba https://en.wikipedia.org/wiki/Schmitt_trigger

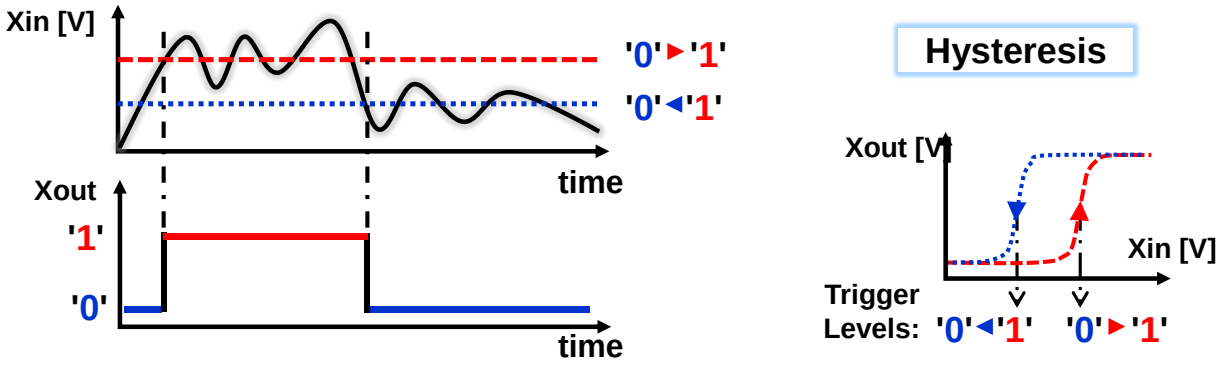


Obrázek 163 - Příklad CMOS zapojení Schmittova klopného obvodu

Vstupní invertor N0 přepíná smyčku jejím zkratováním, což vypadá sice divně, ale jen na první pohled[55]. Smyčka se totiž vmžiku převede do nového stavu, v němž již má výstup svého invertoru N1 shodný s výstupem ovládacího N0.

Na výstup N0 působí tak stejný i od N1 ze smyčky, což vyvolá posun prahových napětí vstupních CMOS v N0. Ta se pak o něco zvýší/sníží. Bude to tedy jako nutnost působit větším napětím na vstup N0, abychom vyvážily vyšší logické zatížení na protějším konci „dvouramenné houpačky“ a převážili ji do opačného stavu, od něhož se i nově nastaví smyčka N1 a N2.

Dostali jsme obvod s hysterezí, který nejen konvertuje hodně rozvlněné vstupní signály na čisté průběhy, ale současně i akceleruje i klopení výstupu Xout.

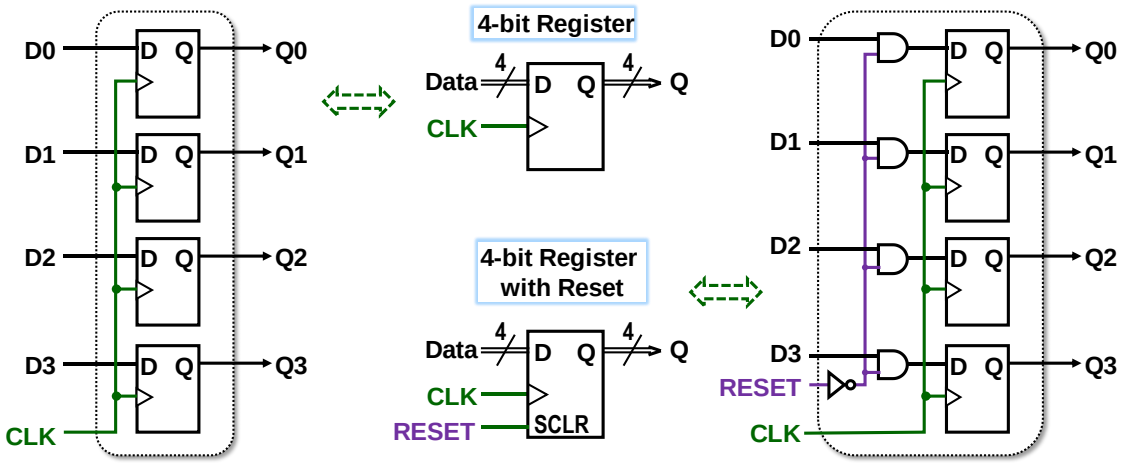


Obrázek 164 - Přiklad činnosti Schmittova klopného obvodu

[55] Podobným zkratem smyčky se nastavují i buňky v SRAM pamětech.

7.5 Registry a čítače

Z obvodů DFF či DFFE lze budovat rozličná zapojení, z nichž nejjednodušším je registr, který paralelně ukládá právě tolik bitů, na kolik si ho navrhneme.

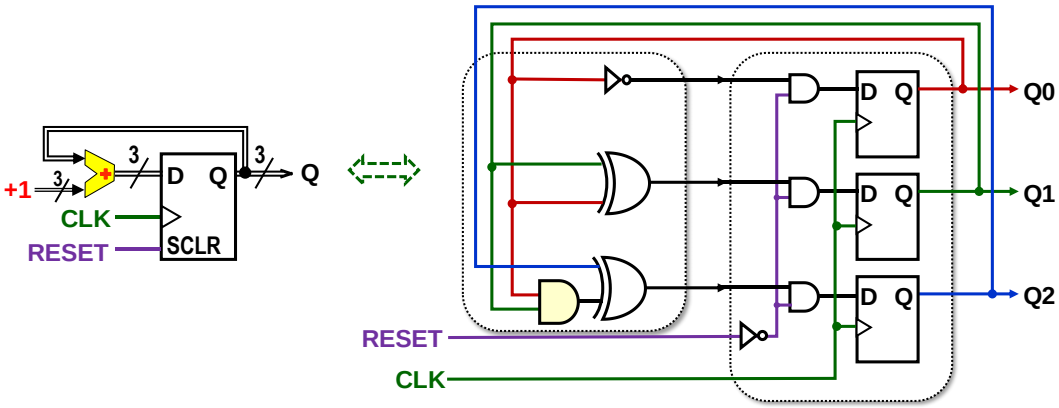


Obrázek 165 – 4-bitový registr

Můžeme k němu přidat i synchronní nulování, pro něj se ustálilo označení RESET, případně SCLR, aby se odlišilo od asynchronního nulování CLEAR. I to by šlo doplnit, kdyby propojily asynchronní vstupy DFF (na obrázku vynechané) a vyvedly se ven.

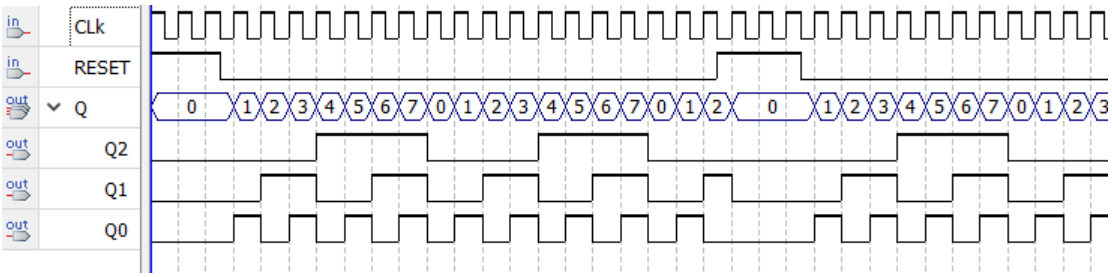
Jak jsme ale již zmínili dříve, asynchronní inicializace by se měly používat jen v kritických částech. *Pozn. Návrhová prostředí nám budou občas konvertovat naše asynchronní inicializace, když zjistí, že jsou zbytečné, na synchronní.*

Z registru snadno zapojíme čítač, když před něj vložíme sčítačku +1. Vytvoříme si třeba tříbitový čítač, který jsme použili v blikajícím hadovi, Obrázek 86 na straně 80.



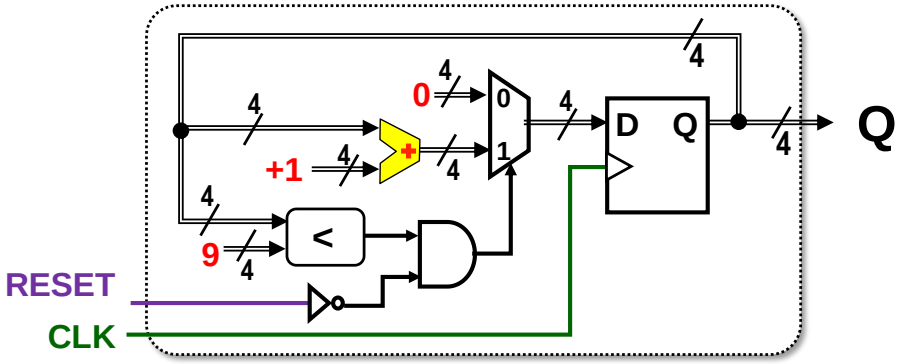
Obrázek 166 - Tříbitový čítač pro blikajícího hada

Samotný čítač běhá v nekonečném cyklu, klopí se na náběžnou hranu. Jeho možné chování ukazuje následující obrázek. Čítače se inicializoval na začátku a poté někdy uprostřed:



Obrázek 167 - Příklad výstupu 3bitového čítače

Chceme-li čítač s kratším čítacím cyklem, pak k 4bitovému registru přidáme sčítačku a komparátor. Nulování si vyřešíme multiplexorem, protože ho potřebujeme ve dvou případech, a to pří dosažení maximální hodnoty a při synchronním reset.



Obrázek 168 - Dekadický čítač

Maximální hodinová frekvence, při níž čítač ještě správně počítá, je určená zpožděním ve smyčce Q na D. Když přijde náběžná hrana hodin CLK, výstup Q se změní. Další náběžná hrana hodin se smí objevit teprve po ustálení sčítačky +1 a logiky kolem ní, když vstup D má již stabilní novou hodnotu.

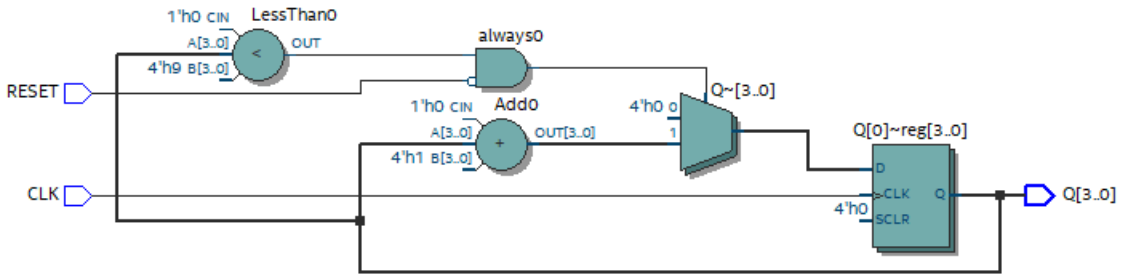
Schéma obvodu jsme tentokrát nerozkreslili na zapojení s hradly, bylo by už příliš složité a ztratilo by názornost. A u obrázků špatně porovnávají jejich verze. I když každé bude znázorňovat stejně fungující obvod, ale může mít jiné grafické rozložením svých prvků.

Jak rostla složitost obvodů, začalo se, někdy před rokem 1980, stále více mluvit o návrhové krizi. Ta iniciovala vývoj jazyků HDL coby textové popisy obvodů.

V HDL jazyce Verilog se dekadický čítač vytvoří mnohem rychleji než schéma:

```
module DecadeCounter(input CLK, output reg [3:0] Q);
always @( posedge CLK )
begin
   if (Q<9 && !RESET)  Q<=Q+1; else Q<=0; end
endmodule
```

Chceme-li vidět, co se nám sestavilo, vývojové prostředí nám nakreslí schéma, ale používá své vlastní značky. V obrázku dole jde úsporné zápisy interního jazyka AHDL[56]. Obrázek dole vyjadřuje shodný obvod s obrázkem nahoře, jen s jinak graficky rozloženými elementy.

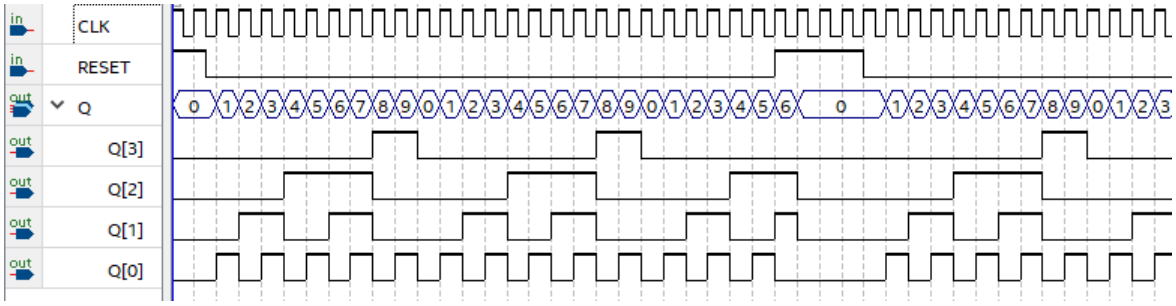


Obrázek 169 - Schéma obvodu vytvořeného z kódu

Zobrazený výsledek odpovídá vnitřnímu meta-schématu sestavenému z našeho kódu. V dalších krocích se bude optimalizovat a rozvrhovat na konkrétní strukturu námi užitého FPGA.

[56] Altera Hardware Description Language

Textová verze obvodu se dá i simulovat, abychom si ověřili, že jsme návrh provedli dobře.



Obrázek 170 - Simulace dekadického čítače

Simulace se vždy hodí. Ve Verilogu se totiž prohřešky příliš nehlídají, předpokládá se, že jeho uživatel dobře ví, co dělá. Mnozí profesionální návrháři ho upřednostňují. Píše se jim v něm rychleji, jiní zase volí přísnější VHDL.

Ve VHDL 2008 bude kód dekadického čítače o něco delší, ale dle našich zkušeností čitelnější pro začátečníky. Dále se zde vkládají nejen knihovny, jejichž volbou lze zvýšit variabilitu, ale používají se v něm i datové typy, což zvýší bezpečnost kódu. Nabízí se i možnost více architektur (vnitřních zapojení obvodu), což se hodí třeba při ladění. V kódu dole je jen jedna.

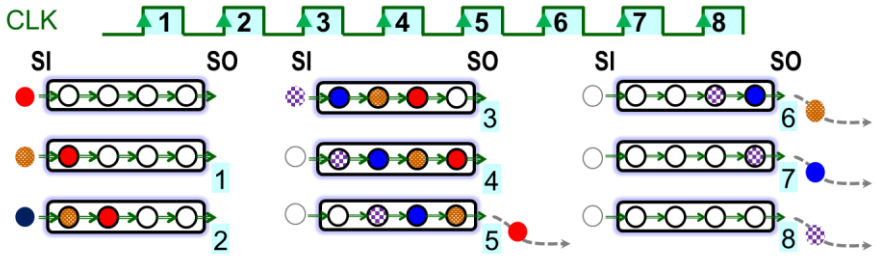
```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity DecadeCounter is
port (CLK, RESET : in std_logic; Q : out unsigned(3 downto 0));
end entity;

architecture RTL of DecadeCounter is
begin
  process(CLK)
  begin
     if rising_edge(CLK) then
          if Q<9 and RESET='0' then Q<=Q+1; else Q<=X"0"; end if;
     end if;
  end process;
end architecture;
```

VHDL kód je upovídanější, ale nepíšeme ho celý sami. Hlavičky knihoven existují v každém VHDL souboru, a tak se kopírují z předpřipravených šablon, které obsahují i prototypy bloků entity a architektury. Delší klíčová slova, jako třeba std_logic, za nás zase vloží automatické doplňování, které dnes bývá běžnou výbavou editorů kódu. Popis obvodu se tak vytvoří jen o něco pomaleji než ve Verilogu.

A doba psaní kódu není rozhodující, jelikož HDL popisy obvodů bývají mnohem kratší, než zdrojové soubory jazyka C. Návrh se nejvíce urychlí, když uděláme minimum chyb v popisu obvodu. Ty se hledají déle než v klasickém programu.

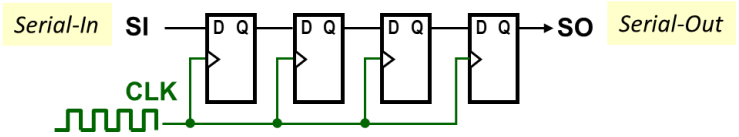
7.6 Posuvné registry

Posuvné registry, *shift registers*, se podobají registrům dat z předešlé kapitoly, avšak jejich klopné obvody DFF jsou propojené do řetězce, výstup jednoho DFF vede na vstup následujícího. Data skrz ně postupně posouvají v taktu hodin podobně jako výrobky na dopravním pásu. Pokud si barevně označíme část vstupních dat, pak s každým taktem hodin, v obrázku dole je to při náběžné hraně, se do registru nasune logická 0 nebo 1, která se právě nachází na vstupu SI (*Serial-In*). Poslední hodnota na výstupu SO (*Serial-Out*) se z něho ztrácí.



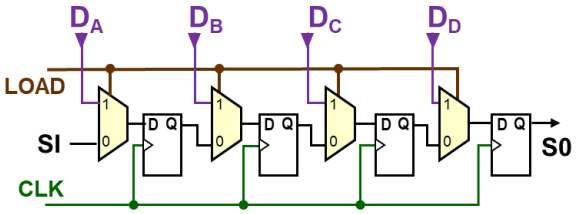
Obrázek 171 - Princip posuvného registru

Obrázek dole ukazuje možné schéma posuvného registru SISO (*Serial-In Serial-Out*), který se může využít i jako zpožďovací linka. Hodnota na vstupu SI se objeví na výstupu SO po počtu taktů hodin daným počtem DFF. Data se posouvají od jednoho DFF k druhému skrz propojky, tedy bez zpoždění, a tak maximální možná frekvence hodin nezávisí na délce posuvného registru. Pro srovnání připomínáme, že u čítačů, viz např. Obrázek 168 na str. 139, klesala se zvýšením počtu bitů kvůli rostoucí době čekání na ustálení sčítačky v jejich zpětné vazbě.



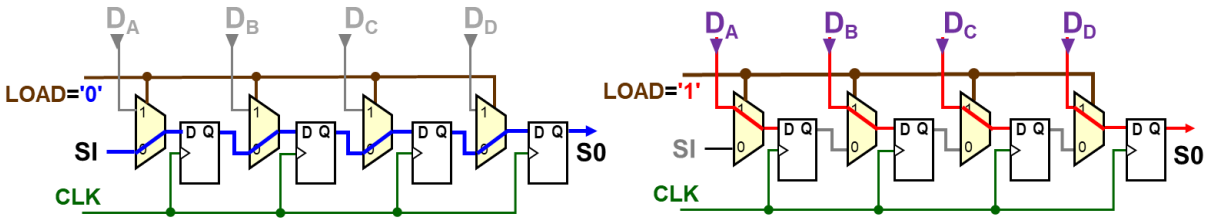
Obrázek 172 - Posuvný registr typu SISO – Serial-In Serial-Out

Srovnáme-li posuvný registr SISO se 4-bitovým datovým registrem, který jsme si ukázali na Obrázek 165 v předešlé kapitole, pak vidíme, že oba se od sebe liší jedině zapojením vstupů svých DFF. Budeme-li je přepínat multiplexory, dostaneme PISO (*Parallel-In Serial-Out)* posuvný registr, který se hojně využívá na sériových linkách, které uvedeme v kapitole 7.7.



Obrázek 173 - Posuvný registr typu PISO

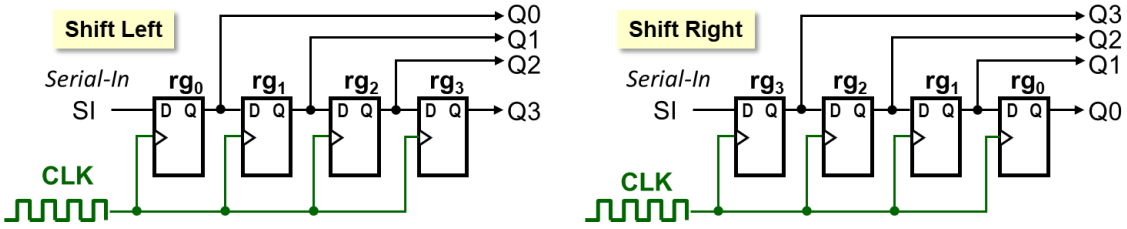
Jeho vstup *Load* mění jeho chování z posuvného registru typu SISO na datový registr:



Obrázek 174 - PISO se mění na SISO posuvný registr či datový registr

Předchozí obrázky nedefinovaly binární váhy dat v klopných obvodech. Zavedeme-li si je, můžeme použít pojmy posun doleva či doprava, neboť oba termíny nezávisí na poloze DFF na schématu, ale jedině na pomyslných vahách výstupních bitů. Jejich přiřazení je na nás.

Vyvedeme-li všechny výstupy ven, získáme posuvný registr SIPO (*Serial-In Parallel-Out*). Posun doleva vynásobí dvěma hodnotu v něm branou jako číslo bez znaménka, zatímco posun doprava provádí její dělení dvěma.



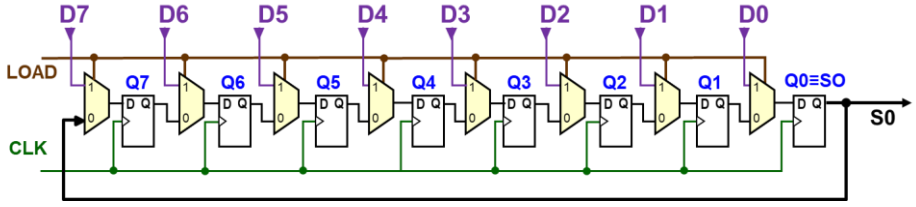
Obrázek 175 - Posuvný registr typu SIPO doleva a doprava

Mějme například ve 4-bitovém posuvném registru bity Q3210 = 0101, které mají dekadickou hodnotu 5, uvažujeme-li je jako číslo bez znaménka, kde bereme Qn jako bit s váhou 2^n. Pokud bude SI (*Serial-In*)='0', pak posun doleva změní registr na Q3210 = 1010, jeho hodnota bez znaménka je nyní 10, tedy 2*5. Další posun doleva s SI='0' ale dá hodnotu Q3210 = 0100 = 4. Došlo k přetečení podobně, jako kdybychom ve 4-bitové reprezentaci čísel provedli násobení 10*2 = 20 = 16+4 = $1\mathbf{0100}_2$. Horní jednička se zde také ztrácí.

Naopak po posunu doprava se v původním Q3210 = 0101 ztratí dolní jednička a hodnota v registru bude nově Q3210 = 0010, dekadicky 2 = 5 *celočíselně dělené* 2.

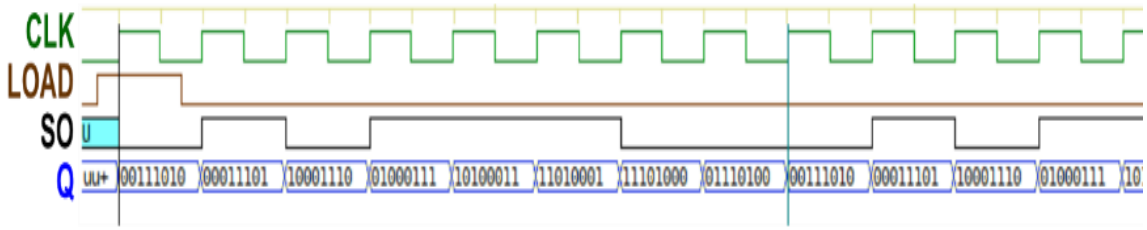
7.6.1 Kruhové posuvné registry a Johnsonovy čítače

Pokud u posuvného registru typu PISO (*Parallel-In Serial-Out*) přivedeme jeho výstup SO (*Serial-Out*) na SI (*Serial-In*) vznikne kruhový posuvný registr, v angličtině zvaný též *Barrel Shifter* nebo *Ring-Counter*.



Obrázek 176 – 8-bitový kruhový posuvný registr vpravo

Kruhové registry se dají využít například ke generování periodických průběhů. Nahrajeme-li hodnotu "00111010"=0x3A do 8-bitového posuvného registru vpravo, ten bude dokola vytvářet signál, který odpovídá cyklickému vysílání písmena A v Morseově kódu.

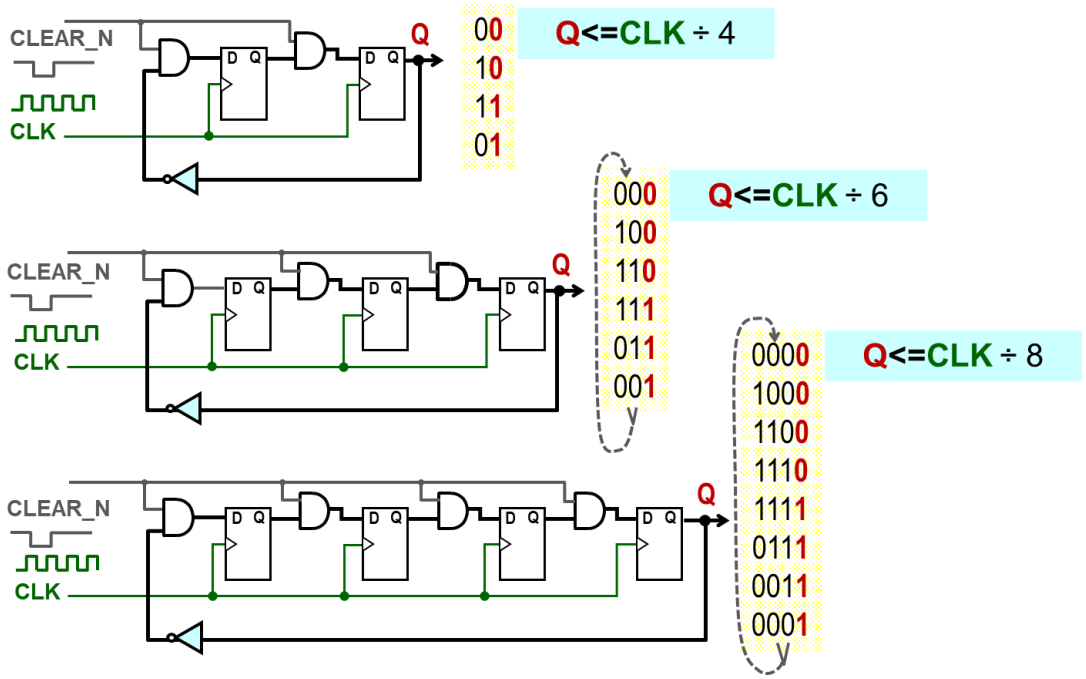


Obrázek 177 - Kruhový posuvný registr vpravo po nahrání "00111010"=0x3A

Všimněte si, že jeho simulace na začátku ukazuje hodnoty 'u', tedy undefined. Až náběžná hrana hodin CLK při LOAD='1' jednoznačně specifikuje jejich stav. Podobné chování nevadí, můžeme přece LOAD generovat při zapnutí napájení k inicializaci registru.

Když v kruhovém posuvném registru zapojíme zpětnou vazbu ze sériového výstupu SO přes invertor na sériový vstup SI, dostaneme velmi rychlý Johnsonův čítač, který je v angličtině známý jako *twisted ring counter*, nebo *switch-tail ring counter*, či *walking ring counter*, eventuálně *Johnson counter*, případně *Möbius counter*.

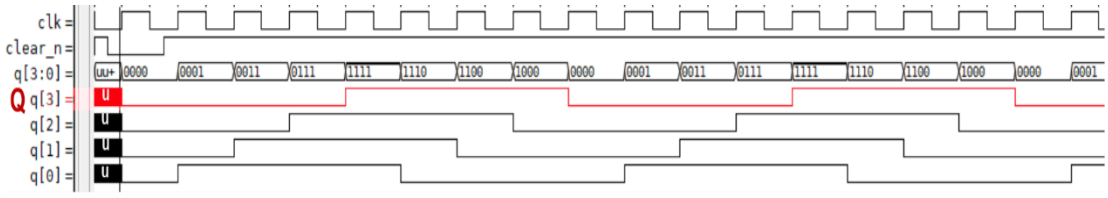
Budou-li jeho registry vynulované po zapnutí napájení, na schématu dole se mažou '0' na vstupu CLEAR_N, pak Johnsonův čítač dokola opakuje sekvenci, která je v Grayově kódu, ale pouze u dvoubitového Johnsonova čítače zahrnuje všechny možné případy.



Obrázek 178 - Johnsonův kruhový čítače

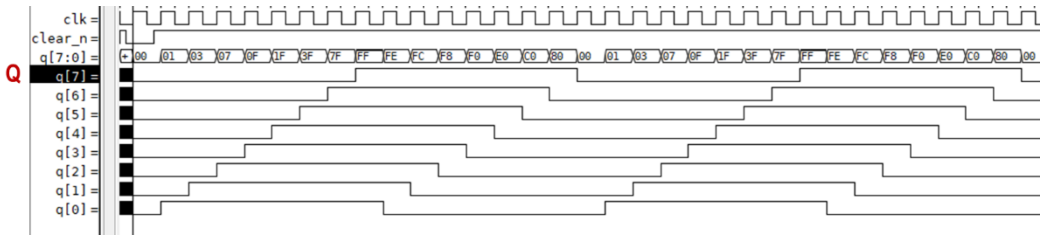
Nezbytná nulování jsou zde synchronní, tedy na hranu hodin při CLEAR_N='0'. Nevyužili jsme asynchronní vstup ACRLN registrů DFF. Synchronní CLEAR_N je výhodnější, lze ho využít jak k inicializaci po zapnutí napájení, tak jako pracovní vstup, viz i kapitola 7.4.1 na str. 133.

Johnsonovy čítače se hodí jako první stupně děličů extrémně vysokých frekvencí, protože posuvné registry neomezuje jejich délka. Vyvedeme-li výstupy ze všech jejich registrů, bude na nich frekvenci vydělenou 2*N, kde N je délka Johnsonova čítače. Výstupy budou ale navzájem fázově posunuté, což se například hodí v některých metodách modulace a demodulace analogových signálu a rekonstrukcích frekvence na sériových linkách, viz str. 148.



Obrázek 179 - Výstup 4-bitového Johnsonova čítače z předešlého obrázku

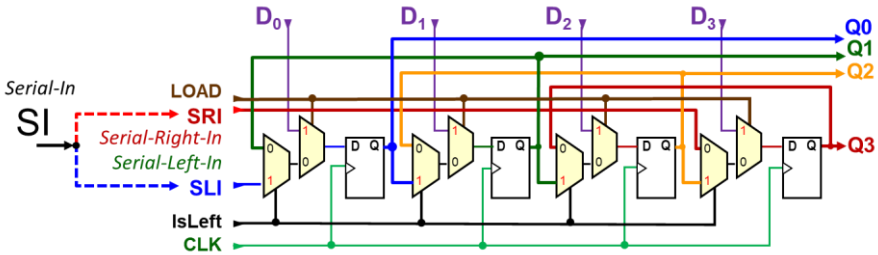
Všimněte si, že výchozí stav registrů DFF není definovaný až do jejich vynulování.



Obrázek 180 - Výstup 8-bitového Johnsonova čítače

7.6.2 Obousměrný posuvný registr a jeho HDL kódy

Pomocí multiplexorů lze měnit i směr posunu a vytvořit obousměrný posuvný registr, z něhož lze snadno udělat i kruhový, když propojíme SLI s Q3 a SRI s Q0. Některé implečmentace mají vnitřně propojené SLI s SRI a vyvádí je ven jako SI. I takové modifikujeme na kruhový registr, když před jejich vstup SI dáme multiplexor, který podle směru posunu přepíná Q3 a Q0.



Obrázek 181 - Obousměrný posuvný registr

Výsledné schéma již ztrácí přehlednost, přesněji řečeno, začíná se podobat změti čar, v níž ani barvení příliš nepomůže, a to jsme nakreslili pouhou 4-bitovou variantu s nahráním dat.

Popíšeme-li však obvod v některém HDL jazyce, jeho kód bude jednodušší a nenaroste ani zvětšením délky posuvného registru, stačí změnit použité číselné meze. Budou-li určené konstantou, bude vhodný pro jakoukoli délku. Výkonná část je v obou kódech zdůrazněná žlutě.

Verilog

```
module shift4BiDir (
   input wire CLK,
   input wire SLI,
   input wire SRI,
   input wire IsLeft,
   input wire Load,
   input wire [3:0] D,
   output reg [3:0] Q );

   always @(posedge CLK) begin
      if (Load) begin Q<=D; end
      else if (IsLeft) begin
         Q <= {Q[2:0], SLI}; // Shift left
      end else begin
         Q <= {SRI, Q[3:1]}; // Shift right
      end
   end

end module
```

VHDL 2008

```
library ieee; use ieee.std_logic_1164.all;
entity shift4BiDir is
port( CLK, SLI, SRI, IsLeft, Load : in std_logic;
      D: in std_logic_vector(3 downto 0);
      Q: out std_logic_vector(3 downto 0):=(others=>'0'));
end entity;
architecture rtl of shift4BiDir is
begin
process (CLK)
 begin
   if rising_edge(CLK) then
      if Load then Q<=D;
      elsif IsLeft then
        Q <= Q(2 downto 0) & SLI; --Shift left
      else
        Q <= SRI & Q(3 downto 1); --Shift right
      end if;
   end if;
 end process;
end architecture;
```

Obrázek 182 - Obousměrný posuvný registr ve Verilogu a VHDL

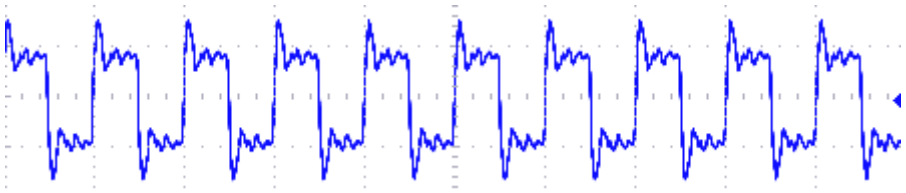
7.7 Přenosy dat

Elektronická zařízení často používají dálkové připojení, a tak učebnici ukončíme zmínkou o několika vybraných technikách přenosu dat z mnoha možných.

Paralelní přenos, kdy datové bity vedeme všechny naráz, je sice "teoreticky" nejrychlejší, ale prakticky funguje jen na kratší vzdálenosti. Běžně se používá uvnitř obvodů, kde propojovací vodiče měří několik milimetrů. I mezi procesorem a pamětmi probíhá paralelní přenos, u grafických karet GPU i v šířce 512 bitů, ale na několik centimetrů mezi přesně definovanými zařízeními, která jsou navíc trvale aktivní. Přenosové charakteristiky linky se mění pomalu, prakticky jen s teplotou zařízení. Přenos se za běhu může i upravit dle momentální situace.

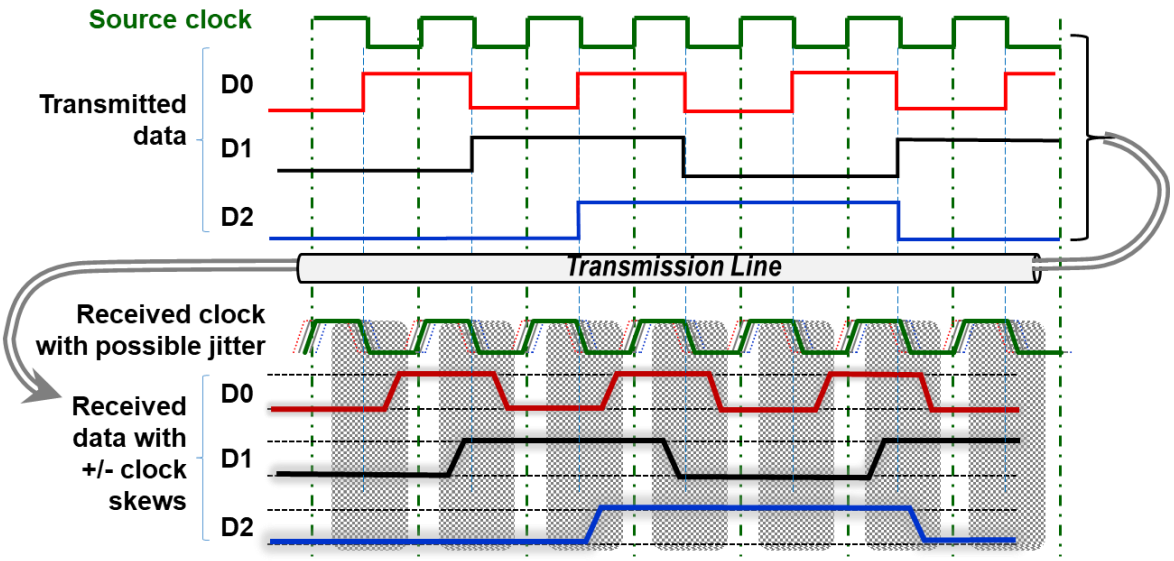
Podobné řešení není výhodné na delší vzdálenosti. Musíme jednak použít několik vodičů, což prodraží instalaci, a jednak se zde již projeví potíže se vzájemnou synchronizací paralelních signálů. Dochází ke zkreslení, a to jak jejich tvaru, tak jejich posunů v čase, a to i v rámci plošného spoje.

Opět připomeneme animaci, k níž jsme již odkazovali na str. 66 u modelu dynamického chování dvou invertorů. Nachází se na konci článku https://practicalee.com/transmission-lines/ o přenosových linkách a poskytne nám dobrou představu o tom, co se děje na vodičích, i důvod vzniku rušení hodin, které ukazuje snímek z osciloskopu.



Obrázek 183 - Osciloskop hodinového signálu 100 MHz na konci vodiče

Odlišnými dobami zpoždění na linkách, *propagation delays*, mohou data získat vůči hodinám různé posuny (*clock skews*), které zmiňoval i Obrázek 135 na str. 121, a to kladné i záporné, a frekvence přenosu se musí výrazně snížit. Na následujícím obrázku je téměř na svém maximu a zůstalo jen úzké místo, v němž jsou data ještě platná. Chybí zde větší časová rezerva. Když se u nějakého signálu zvýší zpoždění změnou různých parametrů, které ho vyvolávají, například teplotou, připojením dalšího zařízení, a podobně, budeme již číst chybná data.



Obrázek 184 - Různá clock skew na paralelní sběrnici

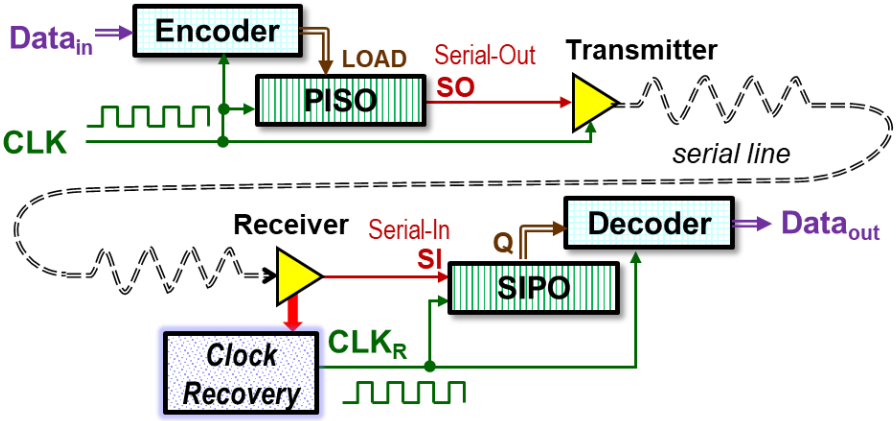
7.7.1 Sériový přenos dat

Sériovým přenosem redukujeme nejen počet vodičů, takže fyzické propojení je levnější, ale současně snižujeme i problémy se vzájemnou synchronizací. Přenášíme-li jen data a hodiny, můžeme lépe adaptovat posun přijatých hodin tak, aby se četlo bez chyb, a přenos urychlit.

Hodiny obecně nepotřebujeme. Někdy dají se rekonstruovat z vhodně kódovaných dat. Některé formy sériového přenosu je ani nepotřebují, např. NEC pro infračervené dálkové ovládání, viz dále. Pokud posíláme sériová data bez hodin, rychlost transferu lze ještě zvýší. Na větší vzdálenosti tak může 1-bitový signál přenášet data rychleji než 64-bitová paralelní sběrnice.

A chceme-li sériový přenos víc akcelerovat, využijeme několika sériových linek. Rozdělíme data na bloky, každý pošleme po jiné lince, a na přijímací straně je zase složíme. Například sériové propojení PCI Express 16x běžné používané u PC periférií (to se již chová víc jako síť než sběrnice) může datové bloky rozložit až na 16 nezávislých sériových linek.

Posuvné registry typu PISO (*Parallel-In Serial-Out*) se hodí k serializaci dat na straně jejich zdroje a přijatá data se zas nasouvají do posuvných registrů SIPO (*Serial-In Parallel-Out*). Rychlost obou posuvných registrů nezávisí na jejich délce, a tak dovolí i vysoké frekvence.



Obrázek 185 - Principiální schéma sériové linky

Data se před přenosem často konvertují, což v obrázku provádí blok *Encoder*. Například se k nim přidají hlavičky a konce paketů nebo start stop bity, aby se poznal začátek a konec. Doplňují se i pomocné informace a kontrolní kódy, aby se daly detekovat chyby v přenosu. Některé linky šifrují data či je konvertují, aby se dosáhlo optimální distribuce 0 a 1 a vysílání se vyvážilo, viz např. 8b/10 až128b/130b zavedené od PCIe 3.0[57], a i jiné metody.

7.7.2 NEC - dálková infračervená ovládání

Dálkové infračervené ovládání tvoří nedílnou součást široké plejády spotřebičů. Vysílá data klíčováním infračervené diody, jejíž světlo je navíc modulované frekvencí zhruba 38 kHz, aby se dalo odlišit od tepelného záření pozadí.

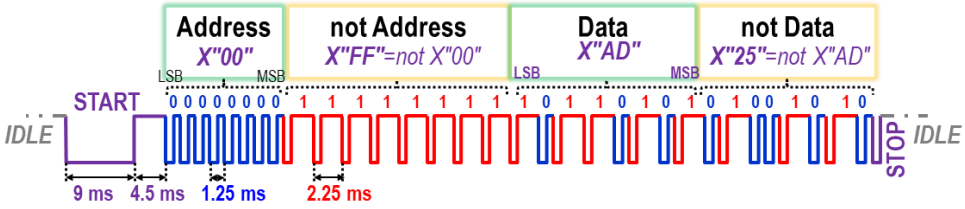
O příjem se nám postará již hotový obvod, který obsahuje i potřebnou optiku. Vybereme si např. IRM-V538/TR1 (s cenou cca 50,- Kč), který přijme a demoduluje signál a na své výstupu posílá logický signál kompatibilní jak s TTL, tak CMOS obvody. Obrázek vlevo je okopírovaný z katalogu výrobce, který najdete např. na Mouser .

[57] https://en.wikipedia.org/wiki/8b/10b_encoding a https://en.wikipedia.org/wiki/64b/66b_encoding

Data se kódují šířkovou modulaci normy podle NEC[58], v níž se vysílají skupiny 16 datových bitů v paketu [Address, Data], kde 8-bitové pole Data specifikuje kód stisknuté klávesy a pole Address specifikuje konstantní identifikátor zařízení.

Data se přenášejí duplicitně, podruhé negovaná, což se v NEC zamýšlelo k odhalení chyb v přenosu. Některá dálková ovládání ale používají tyhle bloky k přenosu přídavných údajů nebo odlišně kódují klávesy. Kvůli tomu je potřeba tolik ovladačů:-)

Následující obrázek ukazuje případ přenosu paketu [0x00,0xAD]. Serializace dat začíná od bitu s nejnižší váhou; 0xAD=10101101_2 se pošle jako->10110101.



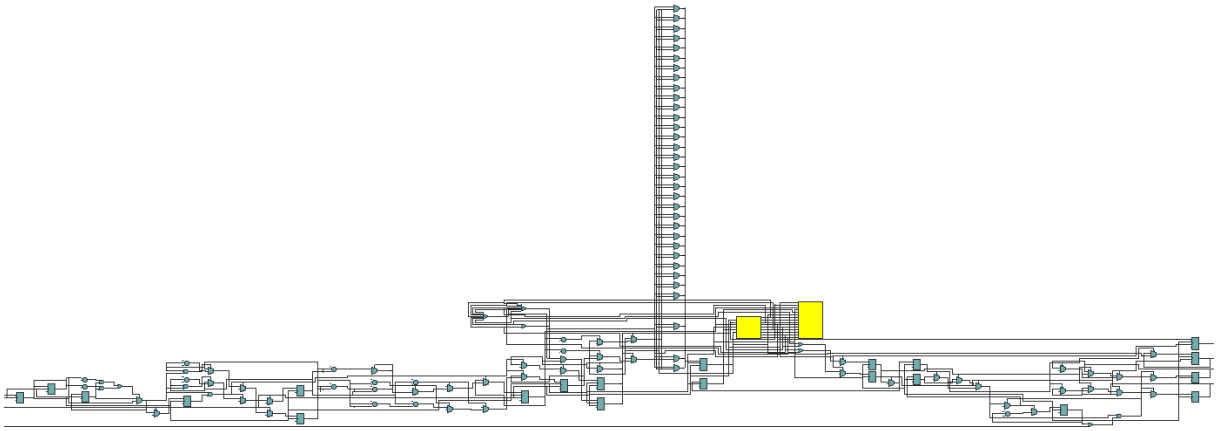
Obrázek 186 - IRDA signál normy NEC pro případ paketu [0x00,0xAD]

Samotné dekódování paketu lze provést měřením délky přijatého signálu v '0' a '1', a to vhodným čítačem. Začátek paketu snadno zjistíme jako pulz START mající zhruba 9 milisekund, který je v negativní logice, v níž je '1' reprezentována nižší úrovní. A proč *zhruba* 9 ms? Levnější dálková ovládání nebývají přesná a třeba zavést větší tolerancí.

Šířkově modulovaná data v 0 a 1 zase rozlišíme srovnáním délky signálu ve stavu '0' a '1'. Data 1 mají vyšší úroveň napětí víc jak dvakrát delší než nižší. Konec paketu určíme nejlépe tak, že uděláme posuvný registr dlouhý 33 bitů, a to doprava, neboť data se posílají od LSB. Při přijetí START začátku ho inicializujeme tak, aby měl '1' ve svém nejvyšším bitu a jinde '0'. Poté do něj nasouváme přijatá data. Jakmile se počáteční '1' posune do nejnižšího bitu registru, víme, že jsme přijali 32 bitů celého paketu.

A co je schéma přijímače NEC signálu? To je nepřehledné a zcela zbytečné. Nikdo ho nebude zapojovat. Ve VHDL má zhruba 256 řádek včetně deklarací a podobných komentářů a překladač zapojí obvod použitím 278 logických elementů, z nichž 138 nakonfiguruje jako registry DFF. Takové množství opravdu není na ruční zapojování.

Ale pokud jinak nedáte a toužíte zapojení vidět, budiž. Překladač ho nakreslil takhle:



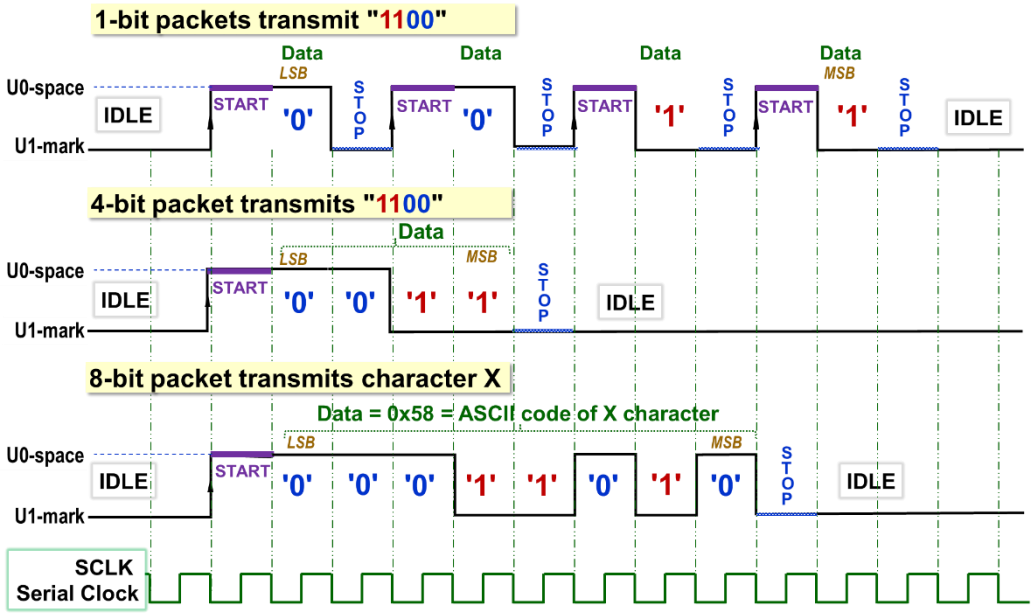
Uprostřed jsou multiplexory vybírající výstupní data z posuvného registru a žluté bloky reprezentují řídicí bloky konečných automatů FSM (*Finite State Machine*), které bývají běžným návrhovým stylem pokročilejších návrhů.

[58] V době psaní publikace existoval popis NEC na https://www.vishay.com/docs/80071/dataform.pdf

7.7.3 Start-Stop sériový přenos dat

Při sériovém přenosu potřebujeme poznat, že začíná další blok dat, a to i v případě, že přenášíme samé '0' nebo '1'. Lze použít několik různých metod. Uvedeme jen jednu z nich, poměrně jednoduchou, a to vložení bitů Start a Stop. Vždy se volí jejich opačné polarity, aby docházelo ke změně stavu linky. Lze tedy mít Start='0' a Stop='1', nebo Start='1' a Stop='0'. Závisí na normě, pokud tohle specifikuje, či na nás, co si sami zvolíme.

Obrázek dole ukazuje nahoře situaci, kdy se přenášejí 1-bitové bloky v negativní logice a máme Start='0' a Stop='1'. Náběžná hrana udává Start a uložíme bit za ním. Přenos jde od nejnižšího bitu k nejvyššímu. Pochopitelně, jiné linky ho mohou mít obráceně od nejvyššího k nejnižšímu bitu, tedy od MSB to LSB, třeba I2C, která bude dále.



Obrázek 187 - Princip sériového přenosu se Start a Stop bitem

Pakety o délce 1-bit nejsou efektivní, použili jsme je jen k vysvětlení principu. Data se vždy sdružují do větších bloků. Obrázek ukazuje délku dat i 4 a 8 bitů. Hodiny SCLK lze vynechat a posílat jen data. Začátek komunikace poznáme změnou stavu linky po delším stavu IDLE.

Kódování i dekódování provádí modul nazývaný UART (*Universal Asynchronous Receiver / Transmitter*). Pokud se nepřenáší SCLK, rekonstruujme je třeba tak, že použijeme referenční frekvenci fref , která bude vyšší než SCLK přenášených dat, typicky 16 x. Dělíme ji konstantou N, zvolenou tak, aby platilo SCLK = fref/N, což dovede i Johnsonův čítač, viz Obrázek 180 na str. 144. Při každé změně stavu linky se čítač nuluje, čím synchronizujeme rekonstruované hodiny. Přijatá data se pak vzorkují některým fázově posunutým výstupem Johnsonova čítače.

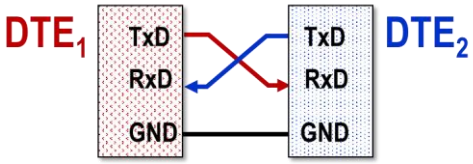
Chceme-li přenášet bloky o délkách stovek bitů, lze paket prokládat pomocnými bity, což zaručí občasnou změnu stavy linky i v případě dlouhých bloků 0 a 1. Jde o *Bit Stuffing* metodu, jejíž název se v odborných textech nepřekládá do češtiny. Znamená vkládání pomocných bitů, které mají opačnou hodnotu než předcházející bit. Například na sběrnicích USB či CAN se po 5 bitech stejné hodnoty musí vždy vložit opačný. Přijímací strana takové pomocné bity ignoruje, využije je jen k synchronizaci svých rekonstruovaných hodin.

Některé jednodušší komunikační protokoly používají i pravidelný *Bit Stuffing*, který je snazší na kódování, a vkládají pomocný bit třeba jako každý osmý.

7.7.4 RS-485 a RS-232

Linka RS-232 je známá jako COM port a kdysi bývala u všech počítačů. Dnes ustupuje do pozadí jako externí sběrnice kvůli své nízké přenosové rychlosti, ale stále se prodávají levné převodníky z USB na RS232, takže se na ni snadno připojíme. Je navíc jednoduchá, jak obvodově, tak na dekódování, a tak se mnohdy přidává do zařízení jako záchranný servisní vstup, který mají jak pevné disky, tak procesorové systémy. Když se vše ostatní pokazí, lze využít aspoň ji. RS-232 používá Start Stop bity a přenáší bloky dat o konfigurovatelné délce od 5 do 8 bitů bez hodinového signálu. Vysílá '0' jako kladné napětí a '1' jako záporné.

Má jednosměrné oddělené signály pro vysílání TxD a RxD pro příjem a zemnicí vodič. Vystačí tak se třívodičovým kabelem. RS-232 norma zahrnuje i přídavné řídicí signály DTR, RTS, CTS, DSR a RI, ale jejich roli dnes limituje to, že většina převodníků z USB na RS-232 umí jen datové signály TxD a RxD.



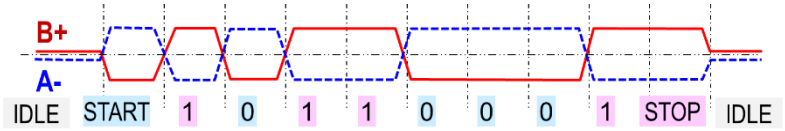
Obrázek 188 - Propojení přes RS-232

Její přenosová rychlost je historickým reliktem, který vychází z násobků rychlosti 200 baudů dávných telefonních modemů. I u RS-232 se rychlost tradičně udávala v baudech [Bd], tedy v modulační rychlosti, jak se občas najde ve starších publikacích. Pro kódování, které používá RS-232, ale platí, že 1 baud = 1 bit/s.

Maximum u RS-232 je 115200 bitů/s na vzdálenost 8 metrů, avšak za předpokladu, že vysílací budiče TxD používají napětí na horním konci normy RS-232 a vysílají '0' třeba jako +12 V a '1' jako -12 V. Žel běžné vyráběné převodníky z USB na RS-232 často volí kvůli hardwarové jednoduchosti dolní konec normy a '0' vysílají jako +3 V a '1' jako -3 V. Na základě našich zkušeností lze takové použít sotva do rychlosti 1200 bitů/s do vzdálenosti zhruba jednoho metru.

Linka RS-485 je pokročilejší variantou a často se používá k propojení různých zařízení jak v domech, tak ve výrobních závodech. Vkládá rovněž start a stop bity, ale na rozdíl od RS-232 vysílá data jako diferenční signály označované jako B+ a A-, přičemž na B+ má v klidu vyšší hodnotu než na A-. Stav '1' je kladný rozdíl mezi B+ a A-, stav '0' zase záporný.

Její diferenční signály potlačují vliv napěťových posunů i souhlasného rušení, tj. rušení stejného pro obě strany. Napojená zařízení se tak mohou více lišit ve svých zemích potenciálech, tedy referenčních bodech 0 V. Existují i její varianty s galvanickou izolací.



Obrázek 189 – Demonstrace kódování na RS-485

Obrázek nahoře ukazuje paket, který pro ilustraci přenáší jen byte. Zpravidla je delší a skládá ze sekvencí několika bytových znaků. Synchronizace rekonstruovaných hodin se pak snadno udrží, použijeme-li Bit Stuffing zmíněný na předchozí stránce. RS-485 může mít jak dvě linky vysílací a přijímací, tak pracovat v poloviční duplexu, kdy oba směry sdílejí jedinou linku. Každé zařízení však musí rozeznat, že smí vysílat. RS-485 norma nespecifikuje komunikační protokol, ten závisí na naší volbě. Komunikační čas může pak přidělovat zařízením na lince například arbiter, který posílá dotazy či přiděluje časová okna, což bývá časté řešení.

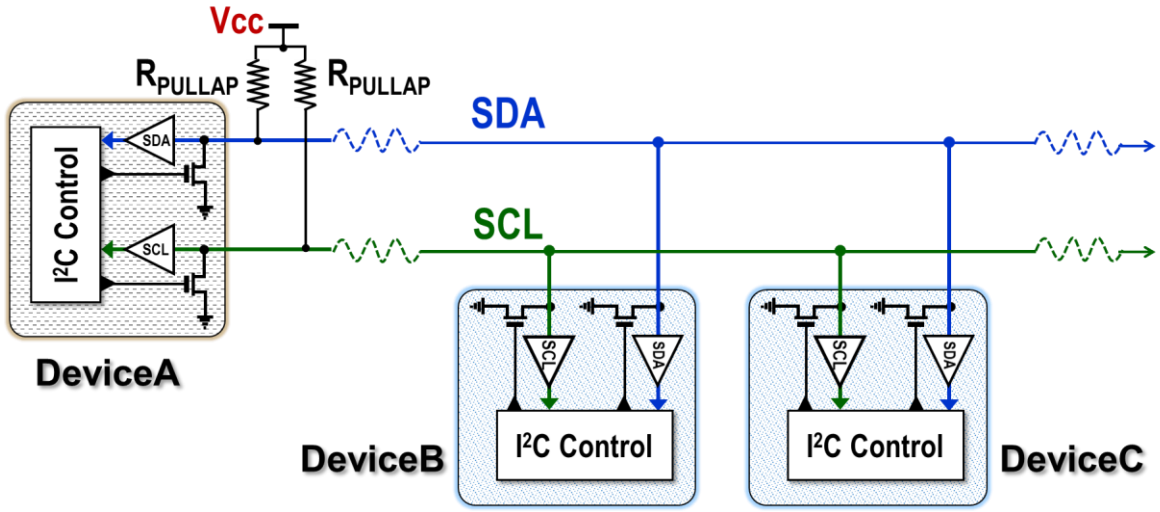
Metoda CSMA/CD (*Carrier-Sense Multiple Access with Collision Detection*), která je známá ze sítí Ethernet, se tady využívá méně pro vyšší hardwarovou složitost. Pokud se aplikuje, každé zařízení si musí měřit stav linky a poslat data jen tehdy, když je volná. Pokud se shodou okol-

ností připojil i někdo druhý, oba detekují se zkrat, odpojí se a za náhodně zvolenou dobu zopakují své pokusy. Existují ověřené algoritmy ke stanovení doby čekání, které zaručí, že každé zařízení časem odešle svá data.

Pro příjem i vysílaní na fyzické vrstvě se běžně prodávají integrované obvody označované RS-485 Interface, cena kolem 100,- Kč. Dle normy by RS-485 měla přenášet až rychlostí 10 Mbit/s na 15 metrů, ale lze dosáhnout až 50 Mbit/s. Na delší vzdálenost se předpokládá snížení frekvenci změn dat, která jsou již více zkreslená. Jako maximum se udává 1200 m při přenosové rychlosti 100 kilobitů/s.

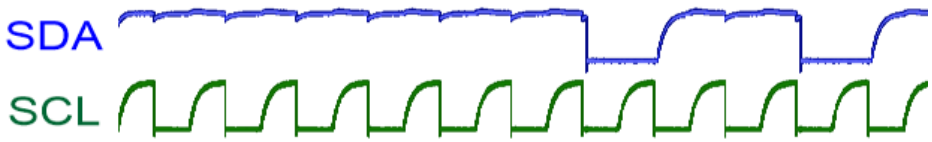
7.7.5 I2C a SPI sběrnice

I2C sběrnice (*Inter-Integrated Circuit*) nabízí nemalou výhodu v tom, že k její obsluze existují řešení *Open-Source*, a to jak ve Verilogu, tak ve VHDL. Používá dva signály, ale oba jako obousměrné a posílané po sběrnicích s otevřeným kolektorem[59], a to hodiny SCL a data SDA. Každé zařízení může být jak vysílačem, tak přijímačem, i zdrojem hodin.



Obrázek 190 -Příklad obvodového řešení obousměrné I2C

Otevřené kolektory mívají R_{PULLUP} odpory zpravidla v rozmezí od 2 do 10 kΩ. NMOS transistor na výstupu sice okamžitě sepne napětí do 0 V, ale to se pomaleji vrací od GND do Vcc.



Obrázek 191 – Osciloskopem změřené průběhy napětí na jedné lince I2C

Ovladače I2C mohou linku sdílet metodami diskutovanými u RS-485, při nichž se využívají i toho, že linka s otevřeným kolektorem se ze svého principu nepoškodí, ani když ji sepne více zdrojů najednou, a to i dlouhodobě. Každé zařízení může tak zabrzdit hodiny v '0', když potřebuje víc času.

Když někdo chce vysílat, vyčká, až bude linka volná, tedy delší dobu v '1', pak začne START bitem ='0'. Za ním vysílá adresu cíle, která se záměrně předává od nejvyššího bitu. Pokud současně s ním zahájilo přenos i jiné zařízení, detekuje se to zpravidla už při ní. Když totiž někdo nastaví bit na '1', ale linka zůstane v '0', pak ví, že současně s ním vysílá jiná jednotka.

[59] Viz Wiki: https://en.wikipedia.org/wiki/Open_collector

Musí se ihned odpojit a čekat na opětovné uvolnění linky. Podobná arbitrace se někdy nazývá distribuovanou, danou pravidly protokolu. Tady při neshodě '0' a '1' vítězí '0'. Pokud se to například přihodilo u adresy cíle, pak kdosi druhý posílal data nadřazené nižší adrese. Jejich přidělením lze přirozeně vytvořit hierarchii.

I2C se hodí k propojení na vzdálenosti v rámci jednoho plošného spoje pro zabudované periférie, jejichž údaje se mění relativně pomalu. Dovolí přenosovou rychlost jen 100 kilobitů/s v pomalém režimu na vzdálenost 1 metru a 400 kilobitů/s v rychlém režimu k blízkým zařízením, ale existují i varianty na 3,4 megabity/s, ovšem vyskytující se ojediněle.

K přenosu na 10 metrů se musí frekvence SCL hodin snížit na extra-pomalou variantu 10 kilobitů/s. Rychlost někdy omezí i nevhodně provedený výstup některého zařízení.

I2C se snadno rozšiřuje a na své dva signály může paralelně napojit až 128 zařízení. Bývá běžná u mnoha vyráběných komponent. Zmíníme například dotykový LCD panel, snímač akcelerace či magnetického pole, modul GPS navigace, teplotní čidla, a podobně. Je skoro ve všech mobilech.

Existuje i ve variantě SMbus pro komunikaci mezi zařízeními na motherboard (základní desce počítače), kde se po ní přenášejí údaje o teplotách a běhu ventilátorů. I paměťové moduly DIMM po ní posílají své časování a teplotní údaje.

Mnohde se používá i několik oddělených I2C sběrnic. Audio obvody se často dávají na samostatnou linku, aby využily její maximální rychlost, či použijí svou specializovanou linku I2S (Inter-Integrated Circuit Sound), která obsahuje navíc i vodič výběru slova. Navzdory podobnému názvu je však v mnohém odlišná, její data nepoužívají start stop bity, ale pulzně šířkovou modulaci k přenosu hodnot audio signálu.

SPI sběrnice (*Serial Peripheral Interface*) má oddělené datové vodiče pro vysílání, port SDO, a pro příjem, port SDI, a přenáší hodiny na portu SCK. Vyžaduje samostatné výběrové linky CSi (*Chip Select*) pro každé zařízení. SPI dosahuje běžně 5 megabitů/s (díky snazší selekci cílového zařízení než u I2C) a její rychlá varianta SPI (Flexcomm8) dokonce až 50 megabitů/s.

SPI má jednodušší komunikační protokol než I2C. Jedno zařízení zpravidla vládne všem a přiděluje čas. Linka je ale elektricky složitější, má nejméně dvojnásobek vodičů.

7.7.6 JTAG

JTAG, pojmenovaný podle *Joint Test Action Group*, která jej vytvořila, je průmyslovým standardem pro ověřování návrhů a testování desek s plošnými spoji po výrobě. Rozhraní se připojuje k portu TAP (*Test Access Port*) na čipu, který implementuje stavový protokol pro přístup k sadě testovacích registrů, často na bázi posuvných registrů, které sbírají data i je nahrávají i do části FPGA. Má komplikovanější zapojení a dosahuje rychlost je 70 až 100 kilobitů/s.

Sběrnice JTAG je základním vstupem FPGA pro ladění i jejich konfigurace. Pokládá se za otevřený nástroj pro ladění.

Chceme-li ji však používat ze svých programů, neuškodí si ověřit, zda můžeme, a nečeká nás nemilé překvapení. Přístup k JTAG může totiž zablokovat vývojová deska FPGA, k níž sice dostanete ovladač, ale bude použitelný jen z programů výrobce.

7.7.7 USB a Ethernet

Zatím jsme vynechali USB a Ethernet. Některé typy FPGA obsahují v sobě vše potřebné k jejich obsluze, k jiným lze dokoupit externí obvody. Potřebujeme však řešení, k němuž výrobce za rozumnou cenu prodá i hotové zapojení na straně FPGA standardně označované zkratkou IP, *Intellectual Property*. Nezaměňujte ho s IP (*Internet Protocol*) adresou.

Bez IP rozhraní zpravidla nepoužijeme, i když ho bude mít FPGA či vývojová deska. Naštěstí existují i otevřená řešení, které lze s menší či větší námahou adaptovat.

Operační systém počítače sice obslouží ethernetové či USB rozhraní, ale totéž je nutné i uvnitř FPGA obvodu. Relativně snadno se z něho napojíme na fyzickou vrstvu jak u Ethernetu, tak u USB. Prodávají se k tomu vhodné obvody. Obě linky ale zapouzdřují přenosy dat do vyšších komunikačních vrstev, OSI (*Open Systems Interconnection*) u Ethernetu a v USB existuje jeho vzdálená analogie *Host+Device Stacks*.

Chceme-li například u Ethernetu pracovat s protokolem TCP/IP, pro který existují na straně počítače běžně dostupné *Network Sockets*, pak si náš obvod musí pro připojení na síť vyžádat svou IP adresu od DHCP (Dynamic Host Configuration Protocol) serveru. Vymění si s ním zprávy, v nichž mu pošle svoji MAC (Medium Access Control) přidělenou výrobcem. Od něho dostane na výběr z několika IP adres, ale smí tu zvolenou užívat jen po omezený čas. Poté musí opět zopakovat komunikaci a vyžádat si její prodloužení.

Existuje sice možnost, že se na DHCP nastaví mapování MAC na pevnou IP, ale tohle není vždy možné a jde o dost nepřenositelný postup.

Chceme-li se tedy napojit na USB či Ethernet, raději si vybereme FPGA, které obsahuje mikroprocesor, případně k němu existuje Soft-core processor (tj. procesor napsaný v nějakém HDL jazyce), k němuž se dá IP rozšíření dokoupit za rozumných podmínek. Psát vše od nuly, to by nám zabralo hodně času.

7.8 Co vše jsme nezmínili?

Učebnice si kladla za cíl uvést do obvodové techniky, a tak zůstala nezávislá na HDL jazyce, aby nevnášela další téma. V závěru sice zmínila HDL popisy, ale jako motivaci pro čtenáře.

Výklad se opíral o schémata, která zůstávají názorná jen do určité své složitosti. Rychle ztrácejí přehlednost s nárůstem součástek, viz obousměrný posuvný registr, Obrázek 181 na str. 144. Výklad ještě složitějších obvodů by se musel pak nutně omezit jen na přehledu vlastností podobně jako u přijímače IRDA signálu dálkového ovládání.

A o obvodech nejede jen číst. I v programovacích jazycích nepostoupíme za určitou mez pouhým studiem jejich syntaxe, ale jen tvorbou vlastních programů. A obvod dnes zapojíme rychle jen přes HDL jazyk.

Co vše jsme přenechali učebnicím HDL jazyků? Je toho dost. Zmíníme jen hlavní položky, aby čtenáři věděli, že existují, až je budou potřebovat.

- V kapitole o posuvných registrech jsme nepopsali velmi zajímavé zpětnovazebné posuvné registry LFSR (*Linear-feedback Shift Registers*). Mají sice složitější teoretické zázemí, ale velmi jednoduchý HDL kód a jen o málo větší schémata. Používají se jako náhodné generátory k šifrování dat či automatickým testům obvodů. Umí velmi efektivně počítat i na vysokých frekvencích bezpečnostní CRC kódy a ověřovat je.
- Nezmínili jsme ani konstrukce dlouhých posuvných registrů pomocí pamětí. Do nich lze pak například uložit celý obrazový řádek a provádět třeba jeho filtrování a konvoluce obrazů v reálném čase.
- Nezařadili jsme také konečné automaty FSMs (*Finite State Machines*), které se často používají v HDL jazycích jako stavební kameny složitějších obvodů.
- Hodí se i znát RDY-ACK protokol pro asynchronní přenos dat mezi různými FSMs.
- Nevysvětlili jsme ani speciální případy konečných automatů jako konečné akceptory k rozeznání sekvencí.
- Zcela jsme vynechali mikroprogramové řadiče a procesory, které lze vytvořit i v HDL jazyce jako tzv. *Soft-core processors*.
- Skoro každá elektronika obsahuje fázové závěsy k násobení frekvencí označované jako PLL (*Phased-Locked Loop*), ale nejsou vysvětlené v naší učebnice.
- Neuškodí ani znalosti o ochraně vstupů a řešení výstupů, jako třeba připojení normální odporové a indukční zátěže, ale i o obsluze tlačítkové klávesnice, a dalších témat. Patří sem i seznámení s nábojovou pumpou ke konverzi stejnosměrného napětí na vyšší bez nutnosti převodu na střídavé napětí. Ta někdy vzniká nechtěně a ohrožuje obvod.

Chybí i mnoho jiného. Logické obvody představují široký obor a bylo by potřeba několik tisíc stránek k vyčerpávajícímu popisu jejich hlavních prvků a konstrukcí.

8 Závěr

Učebnice obsahuje jen obvodové základy nutné k tvorbě v HDL jazycích. Její možné vylepšování necháme na případné budoucí verze.

Čím pokračovat? Začneme výběrem vhodného HDL jazyka, v němž budeme popisovat obvody. Z hlediska lektora lze poradit přísnější VHDL, který zůstává populární v Evropě. V něm si lépe osvojíme správný styl. Pokud projdeme přes veškeré jeho striktní typové kontroly, někdy sice zdánlivě „únavné“, ale dávají nám větší naději, že náš návrh bude fungovat. Ve Verilogu udělají začátečníci snadno chybu a dlouho se moří s jejím odstraněním.

Až si svůj styl vypilujeme na četných příkladech, které sami vyřešíme, poté se rozhodneme, zda budeme pokračovat ve VHDL. Případný přechod na volnější a textově úspornější Verilog bývá snadný, dle znalců z praxe, což prý opačně většinou neplatí. Kdo se bude chtít víc věnovat obvodům, stejně potřebuje znát oba jazyky. Novější SystemVerilog je též velmi nadějnou alternativou. Inspiroval se nejen Verilogem, ale i VHDL a jazykem C++. IEEE ho normalizoval v roce 2005 a znovu v roce 2009, kdy se spojil s jazykem Verilog.

Pokud čtenář není profesionál, kterému firma platí vývojové nástroje, pak doporučujeme následující postup:

- Napřed se podíváme, jaké existují dostupné bezplatné vývojové nástroje.
- V našem předmětu LSP (Logické systémy a procesory vyučovaném na ČVUT-FEL) jsme zvolili vývojové prostředí Intel® Quartus® Prime Lite Edition Design Software, které pořád dovoluje kreslit obvody i v symbolických schématech, vhodných pro počáteční kroky. Obsahuje i simulátor Altera-ModelSim, avšak jen do verze 20.1.1 (z listopadu 2020). Navíc ve své neplacené verzi neemuloval zpoždění logiky, a tak dával jen orientační výsledky.
- Chybějící volnou verzi ModelSim lze ve VHDL plně nahradit freeware simulátorem GHDL, který již od roku 2015 existuje jak pro Linux tak Windows. Jeho vývojová skupina je stále aktivní v době psaní této publikace (duben 2025). Naši studenti si ho oblíbili tak, že větší část návrhů dělají v něm. GHDL simuluje stejně rychle jako ModelSim. Změřeno:-) Celý cyklus, který bereme od okamžiku uložení opraveného VHDL souboru až po zobrazení výsledku jeho simulace, provedeme pomocí snadno ovladatelného GHDL i mnohem pohodlněji, a kvůli tomu i za polovinu času, než v případě užití nástroje ModelSim.
- Bezplatné verze vývojových prostředí často omezují nabídku typů FPGA, které v nich smíme použít. Podle toho si pak zvolíme vývojovou desku, která se nám nejen zamlouvá, ale také musí obsahovat i podporovaný FPGA obvod.
- K nahrání do FPGA obvodu musíme ale kód přeložit vývojovým prostředím výrobce. Ověříme si tedy, jaké konstrukce našeho HDL jazyka podporuje jeho zvolená bezplatná verze. Většina z nich dovolí třeba jen VHDL 1993 a Verilog, ale jen něco z VHDL-2008 a SystemVerilogu. Vše ostatní zůstává dostupné jen uživatelům placené verze.

Pro svou prvotní HDL inspiraci lze s výhodou využívat i cizí kódy, které se jen mírně zeditují. Ty doporučujeme vybírat raději z knih, které prošly recenzí. Profesionální firmy málokdy publikují své návrhy. Většina kódů obvodů, které volně najdete na webech, byla vytvořená začínajícími návrháři. A někteří použili dost krkolomné konstrukce, které akorát komplikují obvod, a tak se hodí pokaždé srovnat několik různých nalezených řešení.

Na závěr popřejeme čtenářům hodně zajímavých a úspěšných návrhů.

9 Seznam číslovaných obrázků a tabulek

Poznámka: Mnohé další obrázky se vkládaly bez pojmenování, pokud jen tematicky rozšiřovaly jiné ilustrace.